

Software Clone Management Towards Industrial Application

Edited by

Rainer Koschke¹, Ira D. Baxter², Michael Conradt³, and James R. Cordy⁴

1 Universität Bremen, DE, koschke@informatik.uni-bremen.de

2 Semantic Designs – Austin, US, idbaxter@semdesigns.com

3 Google – München, DE, conradt@google.com

4 Queens University – Kingston, CA, cordy@cs.queensu.ca

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 12071 “Software Clone Management Towards Industrial Application”. Software clones are identical or similar pieces of code or design. A lot of research has been devoted to software clones. Unlike previous research, this seminar put a particular emphasis on industrial application of software clone management methods and tools and aimed at gathering concrete usage scenarios of clone management in industry, which will help to identify new industrially relevant aspects in order to shape the future research.

Talks were presented by industrial participants and working groups were formed to discuss issues in clone detection, presentation, and refactoring. In addition we developed a unified conceptual model to capture clone information required to support a common notion of clone data and for interoperability to foster exchange of data among researchers and tools in practice. The main focus of current research is clones in source code – therefore, we also looked into ways of extending our research to other types of software artifacts. Last but not least, we discussed how clone management activities may be integrated into the process of software development.

Seminar 12.–17. February, 2012 – www.dagstuhl.de/12071

1998 ACM Subject Classification D.2.7 Distribution, Maintenance, and Enhancement, D.2.13 Reusable Software, K.5.1 Hardware/Software Protection

Keywords and phrases Software clones, code redundancy, clone detection, redundancy removal, software refactoring, software reengineering, plagiarism detection, copyright infringement, source differencing

Digital Object Identifier 10.4230/DagRep.2.2.21

1 Executive Summary

Rainer Koschke (University of Bremen, DE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Rainer Koschke

Software clones are identical or similar pieces of code or design. They are often a result of copying and pasting as an act of ad-hoc reuse by programmers. Software clone research is of high relevance for software engineering research and practice today. Several studies have shown that there is a high degree of redundancy in software both in industrial and open-source systems. This redundancy bears the risk of update anomalies and increased maintenance effort.



Except where otherwise noted, content of this report is licensed under a Creative Commons BY-NC-ND 3.0 Unported license

Software Clone Management Towards Industrial Application, *Dagstuhl Reports*, Vol. 2, Issue 2, pp. 21–57

Editors: Rainer Koschke, Ira D. Baxter, Michael Conradt, and James R. Cordy



Dagstuhl Reports

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Many techniques exist that try to detect clones. Some of them are already available in open-source (e.g., PMD) as well as commercial tools (e.g., CloneDr). There are also lines of research in clone detection that evaluate these approaches, reason about ways to remove clones, assess the effect of clones on maintainability, track their evolution, and investigate root causes of clones. Today, research in software clones is an established field with more than 100 publications in various conferences and journals.

The purpose of this seminar was to solidify and give shape to this research area and community. Unlike previous similar events, this Dagstuhl seminar put a particular emphasis on industrial application of software clone management methods and tools and aimed at gathering concrete usage scenarios of clone management in industry, which will help to identify new industrially relevant aspects in order to shape the future research. Research in software clones is very close to industrial application. Among other things, we focused on issues of industrial adoption of our methods and tools.

To achieve our goals, we invited many participants from industry. We managed to reach a percentage of about 30% industrial participation. Talks were given mostly by industrial participants who shared their experiences with us and posed their problem statements. Academic participants were allowed to give a talk if their talk had a clear focus on industrial experiences, needs, problems, and applications of software clone management and related research fields. The focus, however, was on interaction in form of plenary discussions and smaller working groups. The topics for working groups were gathered by clustering issues the participants wanted to discuss at the seminar. The seminar wiki was used intensively to record the results of the working groups. This agile format was very much appreciated by the participants.

The following working groups were formed:

- **Detection/Use cases:** This working group discussed issues in detecting clones. Because there are already many clone detectors, the focus of this working group was to gather use cases for these. The particularities of a use case dictates what kinds of features a suitable clone detector should have.

The group's result was a list of different use cases for clone detection and an enumeration of distinct features a clone detector should have to support the respective use case. An overview of known limitations and issues of actual clone detectors is also provided along with some research questions oriented towards the improvement of clone detection techniques.

- **Presentation:** Because clone detectors typically find many clones in large systems, the user faces a huge amount of data he or she needs to make sense of. Visualization is a means of presenting large and complex data that takes advantage of a human's ability for visual pattern matching. This working group dealt with presentation issues of clone information. Again, use cases were enumerated because suitability of visualization is task dependent.

The group connected the identified use cases with different existing types of software visualization suitable for these.

- **Interoperability:** To foster collaboration among researchers it is helpful to build interoperable tools. Then, for instance, the result of one researcher's clone detector could be fed into the visualization tool of another researcher. Interoperable tools are also needed to serve practitioners' diverse needs.

This working group created a common model to represent clone information that addresses the needs of a wide range of use cases in research and practice.

- Refactoring: Contrary to the abundance of available clone detectors, there are relatively few tools that help in removing clones. The purpose of this working group was to consider the mechanics and utility of forming clone abstractions and achieving clone refactoring. The group identified various means of eliminating clones that are either provided by the languages the clones are written in or by abstraction outside of the language (e.g., code generation). It also delved into managerial aspects of clone refactoring and particularities of clones in software product lines.
- Clone management (process): Clone management is the set of activities to detect, track, assess, handle, and avoid clones. This working group went into the matter of where clone management may play a role in the development and maintenance process. The group discussed how clone analysis fits into the overall software development process (requirements engineering, development, testing, after deployment). They broached the issue of relation of code search and clone detection and how clone detection could be used in recommender systems.
- Provenance and clones in artifacts that are not source code: Most research in software clones focuses on source code, but as it has been shown by several researchers, clones can also be found in other software artifacts such as models and requirement specifications. This working group investigated needs to extend our research into these fields and the particularities of these fields with respect to clone detection. In addition to that, this working group dealt with provenance of clones, that is, the question where the clone comes from. Although the issues of provenance and clones in other artifacts appear to be largely independent, this working group worked on them jointly for organizational issues. The group elaborated how clones could be detected and handled in binaries, models, and bug reports.

For the remainder of this report, it is important to know the following current categorization of clones:

- Type-1 clone: Identical fragments only.
- Type-2 clone: Lexically identical fragments except for variations in identifiers, literals, types, whitespace, layout, and comments
- Type-3 clone: Gapped clones, that is, clones where statements have been added, removed, or modified.
- Type-4 clone: Semantic clones, that is, clones with similar semantics but different implementations in code.

2 Table of Contents





Executive Summary	
<i>Rainer Koschke</i>	21
Overview of Talks	
Reducing ROM Consumption by Unifying Clones in Safety-Critical Software Systems <i>Gunther Vogel</i>	25
Code Clone Detection Experience at Microsoft <i>Yingong Dang</i>	25
Clones @ Bosch <i>Jochen Quante</i>	25
Semantic Designs' experience <i>Ira Baxter</i>	26
Clone Detection @Google <i>Michael Conradt</i>	27
Industrial Clone and Malware Detection <i>Andrew Walenstein</i>	27
Where is the “business” case for software clones? <i>Serge Demeyer</i>	27
A Controlled Experiment on Software Clones <i>Jan Harder</i>	28
Issues in detecting license violations <i>Armijn Hemel</i>	29
Good and Evil clones <i>Angela Lozano</i>	29
Improving Software Architecture – Role for Software Clones <i>Ravindra Naik</i>	29
Working Groups	
Working group on clone detection <i>Thierry Lavoie</i>	31
Working group on clone presentation <i>Sandro Schulze, Niko Schwarz</i>	35
Working group on interoperability <i>Cory Kapser, Jan Harder, Ira Baxter, Douglas Martin</i>	38
Working group on refactoring <i>Ira Baxter</i>	43
Working group on clone management (process) <i>Jens Krinke</i>	51
Working group on provenance and clones in artifacts that are not source code <i>Serge Demeyer</i>	53
Participants	57

3 Overview of Talks

The seminar asked for lightning talk (a short and intensive talk, typically 5–15 minutes long) on industrial experiences, needs, problems, and applications of software clone management and related research fields. The goal of such talks was to trigger plenary discussions on open, industrially relevant issues rather than to provide found solutions. These problem statements were used during the seminar as work items for the working groups.

3.1 Reducing ROM Consumption by Unifying Clones in Safety-Critical Software Systems





Gunther Vogel (Robert Bosch GmbH, DE)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Gunther Vogel

This talk summarized experiences with clone management during software development of airbag software at Robert Bosch GmbH.

3.2 Code Clone Detection Experience at Microsoft





Yingong Dang (Microsoft Research Asia, CN)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Yingong Dang

This talk presented a clone detector developed at Microsoft Research Asia and some of the experiences gathered in using it within Microsoft.

3.3 Clones @ Bosch


Jochen Quante (Corporate Research at Robert Bosch GmbH, DE)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Jochen Quante

This talk explained clone detection/management activities at Bosch Corporate Research. It stated reasons for clones in Bosch automotive software and discussed their pros and cons. Beyond source code, the talk delves into clones in models of model-driven development. Finally, challenges from Bosch's perspective were listed.

3.4 Semantic Designs' experience

Ira Baxter (Semantic Designs, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Ira Baxter

The Dagstuhl Seminar focused on industrial application of clone detection and management methods, tools, and consequences. Ira Baxter of Semantic Designs built one of the earlier clone detection tools, CloneDR, based on matching abstract syntax trees, and has offered this tool as a commercial product for a decade. This talk sketched Semantic Designs' scalable program analysis and transformation infrastructure, DMS, and described how CloneDR leveraged the DMS machinery to implement an industrial strength clone analysis tool. DMS's ability to handle many languages, and its regular architecture, enables CloneDR to be implemented as a product line parameterized by language front ends; clone detectors for new languages can be constructed in about 15 minutes of effort once a language front end for DMS is completed. Notably, across many different computer languages (C, C++, COBOL, Java, Python, PHP and a variety of others), CloneDR consistently finds 10-20% of the code is cloned. An "impossible software growth" curve with negative growth over time was exhibited for a customer company applying clone removal manually but regularly based on CloneDR analyses. The talk exhibited the HTML report generated by CloneDR, including summary pages and pages shows specific clones. It was a surprise to the author that CloneDR's presentation of parameterized clones and the bindings for the parameters was not standard.

Experience with clone detection has shown variety of nonstandard uses: a) cherry picking of very large clones is easy and valuable; b) isolating a clone makes the code block easier to understand than when it exists in its surrounding code context, c) if bindings of a clone parameter are of inconsistent conceptual types, the clone is often buggy; d) clone abstractions form the basis for domain concepts and realizations, e) there is considerable utility in applying clone detection to DSLs who themselves often have weak abstraction abilities, to determine the kinds of abstractions that might be useful for that DSL. Finally, the complement of clone detection ("what code is the same") leads to a focus on "what code is different", showing a connection between the machinery needed for clone detection and "smart differencing" over ASTs. Semantic Designs has built a product line "Smart Differencers" following this philosophy, and using much of the same machinery. It was suggested that CloneDR might be useful in constructing product lines from forked code bases.

Technology application has proven difficult. The business case for clone detection and removal is not yet clear and management will generally not commit with such business case. Programmers also resist; a) while it is well known that code contains many clones, revealing them shows often embarrassing cloning on the part of individual programmers, b) the absence of IDE integration in their favorite IDE is a significant stumbling block; IDE integration must become a product-line; c) the resistance to "not a free tool" is astonishing considering the value of programmer time. Better models of ROI need to be developed to overcome guesswork about value.

Future developments include better clone detection but perhaps more importantly actual clone removal. Removal requires selection of a specific abstraction method for each subset of a clone set, chosen from both language-supported capabilities (subroutines, macros, etc.), and extra-language capabilities such as general macro processors, configuration conditionals and even wholesale file replacement. The variety of choices here, and the sheer volume of clones to be potentially removed, is a barrier to application because of the level of user effort required. Actual removal requires the ability for an engine to reliably modify the code





according to the abstraction type; as a program transformation engine, DMS is peculiarly well placed for this task, and there are few other practical alternatives. Perhaps integration into an IDE, with “single click to orbit” removal of clones will change to perceived and actual value.

References

- 1 I.D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program Transformation for Practical Scalable Software Evolution. in *International Conference on Software Engineering*, pp. 625-634, 2004.
- 2 I.D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *International Conference on Software Maintenance*, IEEE Press, 1998
- 3 <http://www.semanticdesigns.com>. Semantic Designs Company Website.

3.5 Clone Detection @Google

Michael Conradt (Google, DE)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Michael Conradt

The talk described the experience Google made with clone detection, briefly outlined a few future ideas and what the resulting requirements for a clone detection system are.

3.6 Industrial Clone and Malware Detection





Andrew Walenstein (University of Louisiana at Lafayette, US)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Andrew Walenstein

This presentation looked at commonalities between malware detection and clone detection.

3.7 Where is the “business” case for software clones?

Serge Demeyer (University of Antwerpen, BE)

License     Creative Commons BY-NC-ND 3.0 Unported license
© Serge Demeyer

Joint work of Van Rompaey, Bart; Du Bois, Bart; Demeyer Serge et. al.

Main reference B. Van Rompaey, B. Du Bois, S. Demeyer, J. Pleunis, R. Putman, K. Meijfroidt, J. C. Dueñas, B. García, “SERIOUS: Software Evolution, Refactoring, Improvement of Operational and Usable Systems,” in Proc. of 13th European Conf. on Software Maintenance and Reengineering (CSMR’09), pp. 277–280, 2009.

URL <http://dx.doi.org/10.1109/CSMR.2009.30>

Between 2006 and 2008 our research group was involved in the ITEA project entitled SERIOUS (Software Evolution, Refactoring, Improvement of Operational & Usable Systems) [1]. Code Clones as a symptom of refactoring opportunities were of prime importance during this project as the goal of the project was to deliver a refactoring handbook. As such we attempted to establish a so-called “business case” for code clones; that is, we tried to calculate a potential return on investment of refactorings that would remove clones. During


this lightning talk I shared a few anecdotes on this quest for a business case. *SPOILER ALERT*: Unfortunately, the story ends with an anti-climax. In the end, we abandoned the business case for code clones in favour of project-specific business cases.

References

- 1 Bart Van Rompaey, Bart Du Bois, Serge Demeyer, John Pleunis, Ron Putman, Karel Meijfroidt, Juan C. Duenas, and Boni García. Serious: Software evolution, refactoring, improvement of operational & usable systems. In *13th European Conference on Software Maintenance and Reengineering (CSMR 2009)*. IEEE Press, March 2009.

3.8 A Controlled Experiment on Software Clones

Jan Harder (Universität Bremen, DE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Jan Harder

Joint work of Harder, Jan; Tiarks, Rebecca

Main reference J. Harder, R. Tiarks, “A Controlled Experiment on Software Clones,” in Proc. of the Int’l Conf. on Program Comprehension, 2012.


Most software systems contain sections of duplicated source code—clones—that are believed to make maintenance more difficult. Recent studies tested this assumption by retrospective analyses of software archives. While giving important insights, the analysis of historical data relies only on snapshots and misses the human interaction in between. We conducted a controlled experiment to investigate how clones affect the programmer’s performance in common bug- fixing tasks. The experiment is based on two small open-source games FrozenBubble and Pacman. For each system, we defined one maintenance task that requires fixing a bug. For each of these tasks, we prepared two variations that differ only in the independent variable, which is whether the bug is cloned or not. The participants were drawn from two different populations. In total 21 students of the University of Bremen and 12 participants of the Dagstuhl seminar 12071 participated in the experiment. The dependent variables, we observed, were the time needed to fix the bug and the correctness of the solution. The results do not reach statistical significance. Nevertheless, we observed many incomplete bug-fixes—in all cases only the more apparent bug symptom was corrected. When the bug was cloned up to 54.5% of the students failed to fix both locations. But also many of the experts—up to 33.3%—overlooked cloned bugs even though they participated in the context of a clone seminar and should have expected clones. We also observed some differences in the time needed to solve the tasks. In most cases the tasks variants without a clone were solved quicker. In one case, however, the experts were faster fixing the cloned variant. This peculiarity could be caused by the small sample size. A full report on the experiment has been published to ICPC.

References

- 1 J. Harder, R. Tiarks. *A Controlled Experiment on Software Clones*. Proceedings of the 20th International Conference on Program Comprehension, 2012.

3.9 Issues in detecting license violations

Armijn Hemel (GPL Violations Project, NL)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Armijn Hemel


Joint work of Hemel, Armijn; Vermaas, Rob; Dolstra, Eelco; Kalleberg, Karl Trygve;
Main reference A. Hemel, K. T. Kalleberg, R. Vermaas, E. Dolstra, “Finding software license violations through binary code clone detection,” in Proc. of the 8th Working Conf. on Mining Software Repositories (MSR’11), pp. 63–72, ACM, 2011.

URL <http://dx.doi.org/10.1145/1985441.1985453>

Violations of Open Source licenses such as the GNU General Public License occur very frequently. In this talk the background of violations in the consumer electronics industry was explained, as well as what methods for detection of the presence of Open Source software in unknown opaque binaries, like clone detection, have been successfully applied.

3.10 Good and Evil clones


Angela Lozano (UC Louvain-la-Neuve, BE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Angela Lozano

One of the difficulties when considering clone management as part of the quality assurance process is the lack of support for informed decisions on which clones to refactor. Clones are supposed to affect an application on three aspects: they may increase or reduce the changes required by the application, they may help to introduce or avoid bugs, and they may facilitate or hamper the application’s understandability. There are arguments claiming both positive and negative effects on these aspects; but so far, the evidence gathered is not convincing enough to reach an agreement. This presentation aims at increasing the awareness on the importance of discriminating clones, showing some of the limitations of current research, and stating some challenges on separating good from evil clones. Although current findings indicate that only a minority of clones are harmful on the changes that an application requires, they are incapable of distinguishing a-priori which clones would have negative consequences. Ultimately, to allow practitioners to prioritize clone refactorings, clone research should focus on their long-term consequences instead of quantifying their immediate effect.

3.11 Improving Software Architecture – Role for Software Clones

Ravindra Naik (Tata Consultancy Services – Pune, IN)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Ravindra Naik

The talk presents the problems observed in existing industrial software, primarily business applications, in the context of the role for software clones. For specific problems in migrating towards software product lines, we describe potential solution approaches that can exploit the software clone detection. We describe the problems that were observed with Printer Controller software (engineering application) and Core Banking product (business application). In general our observations are that the enterprise systems are increasingly not able to meet future needs and keep encountering similar function applications in different silos. Some of

the software products, on the other hand, face difficulties in providing new capabilities to all existing customers, and usually customizations (specific to customers) take much longer and are expensive. We note that though exorbitantly expensive, enterprise systems have the option of redesigning and developing from scratch, but the software products do not pragmatically have such an option, lest they are willing to support old customers with versions of old implementation. For software products, migrating to product-line architecture is a potential option [4]. Thus, we observe that software architecture improvement is a common theme across variety of software systems. In the context of software products (especially related to business), we observed that copies are made of the software sources and are customized for every customer. This makes it very difficult for the product team to provide new features to all existing customers, as they have to replicate the new features for every custom implementation. Therefore, among other architecture improvements, migrating to product-line architecture is of prime importance in such cases. Given the situation of one version of the product for each customer and each version having its copy of the source code, the idea is to exploit the capability of Software Clone detection to detect commonality and the variability in the differing assets. The Software Clones in question are a potential variation of the semantic clones or Type-4 clones [2]. Identifying the clones will enable identifying common code, meaning the code that is identical or common in various implementations. Among the variants (which have differing code), identifying the differences, viz. the differing variables, fields, conditional checks, statements, and blocks of code will enable identifying parameters for the variants. Further, the differences need to be detected in functional features or in transactions / processes; there could be constraints under which the differences may (or may not) hold. The critical part of detecting clones is the ability to do so in the presence of multiple functions implemented in a single program or subroutine; also in the presence of already existing but overloaded and inconsistently used parameters [3]. The automation of detection and refactoring, and giving guarantees of the re-factorings are of prime importance for the success of such an approach in the industry.

Acknowledgements

My thanks to various business groups within TCS and my lab head Mr. Arun Bahulkar for the intense discussions and feedback on the software system's problems.

References

- 1 T. Mens and T. Tourwé. 2004. A Survey of Software Refactoring. *IEEE TSE* 30, 2, 126-139.
- 2 S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE TSE* 33, 9, 577-591.
- 3 Hitesh Sajjani, Ravindra Naik, and Cristina Lopes. Application Architecture Discovery – Towards Domain Driven Easily Extensible Code Structure, *WCRE* Oct. 11, 401-405.
- 4 Angela Lozano. An Overview of Techniques for Detecting Software Variability Concepts in Source Code, *ER2011 Workshop – Advances in Conceptual Modelling: Recent Developments and New Directions*, Oct. 11, 141-150.

4 Working Groups


During a brainstorming discussion involving all participants, various issues were gathered that should be discussed in separate and parallel smaller working groups. These issues were grouped into cohesive clusters. A working group was formed for each cluster. The identified clusters were as follows (see Section 1 for a short description of their goals and results):

- Detection/Use cases
- Presentation
- Interoperability
- Refactoring
- Clone management (process)
- Provenance and clones in artifacts that are not source code

The following sections summarize the results of these working groups. In two cases – namely, the working groups on *Clone management (process)* and *Provenance and clones in artifacts that are not source code* – we will just report the notes that were added to the seminar’s wiki in the course of the seminar. All other reports are based on the wiki’s entries, too, but were written down and further elaborated after the seminar.

4.1 Working group on clone detection

Thierry Lavoie (Ecole Polytechnique Montreal, CA)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Thierry Lavoie

4.1.1 Abstract

Although many efficient clone detectors are readily available, it is still unclear how to use them to solve practical industrial problems. In order to address and focus future research on this issue, many use cases for clone detection were identified and characterised with their defining clone detection features. An overview of known limitations and issues of actual clone detectors is also provided along with some research questions oriented towards the improvement of clone detection techniques.

4.1.2 Introduction

Many clone detection tools are readily available today, but few provide insights on how to interpret and use the detected clones. Even if the state-of-the-art tools have solved the problem of detecting Type-1 and Type-2 clones, many issues need to be addressed both regarding higher types detection and result applicability. In order to propose new focuses for clone detection research, the group identified known issues with current detectors as well as many relevant research questions. As a result, the group suggests to do new clone detection research with a focus on use-case oriented results instead of a broad-scope clone detection.

This report is divided in two sections: the first presents known issues with clone detectors with relevant research questions, and the second presents many use cases and their cloning related features.

4.1.3 Known issues and limitations

Many aspects of type 3 and 4 clone detection are still eluding clone researchers. Those types are required for many use cases. Therefore, it is worth looking at some current problems.

Regarding Type-3 clones, the following questions are still open:

- How can we effectively find Type-3 clones?
- How can we scale Type-3 clone detection effectively?
- Is grouping of Type-3 clones into disjoint sets really appropriate?

With respect to the first question, a use-case oriented approach might suggest a way to better quantify and qualify actual Type-3 clones as it points towards a better clone definition (one that is useful for the use case) instead of the now vaguely defined “gapped” clones. Scalability might as well be solved on a case by case basis. Grouping of clones into disjoint clone classes is a natural choice for type 1 and 2 clones. However, disjoint classes suggest an equivalence relation, which may be ill-formed for Type-3 clones. Specifically, transitivity does not seem to trivially, or at all, hold and symmetry is questionable. Therefore, clone classes should be rethought for Type-3 clones. Regarding Type-4 clones, their current definition as semantic clones is above all too vague. Without a clear conception of what should be a Type-4, or semantic, clone, it is hard to state how it should be detected. Nevertheless, there is a common agreement that only few tools can barely deal with semantic clones and semantic clones are relevant because they do occur in practice.

4.1.4 Limitations of clone detectors

Clone detection tools accuracy still needs improvement. Since the group suggests to head towards use-case-based clone detection, it is natural to ask how can human feedback be used to increase the accuracy of results. Distinguishing relevant and irrelevant clones might become an easier problem if tools are configured for one specific task and results are manually inspected. However, it is still unclear how human feedback might be used meaningfully.

With the evolution of malware and the increase of license infringement problems, obfuscated code becomes an issue for which clone detection tools were not conceived to deal with. Binary clone detection is also relevant for those specific problems and for which tools are not well suited. Investigation of these problems might give potent solution to practical clone detection applications.

4.1.5 Category-oriented use cases

In order to define the challenges modern clone detection tools must overcome to solve practical problems, the group identified several clone detection use cases. For each of them, required features of clone detection tools were identified. In Table 1, the relevant clone types for each use cases are identified. Clone types right to other clone types subsume them. For example, *Type-3 large gap* subsumes *Type-3 small gap*, Type-2 and Type-1 and is itself subsumed by Type-4.

In Table 2, other relevant features are identified. A cross in a cell indicates the feature is required. Each feature is defined as follow:

- Precision: A low rate of false positives is required
- Recall: A low rate of false negatives is required
- Online: A fast, realtime tool is required
- Granularity: The desired size of the clones. *Fine* means small fragments are desired whereas *coarse* indicates the need to identify only large fragments. *Fine&Coarse* indicates clone size is not relevant.
- Incremental: The tool needs to handle multiple fragment additions and deletions
- Blacklisting: The tool needs to handle a corpus of code that must not be considered clone
- Binary: The tool needs to find clones in source code as well as in executable binaries
- Counter-obfuscate: The tools need to deal with obfuscated sources or binaries

■ **Table 1** Highest relevant clone types for identified use cases

Use Case	Type-1	Type-2	Type-3 small gap	Type-3 large gap	Type-4
Abstraction identification			X		
Version analysis tasks				X	
Code reduction		X			
License infringement		X			X
Plagiarism			X		
Code Leakage				X	
Provenance			X		
Productivity measurement		X			
Quality assessment			X		
Regulations complianc		X			
Malware					X
Program comprehension					X
Awareness				X	

■ **Table 2** Required features of clone detection tools for identified use cases

Use Case	Precision	Recall	Online	Granularity	Incremental	Blacklisting	Binary	Counter-obfuscate
Abstraction identification	X			Fine&Coarse				
Version analysis tasks				Coarse	X			
Code reduction	X			Fine&Coarse				
License infringement	X			Coarse		X	X	X
Plagiarism		X		Coarse		X		X
Code Leakage		X		Coarse				
Provenance	X			Coarse	X			
Productivity measurement	X			Fine&Coarse	X			
Quality assessment	X			Coarse	X	X		
Regulations compliance		X		Fine&Coarse				
Malware	X			Coarse			X	X
Program comprehension	X		X	Coarse				
Awareness	X		X	Fine				

4.1.6 Business cases and Cost/Benefits analysis

Using the identified use cases for business purposes is not straightforward. In many cases, a cost/benefits analysis must be first performed to decide whether or not clone analysis is worth investigation. The followings are research questions for which an answer would provide a better intuition on how to use clone detectors in industrial applications:

- What is the business case for clone search and reduction?
- How to measure whether code-clone removal takes less effort than clone management?
- How much does the cost to remove a clone increase with age?
- How can we measure the benefits of clone detection?
- Can we empirically characterize the costs / benefits of different clone refactorings?
- How to get statistics about costs / risks associated with existing clones or avoided clones?
- How to determine the relative importance of clones in a project?

For some use cases, some ways of determining the industrial benefits were identified:

- Abstraction identification: speed-up development by refactoring and having better knowledge of the system
- Version analysis task: speed-up in version merging using clone detection instead of other techniques
- License infringement and provenance: avoid legal problems and reduce costs of legal department
- Productivity measurement: increase in management decision quality
- Quality assessment: reduction in maintenance cost, reduction in audit cost, increased quality of internal assessment, increase quality of third-party quality assessment of suppliers (software escrow)
- Program comprehension: decrease time in comprehension

4.1.7 Conclusion

The group identified many relevant clone-detection use cases along with their required clone-detection features. The group also supports reorientation towards application-oriented clone detection instead of self-purposed-oriented clone detection. In many cases, state-of-the-art clone detection tools do not behave well for these features. These observations point to new research opportunities to enhance clone detection technologies.

4.1.8 Participants

The following people took part in the group discussion and contributed the main ideas of this report:

- Andrew Walenstein, University of Louisiana at Lafayette
- Jochen Quante, Robert Bosch GmbH
- Elmar Jürgens, TU München
- Serge Demeyer, University of Antwerp
- Yingnong Dang, Microsoft Research Beijing
- Stephan Diehl, University Trier
- Jim Cordy, Queens University
- Rainer Koschke, University of Bremen
- Michel Chilowicz, Université Paris-Est

- Thierry Lavoie, École Polytechnique de Montréal
- Werner Teppe, Amadeus Germany GmbH
- Martin Robillard, McGill University
- Rebecca Tiarks, Bremen University
- Michael Conradt, Google
- Minh Zibran, University of Saskatchewan
- Jindae Kim, HK UST

4.2 Working group on clone presentation

SSandro Schulze, Niko Schwarz

License  Creative Commons BY-NC-ND 3.0 Unported license

© Sandro Schulze, Niko Schwarz

Main reference S. Schulze, N. Schwarz, “How to Make the Hidden Visible – Code Clone Presentation Revisited,” Technical Report FIN-05-2012, University of Magdeburg, Germany, 2012.

URL http://www.cs.uni-magdeburg.de/inf_media/downloads/forschung/technical_reports_und_preprints/2012/04_2012.pdf

4.2.1 Abstract

Nowadays, a slew of clone detection approaches exists, producing a lot of clone data. These data have to be analyzed manually or automatically. It is not trivial to derive conclusions or even actions from the analyzed data. In particular, we argue that it is often unclear how to present the clone information to the user. As a result, we present our idea of task-oriented clone presentation based on use cases. Hence, we propose five use cases that have to be addressed and suggest clone presentation techniques that we consider to be appropriate.

4.2.2 Introduction

Intensive research has been performed on clone detection and evaluation—presentation is often left as an implementation detail to implementors. While there is a plethora of visualizations, current visualization for code clones is limited [1, 2]; they can not serve different issues (e.g., online clone reporting, quality assessment, refactoring). They are rather directed to a certain task for which they are more or less appropriate.

We want to stimulate the topic by discussing what is needed to present and visualize code clones to an end user. This inherently raises the question: What do we want to discover from the code clones, once they have been found by a detection tool? If we can clearly answer this question, we have the ability to find appropriate methods to present this information.

So far, different tasks, related to detected code clones, require different tools to reveal information that is needed for a particular task. In this report, we propose a mapping that shows which visualizations and presentation concepts can serve which purpose. While our suggestions are far from being complete, the objective is to guide tool builders and give an overview over what is there and how it could be exploited. Our vision is a tool or IDE that seamlessly integrates these approaches to provide different views on clones and thus fit the needs of different stakeholders.

4.2.3 Use Cases for Clone Presentation

Once code clones have been detected and analyzed, they must be accessible for further treatment. This step, called clone presentation, is not an obvious task. First of all, there

might be different stakeholders such as software quality managers or software developers that need different views (including different levels of granularity) on the clones. Second, these stakeholders want to perform different actions. In the following, we propose five use cases that encompass the different views and treatments of clones.

4.2.3.1 Quality assessment (QA)

This use case mainly appears on the management level. For instance, the stakeholder wants to have an rough estimation on how the existing clones affect the overall system quality. Furthermore, the detection of hot spots, i.e., parts of a system that contain a larger amount of clones, to define countermeasures or just reason about the clones are part of this use case.

4.2.3.2 Awareness (AW)

This use case describes the fact that it is important for certain stakeholders, especially developers, to be aware of existing clones and how they are related. In particular, during implementation a developer has to know when he changes a cloned fragment. Additionally, the information where the corresponding clones are located is useful to making consistent changes in an efficient way.

4.2.3.3 Bug prediction (BP)

If a bug has been found in a clone of a code snippet, then all other clones might be incorrect as well. Further, if a code snippet is copied from a source to a destination, a certain similarity between source and destination is implied. This could be exploited to predict bug occurrences.

4.2.3.4 Quality improvement (QI)

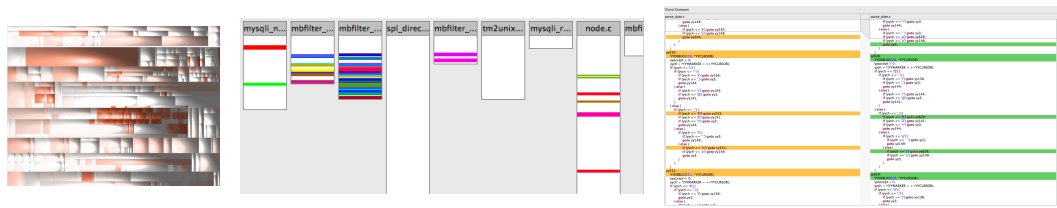
This use case encompasses persistency and removal of clones. For the first, we envision an enrichment of clone information by the clone producer (i.e., the developer) such as whether a clone is harmful or should not be removed. The latter case encompasses refactoring techniques and all information that is needed to apply them to detected clones.

4.2.3.5 Compliance (CO)

This use case encompasses two issues: First, a stakeholder may be interested in whether code in the systems exists that has been copied from external sources (e.g., third party libraries). Hence, he must ensure that the license is not violated. Second, there could be subsystems that contain code, which is not allowed to be used outside this subsystem such as sensitive code or pre-defined architectural or responsibility boundaries. As a result, it is useful to have a presentation that indicates whether such *internal* compliances are violated.

4.2.4 Putting the Pieces Together

Not all visualizations lend themselves equally to all tasks. In the last section we described the use cases we identified and that have to be addressed by an appropriate clone presentation. However, due to the fact that different approaches are possible for clone representation and visualization, for each use case we focus only on a subset of techniques and methods that we commonly agreed on during intensive discussions. For a more comprehensive overview on possible visualization techniques, we refer to existing surveys on this topic [4, 5]. In Table 3,



■ **Figure 1** Examples for (a) a tree map, (b) a seesoft view, and (c) a compare view from the clone detection and report part of ConQAT [3]

we show a compatibility matrix that relates the use cases to clone presentation methods we propose to address particular use cases.

■ **Table 3** Matrix showing which clone presentation feature can be used for which use case.

	QA	AW	BP	QI	CO
SeeSoft View [6]	X	X	?		X
TreeMap [7]	X	X			X
Source code view		X	X	X	
Compare view			X	X	
Links		X			X
Dashboard	X	?			X
Filtering/querying/zooming	X	X			
User-generated meta-data				X	
Revision history	X		X		

Particularly, we argue that clone visualization such as SeeSoft views or TreeMaps are helpful to provide a big picture of the clones in the system and thus support the use cases QA, AW, and CO. To this end, a SeeSoft view (cf. Figure 1, middle) represents each file as rectangle and each clone as a bar within this rectangle, indicating its size and position. Additionally, code clones that belong to the same clone set have the same color. As a result, the stakeholder receives an overview of clones and how they are scattered throughout the system. Similarly, a TreeMap (cf. Figure 1, left) represents each file as a rectangle with information on size and position, relatively to the whole system. Furthermore, the color indicates whether such a file contains many clones or not, which enables an easy detection of so-called *hot spots*. However, we also propose to make such visualizations more interactive by adding filtering, querying, and zooming capabilities. Particularly for large code bases, this allows to focus on subsets of the overall code base, which are of interest.

In contrast to the previously mentioned visualizations, a developer requires methods for clone presentation that are seamlessly integrated in his development process. We propose that the source code view (as provided by common IDEs) and a compare view (cf. Figure 1, right), providing a face-to-face comparison of two code clones, are appropriate to fulfill these demands and thus to support the use cases BP and QI. For the source code view, we even suggest to integrate more sophisticated approaches such as linking between corresponding clones. As a result, the developer could receive information on corresponding clones in case that he changes a cloned code fragment. Furthermore, he could be provided with means to change the corresponding clones consistently. Beyond that, the compare view can provide even more fine-grained information such as highlighting the differences of two code clones.

Finally, the aforementioned approaches can be complemented by further presentation

techniques. For instance, the revision history can be exploited to provide evolutionary information about the clones while user-generated meta-data (e.g., by *tagging the clones*) can provide useful insights about the developer’s view on certain clones.

4.2.5 Summary

We have summarized the most common use cases of clone detectors and mapped them to visualizations that can display them to the user. While we do not claim completeness, we want to stimulate discussion on our categorization of use cases and the respective clone presentation/visualization approaches.

4.2.6 Participants

Participants of this working group were as follows:


- Hamit Abdul Basit
- Saman Bazrafshan
- Daniel M. German
- Nils Göde
- Martin P. Robillard
- Niko Schwarz
- Sandro Schulze
- Gunther Vogel

References

- 1 R. Tairas, J. Gray, and I. Baxter, “Visualization of Clone Detection Results,” in *Eclipse technology eXchange*. ACM, 2006, pp. 50–54.
- 2 J. Cordy, “Exploring Large-Scale System Similarity Using Incremental Clone Detection and Live Scatterplots,” in *ICPC*, 2011, pp. 151–160.
- 3 E. Juergens, F. Deissenboeck, and B. Hummel, “CloneDetective – A Workbench for Clone Detection Research,” in *ICSE*, 2009, pp. 603–606.
- 4 S. Diehl, *Software Visualization*. Springer, 2007.
- 5 C. K. Roy and J. Cordy, “A Survey on Software Clone Detection Research,” Queen’s University at Kingston, Tech. Rep. 2007-541, 2007.
- 6 S. Eick, J. Steffen, and J. Sumner, E.E., “Seesoft – A Tool for Visualizing Line Oriented Software Statistics,” *IEEE TSE*, vol. 18, no. 11, pp. 957–968, 1992.
- 7 B. Johnson, “TreeViz: Treemap Visualization of Hierarchically Structured Information,” in *CHI*, 1992, pp. 369–370.

4.3 Working group on interoperability

Cory Kapser, Jan Harder, Ira Baxter, Douglas Martin

License  Creative Commons BY-NC-ND 3.0 Unported license
© Cory Kapser, Jan Harder, Ira Baxter, Douglas Martin

4.3.1 Abstract

As the field of code clone research grows, the continuing problem of interoperability between code clone detection and analysis tools grows with it. As a working group, we sought to solve this problem by generating a comprehensive model for code clone detection results that can

be used in a wide range of use cases. As a result, we generated a conceptual model of code clone detection results that can be used to specify exchange languages, web services, output formats, and more. Following the workshop we created an online wiki, where we hope to generate discussion and solidify a shared understanding of the core concepts of the problem domain with the code clone detection and analysis community as a whole.

4.3.2 Introduction

Research on code clones in software – segments of similar code within or between software systems – is continually growing. As the number of code clone detection and analysis tools increases, the number of output formats and parsers for those output formats grows. Yet as a scientific community there is an increasing need to share results, not only for the purposes of replication of experiments, but to enable us to efficiently build on top of each others' results. This leads us to the issue of interoperability of our tools and results.

As a group we realized that before we can solve the problem of interoperability, there needs to be a shared understanding of the core concepts of the problem domain. The working group participants employed object-oriented analysis of the problem domain as a method of identifying important domain concepts. Starting with brain storming use cases for clone detection results, we identified a diverse set of use cases where code clone detection results are used. These became our basis for evaluating the completeness of our concept analysis. Using these use cases as a reference, requirements and core concepts were identified and encoded as classes and associations. The results of this work will continue to evolve, and the most up to date information can be found at <http://www.softwareclones.org/ucm>.

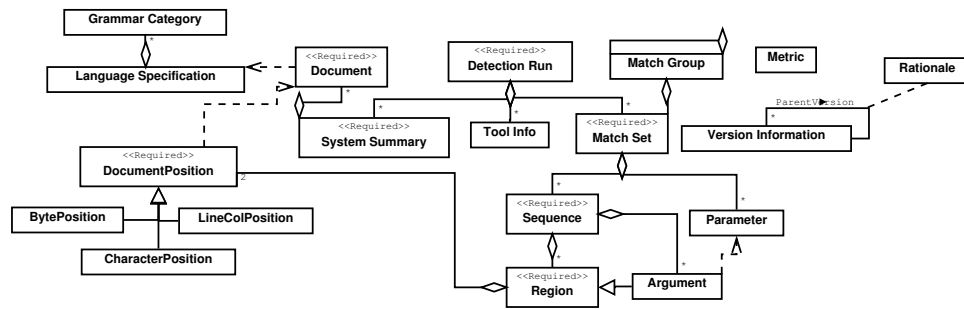
Generic data formats have been proposed [2] but these models may not be complete enough for all available use cases of detection results, nor do they model the core of clone detection results in a truly generic way. Further, these models encode details and constraints specific to their implementations, particularly to suit the models' purpose. For example, RCF specifically models *clone pairs* and *clone classes* separately though it can be argued that the latter is the more general form. Also, the concept of higher level clustering of code clones is not explicitly modelled in RCF. As the model presented here is a description of core concepts and their relationships to one another, potential contributions of this model include:

- a shared decomposition of the problem domain,
- reduced learning overhead for new tool developers and stakeholders as most core concepts have been identified,
- a standardized language for discussing code clone detection results,
- a well defined model to be used to generate a concrete exchange language, and
- a central model for which existing data formats can be documented relative to.

Further, the original use cases can be mapped to the specific concepts in the model, providing a standardized way to communicate minimum requirements for specific usage scenarios.

4.3.3 Use Cases

Three possible domains within which code clone detection would be used can easily be identified: clone detection for computer programming languages, clone detection for non-formal languages (e.g., natural language documents), and clone detection for graph based documents. Working within the first domain during the session, eleven high-level use cases



■ **Figure 2** Unified Clone Model

for clone detection results were identified. These use cases were used to stimulate directed object-oriented analysis going forward as well as verification of the resulting conceptual model afterward. We fully expect this list will be expanded as the larger community is engaged in the discussion. The following use cases were identified:

1. **UC 1: Detect and report.** Detect similarity and simply report it to the user.
2. **UC 2: Detect, report, and track evolution.** Detect similarity and track the evolution of these results across software versions.
3. **UC 3: Detect, report, refactor.** Detect similarity and report them for the purpose of refactoring.
4. **UC 4: Metric analysis.** Generate a metric based analysis of a software system including code clone based metrics (perhaps to study the relationship of code clones, their metrics, and other source code and software development related metrics).
5. **UC 5: Data fusion.** Smarter integration/augmentation of multiple data sources to create more value than the code clone results alone (e.g., improve ROI for code clone analysis by identifying high value/low cost refactoring cases).
6. **UC 6: Scientific replication of a study.** Provide sufficient information about the clones, the detection process, and the source code to replicate the results.
7. **UC 7: Benchmarking.** Benchmark/compare code clone detection tools.
8. **UC 8: Hybrid approaches.** Enable tools to pass data to each other in hybrid clone detection tool chains.
9. **UC 9: Reduce rework.** Provide useful, extra information that could be computed by another tool but presents a significant amount of work. Ensures the results stand completely on their own.
10. **UC 10: Detect for reporting, enable easy navigation and search.** Used to move from clone to clone, snippet to snippet, and enable search within code clones.
11. **UC 11: Plagiarism detection where no source code is available.** May only be able to share minimal results, need to still be able to compare them.

4.3.4 Model

The diagram shown in Figure 2 depicts a model without the concept attributes. In this section, the important features of the model are described and the concept attributes are listed. The conceptual model shown in Figure 2 is encoded as a UML class diagram. Each box represents an important concept identified during the analysis. Those boxes with the stereotype *Required* are deemed to be required for the most basic use case *Detect and Report*. In this case, clone detection results for a single version of the software are simply reported to

the user without any interpretation. As the model reported here reflects the core concepts of the problem domain, it should be noted that this is a model of important concepts, not a data format or OO design.

At its core, the model describes a *detection run* as an instance of a clone detection tool or tools (*tool info*) being run over a document corpus (*system summary*). A detection run also includes a *run start time*. *Tool info* includes the attribute names, version, tool arguments/options, tool chain description, and possibly a boolean to indicate whether or not the code clones detected are returned as classes or pairwise.

The results of the clone detector (code clone pairs or classes¹) are stored as *match sets*. A match set is modelled as a set of *sequences* and *parameters*. Each sequence is an ordered list of *regions*, contiguous segments within a document, that represent the matching fragments detected by the clone detector. The parameters of the match set indicate the points of variability in the mapping (for Type-2 and Type-3 clones this is analogous to gaps in the clone). Each sequence maps an *argument* to each parameter in the match set. In the case of a token based clone detector, a sequence is the whole code fragment that was found to be similar. This sequence is decomposed into the identical fragments (regions) and the differing fragments (parameters/arguments). Match sets, sequences, and regions can have *metrics* and *version information* associated with them as well. This version information could be used to track clones across versions of the document corpus, and versions of sequences across versions of the document corpus. Regions include start and end *Document positions*, text (as found in the system), a checksum, and grammar category (for classification of contained artifacts). Document position representation remains a point of contention within the working group. Three alternatives are suggested in the figure. Byte position, while possibly being the most portable is also the least convenient as that information may be lost in the pre-processing stages. This is similarly true for character position. Line and column position may be the most convenient for many detection tools, but also may require a character interpretation mapping so as to ensure unambiguous interpretation of special characters (such as line feed).

Modelling code clones in this way allows for a sequence of a code clone to span multiple files, and for matching regions to have an arbitrary order (e.g., (1,2,3):(3,1,2)). These scenarios can occur, for example, when clone detectors return results from pre-processed source code where macros have been in-lined [3]. They can also occur when clone detectors that are resistant to line reordering are used, such as PDG based clone detectors [5]. For clone detectors that return clone pairs, a match set would consist of two sequences. For those clone detectors that return clone classes, a match set would contain two or more sequences.

Code clone detectors may apply a clustering of code clones as part of their result set, such as Regional Group of Clones (RGC) [4] or clone classes generated based on the clone pair relationship. In these cases, this can be represented as a *match group*. For higher level clusterings, match groups can also be aggregates of other match groups.

The *system summary* represents a version of the document corpus being analyzed. It consists of *documents*, *metrics*, and *version information*. Documents are the units being analyzed for code clones. Their attributes include the URI, version information, metrics, checksum, original text, preprocessed text, the processed model (such as a serialized AST if one was used) and a *language specification* which provides enough details for the consumer of the detection results to interpret the document position as well as understand how the document was processed by the clone detector. A language specification includes a name

¹ A clone class is a set of two or more code fragments that are considered to match. This is often considered to be an equivalence relation.

(such as C, Java), dialect (such as VS 2008), reference information describing where the language specification can be found, character interpretation to describe how characters map from character or binary offset in the file to line and column positions, and possibly a grammar specification.

Although not shown in the diagram, most concepts can be associated with any number of *metrics*. This is used when additional information, such as similarity, line count, or complexity, are stored. The model is highly extensible in this respect as metrics can be arbitrarily defined using name, value, and type attributes. Also, most nodes can contain *version information*, enabling the tracking of match sets, sequences, and regions independent of versions of the corpus (system summary). This enables, for example, the modelling of genealogies of code clones including linking the origins of specific regions of code. This version information can contain a *rationale* which is used to describe how or why one entity was traced to a prior version.

4.3.5 Conclusion and Future work

The model presented here presents only the beginning of this work. If we wish to create a general model that can be adopted by the community, there needs to be general acceptance by the community. Therefore, a wiki (<http://www.softwareclones.org/ucm/>) has been created to discuss and share the full details of the model. There we will also share the details of reference implementations of database schemas, exchange languages, and web services.


As part of a verification of the basic completeness of the model, a mapping to RCF was performed. The results of this process exposed very few modifications to RCF and no modifications to the model described in this paper. While this is encouraging, we must go further to validate the completeness of the core concepts. In this vain we will perform this mapping to other existing models, including the output of CloneDR [1] but also models not developed by the authors. This will not only ensure we have captured the essence of the problem domain, but also provide examples of how to document existing clone detection result formats relative to this model.

References

- 1 I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. *Clone detection using abstract syntax trees*, in Proceedings of the International Conference on Software Maintenance (ICSM-98), pp. 368, IEEE Computer Society, 1998.
- 2 J. Harder and N. Göde, *Efficiently handling clone data: Rcf and cyclone*, in Proceedings of the 5th International Workshop on Software Clones, pp. 81–82, ACM, 2011.
- 3 I. J. Davis and M. W. Godfrey, *From whence it came: Detecting source code clones by analyzing assembler*, in Proceedings of the 2010 17th Working Conference on Reverse Engineering (WCRE '10), pp. 242–246, IEEE Computer Society, 2010.
- 4 C. J. Kapsner and M Godfrey. *Improved Tool Support for the Investigation of Duplication in Software*, in Proceedings of the 2005 International Conference on Software Maintenance (ICSM-05), pp. 305–314, IEEE Computer Society, 2005.
- 5 J. Krinke. *Identifying similar code with program dependence graphs*, in Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE '01), pp. 301–309, IEEE Computer Society, 2001.

4.4 Working group on refactoring

Ira Baxter (Semantic Designs, US)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Ira Baxter

4.4.1 Abstract

Much of the research on (software) clones has been focused on methods of detection, understanding, and determining evolutionary properties of clones and their actual impact on software maintenance. However, an implicit assumption behind clone detection is that most or at least some clones should be “refactored” out of existence, unifying the instances into some kind of effective abstraction. Yet there are extremely few tools or methods for actually forming clone abstractions from clones in code or other formal documents, and/or clone refactoring²: replacing cloned artifacts with abstract invocations, and inserting the clone abstraction at some other accessible point in the code.

The purpose of this working group was to consider the mechanics and utility of forming clone abstractions and achieving clone refactoring.

4.4.2 Format

Like the other working groups, we started with index cards containing all-attendees brainstormed one-line topics, that had been filtered into the category that seemed to be “refactoring” (thus the group title). We further classified these into finer sets and tackled each in turn to understand where there might be a synthesis of ideas.

We grouped the topics into several major subtopics, which we discuss below:

- Normal refactoring “within” the formal document
- Using abstractions from outside the language system of the formal document
- Managerial aspects of refactoring: cost, benefits, risks
- Refactoring to product lines

There were several subtopics we did not get to explore and surely deserve the attention of a working group at some future date:

- Clone refactoring applied to non-formal texts (documentation, requirements, parallel refactoring of multiple sets of documents)
- Clone refactoring for graph-structured artifacts, including various types of models
- The relationship of domain analysis/engineering, e.g., clone abstraction to mine reusable components. The observation is that detected clones are often recognized by the programmers that work on a system as to intent, and therefore an abstracted clone has both a concept and a realization, as well as an obvious use in the type of software from which it is extracted.

4.4.3 Clone refactoring: State-of-the-art

At present, most clone detection systems are not associated with any ability to refactor clones for removal (exceptions: CloneDR³ [5], Erlang [12] and Haskell [7], a functional language).

² We suggest using the specific term “clone refactoring” to distinguish this specific activity in the more generic set of activities called “refactoring”.

³ CloneDR was able early in its life to refactor C code with macros, and COBOL code with COPYLIBS. That capability is presently not used.

We considered what technology is available to support clone removal.

The maturity of certain conventional refactorings suggests clone refactoring is practical: “pull up method”, and “form procedure”, both having elements needed for clone refactoring, are generally reliable in participants experience. This suggests that clone refactoring itself should be reliably implementable. There was some contradictory concern that behaviour preserving refactoring is not solved reliably (especially for sequences of refactorings).

One key problem is to obtain robust tools to manipulate the program representation (ASTs, symbol tables, flow analysis), especially for the essentially endless variety of languages for which clone detection seems to be applicable. Most clone analysis tools, e.g., those that match text strings, token sequences, or class files, do not actually have access to such a representation; they have to be integrated with some other tool ecosystem (Eclipse, Clang, ...) to support such refactoring. Clone detectors such a CloneDR [5] built with a general purpose program transformation engine such as DMS [4] should be easier to morph into a clone refactoring tool; such tools have been used to carry out complex code restructurings on languages such as C++ [1].

4.4.4 Clone Refactoring Issues

It is not easy being green. All kinds of issues must be addressed to refactor clones.

- Under the somewhat suspect notion that one wants to remove all clones, can one use an entirely automated approach to remove them? We think this is unlikely: the resulting code is not likely to be understandable, as it is unclear how such a tool would choose a sensible clone abstraction name. Perhaps there are clues in names of variables or in comments or in the nature of the detected clone.
- What are the criteria for suggesting a reasonable refactoring candidate? Can it be tiny (perhaps, if there are hundreds of instances)? Can it have a large number of parameters? Must it be abstractable using a language capability? Should it have some indication of high code churn within the individual clones?
- The right abstraction depends on a lot of information: the parameters (e.g., relationship and count of the different locations), and this seems to be different for each clone. One might desire to refactor (or not!) subsets of a large clone set differently to take advantage of identical parameter bindings. The implications are that removal is likely to be an interactive task.
- Is there only one way to remove a clone? Likely not: several abstractions may be available in the programming language (conditionals, macros, subroutines, ...) for procedural clones, and others available for declaration clones (macros, classes, ...). For any given clone instance, syntax/semantic category, language or client, is there a single preferred way? If so, a clone refactoring tool could have a default method for removal; the user decides if she wants a clone refactored that way. If not, can we provide a catalog of prioritized abstraction possibilities for each detected clone type to help a user choose quickly?
- Is clone removal done only by abstraction capabilities available in the language in which the clone was found, or can one step outside the language and use external metaprogramming techniques to abstract the clone? How does a clone removal tool know about the abstraction methods offered by all the languages it can handle? How does it know about the external metaprogramming facilities?

4.4.5 Structural Clones

Before we discuss abstractions outside the language, we will first examine structural clones [2], as they will play a big role in later discussion on refactoring in this report. The key notion here is that smaller clones may in fact be part of a larger pattern; getters tend to be associated with setters, and so one should expect cloned getters to have corresponding cloned setters. Structural clone detectors use potentially multiple conventional code clone detectors to find clones that form elements of structure clones, and then hunt for repeated patterns of such elements in larger container structures (methods, classes, files, entire directories).

[2] offers one structural clone detector based on item-set frequency analysis. Are there other means to detect structural clones? Can we measure or compare the quality? An open question is “what kind of patterns can be formed from elements to make up a structural clone”? Is the pattern a possible parameter of a structural clone? (e.g., elements $A \dots B$ are found in one container; elements $B \dots A$ are found in another, forming a structural clone with a boolean parameter: “forward order of elements”).

Abstracting structural clones is conceptually very difficult; they may cross many types of language, abstraction and file boundaries. A lot of domain/expert knowledge may be required to abstract a structural clone.

4.4.6 Abstractions outside the language

When refactoring clones, one set of abstraction mechanisms are those offered by the formal language in which the clones were found, e.g., macro and function calls for C, templates and classes for Java, etc. We discussed the idea that a clone refactoring tool might offer refactorings using abstraction mechanism that are not available to that language. A variety of useful generic abstraction/reification mechanisms are available:

- **General macro processors:** One might use (Unix) M4 to supply text-based macro capability to languages that do not have it. (An uglier but similar idea already occurs commonly in large Fortran codes that use the C preprocessor to provide configuration conditionals as well as macros.)
- **Frame generators:** These provide what amounts to tree-structured text macros that generate entire files from explicit configuration parameters driving sophisticated conditionals (Frames [3] or XVCL [10]). GenVoca has been suggested as a generalization of frame technology [6]. We remark that XVCL, being able to produce arbitrary text artifacts, has been used successfully to abstract structural clones.
- **File level selection:** These tools choose between alternatives for files based on configuration conditionals. In essence, these are implicit preprocessor conditionals at the file level. (A product line management tool, Gears [11] offers this as one of its features).
- **Code generators and DSLs:** These generate result code given an input specification in some chosen specification language. A key problem is choosing an appropriate specification language (raising the domain analysis/engineering question), and determining a specification that can be realized to match the clone instances. A special case of this are program transformation systems (e.g., DMS [4]), which can convert abstract operations to target code by applying refinement transformations, and can abstract optimizations as guided sets of transformations. A special case of program transformation systems is intentional programming [13] which might be considered to be a kind of feature language.
- **Feature languages:** These are DSLs that used named, possibly parametrized “features” (perhaps contingent upon others) as abstractions to be realized [8].

■ **Table 4** Abstraction methods for clones in executable and declarative code

method	executable	declarative
common lambda inference (e.g., extract method) ... extended to lift lambda to common area	x	–
“template method” (abstract algorithm, interface, API) may need to merge several clones	x	–
text-based techniques	x	x
– macros, includes	x	x
– compile-time configuration (preprocessor) conditionals, file level	x	x
– frame generator (e.g., XVCL)	x	x
code generator (transformation, intention [Simony95])	x	x
runtime configuration conditionals (if/then/else, switch/case)	x	–
aspects	x	-
typedefs (e.g., structs => union)	–	x
abstract data types	–	x
object formation (e.g., legacy => OO)	x	x
normalized representation (e.g., date format)	–	x
feature formation intentional/conceptual (Type-5)?	x	?
purpose annotation for conceptual clones	x	x

4.4.7 Summary of possible code abstraction mechanisms

Most clone detection work seems to focus on cloned executable code. With CloneDR, it was observed that there are many clones in (data) declarations as well as code. We considered what kinds of abstractions might be available for code clones, and for declaration clones, to produce Table 4.

Table 4 should be extended if possible; a survey might be a useful research topic. It might be useful to collect a rather complete set of abstraction mechanisms used in software engineering (category theory, anyone?), and consider the mechanisms required to support these in clone refactoring. Which of these should be offered as a standard set to support clone refactoring opportunities? Is this standard set independent of the language in which the clones are found? How does one handle the availability of a customer-unique abstraction mechanism?

A brief discussion ensued about “clone types”. Clone Type-1 corresponds to exact-match code (perhaps modulo blanks), Type-2 to clones with single-token parameters, Type-3 to clones with larger parameters, although it is unclear how complex a parameter might be or even if it must be contiguous in the code text. Type-4 has been used to to classify clones that match semantically; since the discussion is often about clones detected with automation, presumably these are clones recognized using some semantic comparison mechanism (e.g., isomorphic dataflows), for which there are practical and theoretical limits on capability.

In considering how one might abstract code in general, it struck us that in general one might have conceptual clones, that is, blocks of code whose purpose has similar abstractable intention, but for which no mechanical detector is available (e.g., bitonic sort and radix sort routines), but have reasonable abstractions (e.g., intentions [13]). Such conceptual clones must be discovered in part by use of a human oracle. There does not seem to be a (conceptual!) problem with conceptual parameters for such clones. Should such clones have a designated type (e.g., Type-5?)

It is also a little unclear where structural clones fit in this spectrum; it is clear they have common parts including conceptual clones; as an odd extreme, one might have a structural clone that was composed entirely of conceptual subclones. Structural clones may have parameters (Type 2 or 3) induced by their component subclones, but are there “parameters” induced by the regions around the structural clones? We do not seem to have any kind of useful characterization of the nature of parameters that a clone may have; how are we to abstract without understanding what parameters might be? In the same vein, what are the parameters of variation of the structure itself?

4.4.8 Managerial Aspects of Clone Refactoring

Assuming that one can refactor clones, there is the issue of should one refactor? We briefly discussed the following:

- Decision to declone based on cost-benefit: We do not have clear economic models of the benefit of removing clones. It is becoming clearer that removing all clones may not pay off. How do we measure costs and impacts? How do we decide which ones to remove? How do we decide which abstraction mechanism to apply? How do we manage the rest, if at all?
- Cost of having changed the code: When determining cost benefit, completion of refactoring a clone is not the end of the cost; as a practical matter, changed code must be re-compiled, re-tested, re-deployed. Management often has a “do not touch anything that works” attitude partly to control this. Are some clones easier to remove? Can some be removed without doing retest? What about performance impacts of functionally reliable decloning?
- How to minimize manual refactoring work: One can automate the removal of all clones, but this is generally not a good idea. Given the possibility that each clone pair/set might be remedied differently (including not remedied), fully automated removal is likely to produce clones remedied inappropriately⁴. So there likely needs to be some interactive selection of how individual clones are removed. Given that 10% of a code based might be cloned, a million line system might have 20,000 5-line clones in 10,000 clone pairs. Interactive review of such a huge amount of data is daunting at best. Perhaps one can design defaults or heuristics so that the reviewing engineer has a simple interaction once per clone (“one click to orbit”) remediation to accept the default.

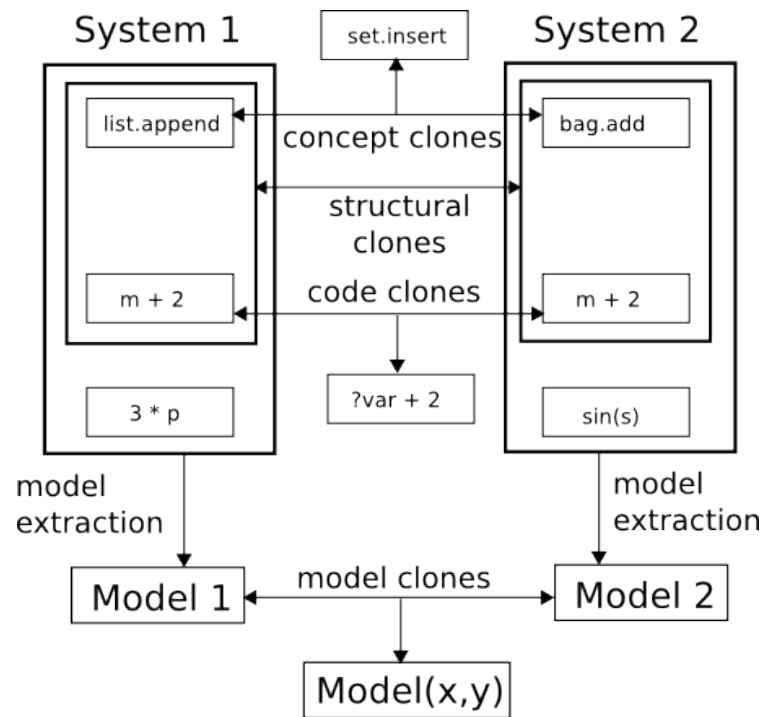
4.4.9 Refactoring To Product Lines

The code base for large software systems sometimes gets forked (multiple times!), and the resulting large-scale clones (e.g., full-code bases) then begin largely independent lives at great maintenance costs to the owning organization⁵. It is often clear after the fact that a product line should have been constructed, but the sheer scale of the systems and lack of deep understanding of process or usable tools prevent the organized construction of such a product line by somehow merging the forks. Can we refactor such enormous clones⁶ into a product line? What process and technologies are needed?

⁴ An early version of CloneDR removed all clones in a C system by converting them into macros, producing legal, compilable, runnable code, that the programmers instantly rejected because their individual code was not remedied as they would have desired.

⁵ Semantic Designs has a client with 30 copies of a large-scale core-banking system, customized to different countries legal systems and cultural product needs.

⁶ Clearly one can apply clone detection management techniques within a product instance, but for product lines, we are interested only in clones across the product line instances.



■ **Figure 3** Forming a product line from system instances

This is especially difficult in that the component languages which comprise the code base for the product line as a practical matter almost surely cannot express an abstraction that covers the entire code base. To abstract system clones, one must step outside the component languages.

What is needed are:

- means to describe the resulting product line (e.g., a specification-type of abstraction)
- methods to detect the similarities across the cloned systems
- methods to abstract the similarities into elements controlled by the specification style
- methods to manage the differences in the systems

Abstractions for classic software clones can usually be expressed in the language in which the clone was found. For product lines, often composed of multiple different computer languages as well as informal documentation, no obvious unified abstraction mechanism exists. In essence, one has to move to some kind of generator scheme in which the abstraction is expressed as some kind of specification, and a corresponding generator can produce the instance system code needed for a particular specification instance. One might choose some kind of abstract interconnection model (e.g., UML, Component/Connector architectures, [9], Module Interconnection Languages [14]) or a (set of cooperating) domain specific languages; if the latter, where does domain knowledge come into the process? A “generic” class of DSLs such as feature description languages [8] appear to be reasonable candidates for encoding the abstraction, to the extent that the features can be coupled to some kind of generative process that can produce instance systems from a selected set of feature specifications. Oversimplifying, Gears [11] suggests abstracting product lines with features that select entire files that comprise the product, and offers a commercial product for managing product lines using this technique.

Product line abstraction from code. Regardless of the final abstraction, somehow the similarities and the differences of the system clones must be found and eventually integrated into the product line. In the seminar, we generated Figure 3. We see the variety of ways in which various types of clone detection and abstraction techniques might be applied to two systems instances. At the lowest level, standard code clone detection techniques can be used to discover parametrized code abstractions that the product line generator will likely need to instantiate. Because code clone detectors cannot identify code blocks with similar intent but unsimilar code, one is likely to need to allow conceptual clones to be interactively identified; perhaps domain ontologies would be helpful. Structural clone detectors are needed to determine where sets of clones across the system instances indicate a higher level application structure; we draw attention to the fact that such structure-clone detectors must operate over the results of any of the lower-level clone detectors and even recursively over smaller detected structural clones. Any remaining code fragments not allocated to structural clones or abstracted away become (possibly enormous) parametric values of the product line instances themselves.

Product line abstraction from models. An alternative is to somehow model the systems, forming corresponding models (e.g., UML, Petri Net, ...), and apply clone detection over the models to generate an abstract model. Since feature models are models, it might be useful to build a feature model of individual systems, and do clone detection over those features. Is it possible or useful to do both abstraction from models and code in a synergistic way? We remark that structural clones might contain subclones derived from code and subclones derived from models.

Either approach leaves open the question of how the detected (structural or model) clones are abstracted back to features, specifications, or DSL elements.

It would be interesting research to manually construct a product line using clone detection processes on system instances, to provide some insight and details about how such a product line forming process might work.

4.4.10 Summary

It is remarkable how much ground one can cover in small, lively subgroup in just a few hours. The discussion was not anywhere near as linear as this report implies. This reporter has tried to do the discussion justice and augmented it somewhat, particularly adding references that seemed relevant. Any errors or misconceptions in this summary are the fault of the reporter, not the group. Thanks go to Jochen for capturing excellent notes, and to the seminar organizers for enabling us to have this discussion.

We close with a summary in Table 5 describing how to apply clone refactoring to various types of artifacts. This table should be extended to handle non-code artifacts.

4.4.11 Participants

Participants of this working group were as follows:

- Ira Baxter, Moderator, Reporter
- Sandro Schulze
- Ravindra Naik
- Hamid Abdul Basit
- Angela Lozano
- Yingnong Dang
- Jochen Quante, Scribe

■ **Table 5** Clone refactoring in various types of artifacts


<i>artifacts</i>	<i>refactoring technology</i>
code	see Table 4
data	see Table 4
feature model (UML, ontology, ...)	unknown
conceptual clones	human provided abstraction
product line instances – same language/technologies	combination of above refactorings
product line instances – different technologies	unknown. Likely conceptual clones, domain analysis/engineering

References

- 1 R. Akers, I Baxter, and M. Mehlich. Re-Engineering C++ Components Via Automatic Program Transformation. 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation
- 2 Hamid Abdul Basit, Stanislaw Jarzabek. Towards Structural Clones – Analysis and Semi-Automated Detection of Design-Level Similarities in Software. VDM 2010: I-XII, 1-153
- 3 P.G. Bassett. The Case for Frame-Based Software Engineering, IEEE Software, July 2007, pp. 90–99
- 4 I.D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program Transformation for Practical Scalable Software Evolution. International Conference on Software Engineering, pp. 625–634, 2004.
- 5 I.D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. International Conference on Software Maintenance, IEEE Press, 1998
- 6 James Blair and Don Batory. A Comparison of Generative Approaches: XVCL and GenVoca, Department of Computer Sciences. www.cs.utexas.edu/ftp/predator/xvcl-compare.pdf
- 7 C. Brown and S. Thompson. Clone Detection and Elimination for Haskell, PEPM’10, January 18–19, 2010.
- 8 Krzysztof Czarnecki. Understanding Variability Abstraction and Realization. International Conference on Software Reuse ICSR 2011: 1-3
- 9 D. Garlan and R. Allen. Formalizing Architectural Connection, Proceedings ICSE 16, IEEE 1994.
- 10 S. Jarzabek and S.Li. Eliminating Redundancies with a ‘Composition and Adaptation’ Meta-Programming Technique, Proc. European Software Eng. Conf./ACM/SIGSOFT Symp. Foundations of Software Engineering, (ESEC/FSE 03), ACM Press, 2003, pp. 237–246;
- 11 Charles W. Krueger. The BigLever Software Gears, Systems and Software Product Line Lifecycle Framework. SPLC Workshop 2010: 297
- 12 H. Li and S. Thompson. Clone Detection and Removal for Erlang/OTP within a Refactoring Environment. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM’09).
- 13 Charles Simonyi. The Death of Computer Languages, the Birth of Intentional Programming (technical report) 1995
- 14 R. Prieto-Diaz and J. Neighbors. Module Interconnection Languages, Journal of Systems and Software 6, 1986.

4.5 Working group on clone management (process)

Jens Krinke (UCL, GB)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Jens Krinke

What follows are the notes kept on the seminar wiki of this working group.

4.5.1 Clones and Process

Where may clone analysis play a role in the development and maintenance process?

4.5.1.1 During Development

- As a part of QA in Continuous Integration, seen as a testing / integration / metric activity.
- It can be integrated with commits.
 - Generate commit messages automatically (“Copied X from Y.”) which can be edited and/or augmented by the user.
 - Prevent commits that would create clones (or too large clones).
 - Automatically create annotations (traceability links) between the copied code and the copy.
- It can be used before and after commits.
- During editing, providing immediate feedback (“similar code to the one you are editing exists at A, B, and C”).
- Tracking the copy/paste operations may generate useful information but may also create too much information.

4.5.1.2 During Requirements Engineering

Observation: clones in requirements may lead to semantic clones in the code (different developers implementing the same feature because of cloned requirements).

Clone detection during requirements engineering may prevent clones in later stages.

4.5.1.3 During Testing

Lots of clones exist in (unit) test suites. If code is cloned, is the test code cloned with it? Must the test code be cloned first in TDD?

4.5.1.4 After Deployment

Is my code leaking to other products? (Provenance)

4.5.2 Code Search and Cloning

Programmers use code search to find code that already does what they want to do, which is then copied. This may increase copied code – however, is this bad? Not necessarily, because cloning code is cheaper than developing a feature from scratch. Moreover, detecting of such clones is easier / possible in comparison to detect semantic clones due to reimplementations from scratch. Maybe another case of good cloning?

Other code search: search for similar / cloned code: Where does similar code exist?

Notifications of clones: what are the implications of them at edit time and commit time?

4.5.3 Recommender Systems

Recommender systems for cloned code: “You edit a clone, maybe you want to edit X and Y, too?” Implications may be similar to co-change based recommender systems: “You edit X and you have always edited X with Y and Z in the past.” However, there is no correlation between co-changes and clones. Moreover, half of the time clones evolve independently (Lague did such a study already in 1997).

- What can be recommended? (e.g., API mining)
- In which way? As wizard? As clippy?
- Depends on use / business case.

4.5.4 More questions

- What are interesting clones?
- Can clones be ranked? How?
- What to do with bugs and clones, bugs in clones?
- Are large Type-1 clones the most interesting ones?

All of the above questions may have a different answer for different development tasks. “Often, developers/management think that they are in control of the cloning and don’t have to act on it. What if they are wrong?”

- How to define “wrong”?
- Is a general question, not specific for cloning

When to

- track
 - At commit.
 - At copy/paste actions? Necessary for immediate feedback.
 - What is the granularity of the tracking?
- detect in real-time
 - Good as long as it does not get in the way.
- refactor
 - When the user needs it.
 - Depends on the use /business case.

Can copy/paste information be used for clone detection? “Maybe there is a clone...”

We need an ethnographic study.

Clone analysis may play an important role in a software product line development process.

Clones are created because of code ownership as it is hard to change other developer’s code (clone-to-own).

Larger issues: Forks are created of “social” reasons (see forks of major open-source software). We are missing an “integration culture”.

4.5.5 Participants


Participants of this working group were as follows:

- Michael Godfrey, University of Waterloo, CA
- Jindae Kim, The Hong Kong University of Science & Technology, HK
- Jens Krinke, University College London, GB

- Angela Lozano, UC Louvain-la-Neuve, BE
- Ravindra Naik, Tata Consultancy Services – Pune, IN
- Werner Teppe, Amadeus Germany GmbH, DE
- Minhaz Zibran, University of Saskatchewan, CA

4.6 Working group on provenance and clones in artifacts that are not source code

Serge Demeyer (University of Antwerp, BE)

License  Creative Commons BY-NC-ND 3.0 Unported license
© Serge Demeyer

What follows are the notes kept on the seminar wiki of this working group.

4.6.1 Clone Analysis in Binaries

4.6.1.1 Use cases

- License infringement. Example: The app store problem – is open source used in the app store?
- Malware Detection. Example: Microsoft releases a patch – detect the differences; where are you vulnerable?
- Abuse case example: detect the difference; where can I exploit?

4.6.1.2 All boils down to two different cases

1. You know that the corpus contains the subject (in that case you can try all techniques until you find whatever you are looking for).
2. You do not know whether the corpus contains the subject (in which case you can just argue adequacy; legal term = “due diligence” = I did my best to according to the state of the art in the field).

4.6.1.3 What is the information you can exploit?

- Call graphs
- Libraries used
- Signatures of methods/classes
- Strings (and constants)
- Runtime analysis (observing behaviour)
- File name
- Call usage (call graphs)
- No code (data files used, services used)
- Metrics of the binary
- Op codes
- Frequency based analysis (spectography)

Open question: what is important (in a pool of information)?

4.6.1.4 Research agenda problems/questions

- How to create traceability links to maintain the history of an artifact?
 - How to insert this into organizational process / awareness / ...?
 - How to certify the origin; what should be in the “manifest” that accompanies a software artefact?
- Diffing
 - in the case where you have two binaries which you know are descendants from one another;
 - challenging because some differences are caused by irrelevant changes (change in compiler version, options)
- Origin
 - Which point in time in the VCS was used to generate this binary? Example: You have the DEBIAN VCS and a binary; which version of DEBIAN does it come from?
 - Given a binary and the source code; is the source code the actual source code used to create the binary?
 - People copying JAR files and dropping version info; what version did I use?
 - Which version of telnet was used in malware?
- How do we evaluate that our methods are good? (⇒ Benchmarks)
 - What are the common tasks?
 - Malware detection problems?
 - What in the case of obfuscation?
- Building a Corpus in the case of provenance
- Language dependent issues
- Adversarial? (incl. obfuscation)

4.6.2 Provenance

4.6.2.1 Can we automatically add some meta-data, signatures to binaries?

- Similar to EXIF for JPG files;
- currently based on a web of trust; when downloading open-source software the signatures and the binaries are kept together; there is no separate authority that authorizes signatures

4.6.2.2 Who did what post-mortem?

Manifest of a software artefact; like in ship cargo (what’s inside the container) or like a software bill of materials

- There is an industry motivated group who is standardizing this software manifest concept
 - Software Package Data Exchange
- Clarity of the supply chain; which are the organizations who produced a given component?

4.6.2.3 IEEE malware working group proposal

Working group has a taggant effort⁷. Packers compress executables. The IEEE working group would like the packer vendors to create packers that sign the packed executables with a digital signature that can be traced back to the packer vendor and packer vendor’s customer,

⁷ <http://standards.ieee.org/develop/indcomm/icsg/malware.html>

and permit the signature to be revoked if the packer is stolen or the packer vendor plays both sides (sells to both white and black hats).

4.6.2.4 Where would such a “centralized authority” come from?

www.OHLOH.net: a web-site which keeps statistics of all open source software

4.6.2.5 Research Agenda

There are various non-technical issues which severely challenge the use cases applications.

Tool support is really missing. Triangulating with partial information is a potential way to go. Clone detection may contribute there.

4.6.3 Clones in models

4.6.3.1 Use Case; e.g., Simulink

Jim Cordy is looking for clones in Simulink models. GM wants to answer a question like “Is a given piece of a model –where we suspect there is a safety issue– used elsewhere in the car?” This boils down to the question we have seen elsewhere. If you discovered an issue in one model, can you look for it in others?

4.6.3.2 Observation: Culture of clones in other engineering disciplines

- In other engineering it is an accepted practice of scaling up by replicating proven designs.
- In computer science, we do not do that; we create our own abstractions and then repeat the abstractions.
- Clones are a symptom for the potential of creating such abstractions.
- However, the language must allow for “program-like” abstraction facilities.
- Most engineering disciplines lack the languages for expressing said abstractions.
- Within engineering modelling there is a new wind; with expressing higher order abstractions.
- Automotive is a good example: engineers would like to control (hence model) the emerging properties of systems.
- Example: if I push on this emergency button, will the system stop in time? The current best practice is to run many simulations and worst case scenarios.

4.6.3.3 Questions

How does duplication in models trigger abstraction?

- Having a replication of a good idea;
- pattern matching on languages/models used in other engineering disciplines is a prerequisite.

4.6.3.4 Clones in pictures

Models usually have a graphical representation; couldn’t we just use clone detection of images? Example: Getty wants to protect its copyright and spies the web for copies of its images.

- Google is now able to detect “clones” of PNG files.
- What would happen if you use that kind of facility on UML diagrams?

- Here as well: having copies of UML diagrams implies that it is a good design since people want to copy it; it is not about license infringement or so.

4.6.3.5 Research Agenda

- Look how other disciplines how they deal with duplication / replication: First thing to watch out for: Do there exist “program like” abstraction facilities?
- Reach out to other communities: Show nice examples of things we have achieved with clone research.

4.6.4 Clones in bug reports

Clones in stack traces / debug back traces (i.e., the stack traces associated with a bug).

- Clone detection there might help to identify most frequently (re-)occurring bugs.
- Canonical (the company behind the Ubuntu linux version) would be very eager for knowing which bugs occur frequently in the field.
- Integrators (= organizations combining components coming from various sources) might very interested as well. It would help them to identify which subcontractor caused the bug. Same applies for (distributed programming) teams; assigning a bug report to the right team is critical to reduce bug resolution time.

4.6.4.1 Research agenda

Clone detection on stack traces looks promising.

4.6.5 Overall research agenda

- When approaching “other” documents to search for clones there are two starting points:
 1. look for catalogs of patterns/abstractions/domain concepts; knowing what to search for is important as a first step
 2. verify whether there exist “program like” abstraction facilities in the languages used. This helps in identifying potential for removal of clones, or whether it stays at searching for similar occurrences.
- Observation: Clone detection, diffing, provenance and even search are intimately linked; a common thread throughout all what we discussed

4.6.5.1 Side note: Bizarre application for Type-4 clones

In n-version programs, verify whether the versions are indeed Type-4 clones (= semantically equivalent) but not Type-3 or lower (= syntactic equivalence).

4.6.6 Participants

Participants of this working group were as follows:

- Mike Godfrey
- Andrew Walenstein
- Serge Demeyer (scribe)
- Niko Schwarz
- Jens Krinke
- Armijn Hemel
- Daniel M. German
- Douglas Martin

Participants

- Hamid Abdul Basit
LUMS – Lahore, PK
- Ira D. Baxter
Semantic Designs – Austin, US
- Saman Bazrafshan
Universität Bremen, DE
- Michel Chilowicz
Université Paris-Est –
Marne-la-Vallée, FR
- Michael Conradt
Google – München, DE
- James R. Cordy
Queen's Univ. – Kingston, CA
- Yingnong Dang
Microsoft Research – Beijing, CN
- Serge Demeyer
University of Antwerpen, BE
- Stephan Diehl
Universität Trier, DE
- Daniel M. German
University of Victoria, CA
- Michael W. Godfrey
University of Waterloo, CA
- Nils Göde
CQSE GmbH – Garching, DE
- Jan Harder
Universität Bremen, DE
- Armijn Hemel
GPL Violations Project, NL
- Elmar Jürgens
CQSE GmbH – Garching, DE
- Cory J. Kasper
Calgary, Alberta, CA
- Jindae Kim
The Hong Kong University of
Science & Technology, HK
- Rainer Koschke
Universität Bremen, DE
- Jens Krinke
University College London, GB
- Thierry Lavoie
Ecole Polytechnique –
Montreal, CA
- Angela Lozano
UC Louvain-la-Neuve, BE
- Douglas Martin
Queen's Univ. – Kingston, CA
- Ravindra Naik
Tata Consultancy Services –
Pune, IN
- Jochen Quante
Robert Bosch GmbH –
Stuttgart, DE
- Martin P. Robillard
McGill Univ. – Montreal, CA
- Sandro Schulze
Universität Magdeburg, DE
- Niko Schwarz
Universität Bern, CH
- Werner Teppe
Amadeus Germany GmbH, DE
- Rebecca Tiarks
Universität Bremen, DE
- Gunther Vogel
Robert Bosch GmbH –
Stuttgart, DE
- Andrew Walenstein
University of Louisiana at
Lafayette, US
- Minhaz Zibran
University of Saskatchewan, CA

