IEEE 13th International Working Conference
on

Source Code Analysis and Manipulation

22-23 September 2013
Eindhoven, The Netherlands

SCAM

IEEE computer society

GRAMMATECH

SEMANTIC DESIGNS

IEEE Catalog Number: CFP13SRC-ART
ISBN: 978-1-4673-5739-5

Editors
Bram Adams, École Polytechnique de Montréal, Canada
Juergen Rilling, Concordia University, Canada
Foutse Khomh, École Polytechnique de Montréal, Canada

Sponsors
IEEE Computer Society
GrammaTech
Semantic Designs

Message from the Chairs

Dag allemaal!

On behalf of the SCAM 2013 Conference and Program Committee, we would like to welcome you to the capital of Dutch industrial design, i.e., Eindhoven, the Netherlands, for the 13th IEEE International Working Conference on Source Code Analysis and Manipulation, co-located with the 29th IEEE International Conference on Software Maintenance (ICSM 2013).

Some of you might ask whether, after 12 highly successful editions chock-full of innovative ideas, case studies and tools, one still needs a 13th edition: isn't SCAM a solved problem? Of course, the chairs of the conference and (more importantly) the SCAM community will answer NO to that question. It is true that more and more attention in the wider software engineering community is being directed towards other aspects of systems development and evolution, such as specification, requirements engineering, design, evolution and maintenance. However, a fundamental artefact in all these domains is still the source code, which contains the precise, and sometimes only, definitive description of the behaviour of a system. Hence, even in a world of mobile applications, web services, model-driven engineering and cloud computing, new and innovative SCAM techniques will still be required, even essential.

So, for the 13th time in a row, the SCAM working conference aims to bring together researchers and practitioners working on theory, techniques and applications for the analysis and manipulation of the source code of computer systems, in order to help researchers tackle the software engineering challenges and problems of today and tomorrow. SCAM focuses on the techniques and tools themselves - what they can achieve, how they can be improved, refined and combined. It does this through a highly interactive format: paper presentations are kept short and focused, with ample time reserved for general discussion of challenges and controversial opinions raised during each session. An important side-effect of these discussions is that it helps the community to stay focused and become a tighter group.

This year we received 29 research papers and 15 tool papers, from which we have selected 14 excellent research papers and 10 amazing tool papers for presentation and inclusion in the proceedings. The papers cover a broad range of topics including: static analysis, dynamic analysis, source code transformation, metrics, software mining, databases/ontologies and source code visualization. We want to thank the authors of all submissions for sharing their research with the SCAM community.

Every paper was fully reviewed by three or more program committee members for relevance, soundness and originality, and discussed openly by the entire program committee before a unanimous, final decision was made. Hence, a big thank you to the program committee and external reviewers for their timely and constructive reviews, and special thanks to those who actively participated in the discussions of the final selections.

This year's SCAM also features an expanded tools track, where each accepted tool paper will be presented during one of the regular paper sessions and demonstrated live in a hands-on tool demonstration session. Tool papers were evaluated by a separate committee, using the same acceptance criteria as the main track, complemented by tool-specific criteria like making the tool (and example data) available for download and explaining the tool's architecture and inner workings. These criteria follow directly from SCAM's vision of providing a platform for sharing new advances and results with fellow researchers and practitioners, enabling the rapid progress of the field.

We gratefully acknowledge the several sponsors of SCAM 2013 who have made the event possible: the IEEE Computer Society Technical Council on Software Engineering, Semantic Designs Inc. and GrammaTech Inc.

Last but not least, also a special thanks to the organizers of ICSM 2013 for their help in co-location and to the outstanding SCAM 2013 Organizing Committee: social media wizards Veronika Bauer and Felienne Hermans for taking care of advertising the conference on the various social media channels that matter; Davy Landman for the flashy website; the long-suffering Dave Binkley for financial arrangements; the ever-diligent Marcin Zalewski, for managing and producing the proceedings, and Sacha Claessens and Martijn Labbers for the amazing local arrangements in Eindhoven.

We hope that you will find the conference stimulating and rewarding, and that you have a very enjoyable visit to Eindhoven and its surrounding regions (including Belgium ;-)).

Welkom in Eindhoven!

Jurgen Vinju	**Bram Adams and Juergen Rilling**	**Foutse Khomh**
SCAM 2013 General Chair	**SCAM 2013 Program Chairs**	**SCAM 2013 Tool Demo Chair**

Organizing Committee

General Chair:

Jurgen Vinju, Centrum Wiskunde & Informatica and Universiteit van Amsterdam, The Netherlands, and INRIA Lille, France

Main track PC co-chairs:

Juergen Rilling, Concordia University, Canada

Bram Adams, École Polytechnique de Montréal, Canada

Tool track PC chair:

Foutse Khomh, École Polytechnique de Montréal, Canada

Publication chair:

Marcin Zalewski, Indiana University, USA

Local organization chairs:

Sacha Claessens, Eindhoven University of Technology, The Netherlands

Martijn Klabbers, Eindhoven University of Technology, The Netherlands

Publicity:

Veronika Bauer (chair), TU München, Germany

Felienne Hermans (social), TU Delft, The Netherlands

Davy Landman (web), Centrum Wiskunde & Informatica, The Netherlands

Program Committee

Research Paper Track

Paul Anderson, GrammaTech, Inc.
Giuliano Antoniol, Ecole Polytechnique de Montréal
Ira Baxter, Semantic Designs
Árpád Beszédes, Department of Software Engineering, University of Szeged
Dave Binkley, Loyola University Maryland
Mariano Ceccato, Fondazione Bruno Kessler
Luigi Cerulo, University of Sannio, Italy
Pascal Cuoq, CEA LIST
Sebastian Danicic, Goldsmiths College, University of London
Thomas Dean, Queens University
Massimiliano Di Penta, RCOST, University of Sannio, Italy
Nicolas Gold, University College London
Mark Harman, University college london
Emily Hill, Montclair State University
Abram Hindle, University of Alberta
Rob Johnson, Stony Brook University
Akos Kiss, Department of Software Engineering, University of Szeged
Rainer Koschke, University of Bremen
Jens Krinke, University College London
Ralf Lämmel, Universität Koblenz-Landau
Cristina Marinescu, Politehnica University of Timisoara
Philip Mayer, Ludwig-Maximilians-Universität München
Ettore Merlo, Ecole Polytechnique de Montreal
Marius Minea, Politehnica University of Timisoara
Rocco Oliveto, STAT Department, University of Molise
Denys Poshyvanyk, College of William and Mary
Ju Qian, Nanjing University of Aeronautics and Astronautics
Filippo Ricca, DISI, Università di Genova, Italy
Chanchal K. Roy, University of Saskatchewan
David Shepherd, ABB, Inc
Jeremy Singer, University of Glasgow
Alexandru Telea, University of Groningen
Vadim Zaytsev, Centrum Wiskunde & Informatica

Tool Paper Track

Steering Committee

Pascale Cuoq (2010–2013), CEA-Recherche Technologique, France

Thomas Dean (2004–2007, Re-elected 2009-2012, Re-elected 2012-2015), Department of Electrical and Computer Engineering, Queen's University, Canada

Leon Moonen, (2003-2006, Re-elected 2006-2009, Re-elected 2011-2014), Software Engineering department, Simula Research Laboratory, Norway

Sibylle Schupp (Chair, 2009-2012, Re-elected 2012-2015), Institute for Software Systems, Hamburg University of Technology, Germany

Jurgen Vinju (2010-2013), Centrum Wiskunde & Informatica, The Netherlands

Andrew Walenstein (2011-2014), University of Louisiana at Lafayette, USA

TABLE OF CONTENTS

Session 4 — Source Code Metrics

Session 5 — Code Transformation

Session 6 — Databases and Ontologies

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

Empirical Investigation of SEA-Based Dependence Cluster Properties

Árpád Beszédes*, Lajos Schrettner*, Béla Csaba†, Tamás Gergely*, Judit Jász* and Tibor Gyimóthy*

*Department of Software Engineering, University of Szeged, Hungary
E-mail: {beszedes, schrettner, gertom, jasy, gyimothy}@inf.u-szeged.hu

†Department of Set Theory and Mathematical Logic, University of Szeged, Hungary
E-mail: bcsaba@math.u-szeged.hu

***Abstract*—Dependence clusters are (maximal) groups of source code entities that each depend on the other according to some dependence relation. Such clusters are generally seen as detrimental to many software engineering activities, but their formation and overall structure are not well understood yet. In a set of subject programs from moderate to large sizes, we observed frequent occurrence of dependence clusters using Static Execute After (SEA) dependences (SEA is a conservative yet efficiently computable dependence relation on program procedures). We identified potential linchpins inside the clusters; these are procedures that can primarily be made responsible for keeping the cluster together. Furthermore, we found that as the size of the system increases, it is more likely that multiple procedures are jointly responsible as sets of linchpins. We also give a heuristic method based on structural metrics for locating possible linchpins as their exact identification is unfeasible in practice, and presently there are no better ways than the brute-force method. We defined novel metrics and comparison methods to be able to demonstrate clusters of different sizes in programs.**

***Index Terms*—Source code dependence analysis, dependence clusters, linchpins and linchpin sets, Static Execute After.**

I. Introduction

Dependences in computer programs are natural and inevitable. We can talk about dependences among any kind of artifacts such as requirements, design elements, program code or test cases, but dependences within the source code capture the physical structure as implemented best. A dependence between two program elements (*e. g.* statements or procedures) basically means that the execution of one element can influence that of the other, hence the software engineer should be aware of this connection in virtually any software engineering task involving the two elements. One of the fundamental tasks of program analysis is to deal with source code entities and the dependences between them [1].

Dependences cannot be avoided, but they do not always reflect the original complexity of the problem. Sometimes unnecessary complexity is injected into the implementation, which may cause significant problems. A relatively new research area explores *dependence clusters* in program code, which are defined as maximal sets of program elements that each depend on the other [2]. The current view is that large dependence clusters are detrimental to the software development process; in particular, they hinder many different activities including maintenance, testing and comprehension [3], [4], [5], [6], [7]. The primary problem is that in any dependence-related examination, encountering any member of a cluster forces us to consider all other cluster members. If large clusters covering much of the program code exist in a system, then it is very likely that one cluster member is encountered and consequently a large portion of the program code should be enumerated eventually.

978-1-4673-5739-5/13/$31.00 © 2013 IEEE

The root causes of this phenomenon are not well understood yet; it seems to be an inherent property of program code dependence relationships. As apparently dependence clusters cannot be easily avoided in the majority of cases, research should be focused on understanding the causes for the formation of clusters, and the possibilities for their removal or reduction. Previous work revealed that in many cases a highly focused part of the software can be deemed responsible for the formation of dependence clusters [4], [5]. Namely, program elements called *linchpins* are seen as central in terms of dependence relations, and are often holding together the whole program. If the linchpin is ignored when following dependences, clusters will vanish, or at least decrease considerably.

Of course, it is useful if one is aware of such linchpins, let alone be able to remove them by refactoring the program. However, currently even the first step (identifying linchpins) is largely an unexplored area. We still do not understand fully what makes a particular program point a linchpin, how they can be identified, or whether there is always a single element to be made responsible in the first place. The possibilities for linchpin removal by program refactoring are even harder to assess. Sometimes, dependence clusters are avoidable because they actually introduce unnecessary complexity to the implementation; this is what Binkley *et al.* call "dependence pollution" [2]. In such cases the program can be refactored using reasonable effort, but this is not always the case.

In this work, we present the results of our empirical investigation of dependence clusters in a range of programs of moderate (up to 200 kLOC) and large sizes. In the latter category we investigated two industrial size open source

software systems, the GCC compiler [8] and the WebKit web browser engine [9], each consisting of over a million LOC.

We are dealing with procedure-level program dependences computed using the *Static Execute After (SEA)* approach (on C/C++ functions and methods). The SEA relation between two procedures is a conservative type of dependence that takes into account the possible control-flow paths and call-structures in the program elements [10]. SEA-based dependences can be used, among others, in software change impact analysis. The main advantage of SEA is that it achieves acceptable accuracy, yet can efficiently be computed even for large systems of millions of LOC. We computed SEA-based dependences of all procedures in our subject programs and investigated the resulting dependence clusters in terms of their frequency of occurrence and by identifying potential linchpins in them. We summarize our findings as follows:

- We introduce the term *clusterization* to indicate the extent programs exhibit dependence clustering, and define novel metrics that indicate this property quantitatively.
- We computed SEA-based dependence clusters for realistic size programs. Among the moderate-size ones there were many clusters, but only one of the big programs included significant clusters, which is an interesting result.
- We were able to identify linchpin elements in most of the clusters using a naïve approach that enumerates all possibilities. In many cases however, especially with the big programs, it is not to be expected that only one program element (procedure) is responsible for the formation of clusters.
- We give a heuristic method based on local procedure metric for linchpin approximation. We found that the number of outgoing invocations from a procedure was quite a good estimator for linchpins.

The rest of the paper is organized as follows. Sections II and III provide relevant background information about the motivation, related work and our experimental environment. Cluster identification is discussed in Section IV, while the topics related to linchpin determination are given in Section V. Section VI discusses threats to validity, and finally we conclude in Section VII.

II. Background, motivation and goals

The phenomenon of *dependence clusters* was first described by Binkley and Harman in 2005 [2] based on program slices and Program Dependence Graphs [11], [12]. Initially, they were defined as maximal sets of statements that all have the same backward slice, which also means that the elements of a dependence cluster each depend on the other. The notion of dependence clusters can be generalized to other kinds of dependence types and different program elements at various granularity. Furthermore, it seems that dependence clusters are independent of the programming language and the type of the system [3], [13], [14].

The current view is that large dependence clusters hinder many different software engineering activities, including impact analysis, maintenance, program comprehension and software testing [3], [6], [7]. It has been suggested that large dependence clusters leading to "dependence pollution" should be refactored [7], [2], but for such opportunities the identification of the dependence cluster causes is essential. Specifically, the identification and possible removal of *linchpins*, the directly responsible program elements, is an active research area. Virtually, the only existing approach to identify linchpins is based on a brute-force method that tries all possibilities. Binkley and Harman predict in their work [4] that it will be possible to pinpoint certain program elements that cannot be identified as linchpins, hence the search could be optimized this way. We employ heuristic methods to identify linchpins, and we are not aware of any previous work that used a similar approach. Global variables can play special role in the formation of dependence clusters, which has been investigated by Binkley *et al.* [5].

In a previous work [15], we investigated the concept of dependence clusters on procedure-level program dependences computed using the *Static Execute After (SEA)* approach. The SEA relation between two procedures is a conservative type of dependence that takes into account the possible control-flow paths and call-structures in procedures [10]. This approach is more efficient at the expense of being a bit less accurate, and is defined as follows. For program elements (procedures, in our case) f and g, we say that $(f, g) \in$ SEA if and only if it is possible that any part of g is executed after any part of f in any one of the executions of the program. Similarly to the definition of slice-based dependence clusters, we regard two procedures to be in the same SEA-based cluster if their dependence sets coincide. This kind of cluster definition is usual as the maximal mutual dependence-based cluster definition is prohibitively expensive to compute. This definition has the additional good property that it gives a partitioning of the procedures into clusters.

Note, that there is no obvious relationship between SEA-based and slice-based dependence clusters. From our preliminary investigations we found that in many cases similar cluster structures will be formed, but we plan to investigate this more systematically in the future, and see if our findings can be applied to other notions of dependence clusters.

We computed SEA-based dependence clusters in the WebKit system – one of the subjects in the present work as well – and found that it exhibits a certain level of clusterization. In the same work [15], we then used the identified clusters to verify the connection to the performance of change impact analysis in a practical situation, and to enhance our test case prioritization method based on code coverage analysis.

Monotone Size Graphs (MSG) and the related "area under the MSG" metric [2] are often used to characterize dependence clusters in programs. An MSG of a program (see Figure 1 for examples) is a graphical representation of all dependence sets belonging to the procedures of the program by drawing the sizes of the sets in monotonically increasing order along the x axis from left to right. Then the area metric mentioned above is the total sum of all dependence set sizes. In the case of SEA-based dependences the total number of dependence sets equals

the number of procedures in a program, and this number is also the maximal dependence set size. In Figure 1, we can see MSGs of such dependences in which – despite its rectangular shape – the same number of procedures is represented on both axes. The most straightforward way of interpreting this graph is to observe dependence clusters as plateaus, whose width corresponds to the cluster size and the height is the dependence set size. Note, that it may happen that the same dependence set size incidentally corresponds to different sets which will not be noticeable in this graphical representation.

It has not yet been investigated thoroughly whether the MSG and its associated area metric are good enough as descriptors of the level of clusterization. We further elaborate on this concept with SEA-based clusters and verify the ways of characterizing dependence clusters using this and other kinds of metrics. A related investigation was performed by Islam *et al.*, who defined alternative descriptions of the clusterization in form of various graphical representations [16]. These approaches, however, resort to visual investigation only.

As noted above, it is believed that a single linchpin can be associated to a program with dependence clusters in many cases. In this work we investigate the effect of joint removal of linchpin candidates in cases where the removal of a single element does not produce the desired result.

An additional problem is linchpin identification itself. The naïve linchpin identification algorithm – a brute-force method trying all possible solutions one by one – is not scalable. Hence, previous research that employed fine grained analysis could deal with programs of up to 20 kLOC only [4]. In a similar fashion, our SEA-based analysis makes it possible to investigate programs with sizes of a magnitude larger thanks to the higher level granularity and a simpler, albeit less precise, analysis method. However, this method is still not usable for bigger programs as is the case with our two large systems.

We articulate the following **Research Questions**:

- RQ1 How typical are SEA-based dependence clusters in a variety of programs of different sizes and how can we categorize the programs more objectively in terms of their clusterization?
- RQ2 How typical are cluster structures that are held together by at most a few clearly identifiable procedures (linchpins), *i. e.* whose removal could reduce program clusterization significantly?
- RQ3 Currently the exact linchpins can be determined by brute-force only, which is infeasible for bigger programs; how closely can low-cost heuristic methods approximate linchpins?

III. Experiments setup

We collected a set of programs that served as the subjects following these principles: the set had to be comparable to other researchers' and our own previous results, our existing tools had to be able to handle them with ease, the programs had to be from different domains, and their sizes had to vary in a wide range. Based on these requirements, we fixed the language of the programs to C/C++, and in the first instance started with the collection of programs Harman *et al.* used in their experiments [6]. We could reuse 60% of these programs but also extended this set to finally arrive at 29 programs written in C (we will refer to this set as the *moderate size* programs). The basic properties of these programs can be seen in the first three columns of Table I. We provide names, lines of code (LOC) and the number of procedures (NP). The purpose of the other columns will be explained later.

TABLE I
Moderate size subject programs with clusterization information, sorted by Visual class and NP

Program name	LOC	NP: # of procedures	Visual class	Clusterization metrics AREA ENTR REGU REGX
lambda	1766	104	▼ low	
epwic	9597	153	▼ low	
tile-forth	4510	287	▼ low	
a2ps	64590	1040	▼ low	
gnugo	197067	2990	▼ low	
time	2321	12	■ med	
nascar	1674	23	■ med	
wdiff	3936	29	■ med	
acct	7170	54	■ med	
termutils	4684	59	■ med	
flex	22200	153	■ med	
byacc	8728	178	■ med	
diffutils	17491	220	■ med	
li	7597	359	■ med	
espresso	22050	366	■ med	
findutils	51267	609	■ med	
compress	1937	24	▲ high	
sudoku	1983	38	▲ high	
barcode	5164	70	▲ high	
indent	36839	116	▲ high	
ed	3052	120	▲ high	
bc	14370	215	▲ high	
copia	1168	242	▲ high	
userv	8009	255	▲ high	
ftpd	31551	264	▲ high	
gnuchess	18120	270	▲ high	
go	29246	372	▲ high	
ctags	18663	535	▲ high	
gnubg	148944	1592	▲ high	

The second part of our data set consisted of two large industrial software systems from the open source domain. The first one was the WebKit system, which we have already used in some of our previous investigations [15], [17]. WebKit is a popular open source web browser engine integrated into several leading browsers by Apple, KDE, Google, Nokia, and others [9]. It consists of about 2.2 million lines of code, written mostly in C++, JavaScript and Python. In this research we concentrated on C++ components only, which attributes to about 86% (1.9 million lines) of the code. In our

measurements we used the Qt port of WebKit called QtWebKit on x86_64 Linux platform. We performed the analysis on revision $r91555$, which contained 91,193 C++ functions and methods as the basic entities for our analysis.

The other large system we used was the GNU Compiler Collection (GCC), the well-known open source compiler system [8]. It includes front ends for C, C++, Objective-C, Fortran, Java, Ada, and Go, as well as libraries for these languages. The GCC system is large and complex, and its different components are written in various languages. It consists of approximately 200,000 source files, of which 28,768 files are in C, which was the target of our analysis. In terms of lines of code, this attributes to about 13% of the code, 3.8 million lines in total (note that in C, the size of individual functions is usually larger than that of an average C++ method, hence this difference in lines of code compared to WebKit). We chose revision $r188449$ (configured for C and C++ languages only) for our experiments, in which there were 36,023 C functions as the basic entities for our analysis.

In our experiments we used our custom build tools as well as some existing components. To extract base program representations, as parser front ends we used Grammatech CodeSurfer [18] in the case of moderate size programs and Columbus [19] for the big programs. For the SEA dependence computation, our existing implementation of the SEA algorithm using ICCFG graphs [13] was applied. The benefit of using Columbus with ICCFG graphs for the bigger programs is that it is more scalable due to higher granularity of analysis.

We modified the SEA computation tool by adding the capability to ignore one or more procedures, which was required for linchpin determination. Since we needed to process and store a large number of dependence sets, we implemented additional tools (for MSG computation, cluster metrics computation, etc.) that employ efficient specialized data structures and algorithms. The calculation of the final results and the whole measurement procedure was performed using shell scripts.

We will describe our set of experiments and additional details regarding the measurements and tools in the corresponding sections.

IV. Existence of Dependence Clusters

A. Identification of clusters

To obtain the dependence clusters first we needed to compute the SEA-based dependence sets for all procedures in our subject programs. Similarly to the definition of slice-based dependence clusters, we regard two procedures to be in the same SEA-based cluster if their dependence sets coincide. This is a sensible definition because the SEA relation is reflexive, so if two procedures have the same dependence set, then they depend on each other as well. Note, that we did not apply the approximation of comparing dependence set sizes only as other studies suggested [2], [6].

In the following, we will investigate the *clusterization* of programs, which is the extent they exhibit dependence clustering. To express clusterization we followed two approaches:

Visual classification is carried out by (subjective) visual inspection of the MSGs, and assigns one of three levels to each program: *low*, *medium* and *high*.

Clusterization metrics are rigorously defined measures that are designed to express clusterization in easily quantifiable numerical form (values from $[0, 1]$).

For the moderate size programs, the fourth column (Visual class) of Table I shows the results of the visual classification we performed by inspecting the MSGs of the programs. As an illustration, Figure 1 shows MSGs of programs that are typical for each category. A cluster reveals itself as a wide plateau consisting of a number of equal-sized dependence sets. Typically, in a low class we cannot identify any plateaus, while for the high class there are one or two big ones, the rest being medium.[1] Visual classification reveals the following:

- The first program (`epwic`) does not show any plateaus, the landscape ascends in small increases.
- The second one (`findutils`) contains some moderately wide plateaus. They are not significant individually, but altogether cover much of the width of the landscape.
- For the last program (`gnubg`) we can see a single plateau occupying nearly the whole width of the landscape.

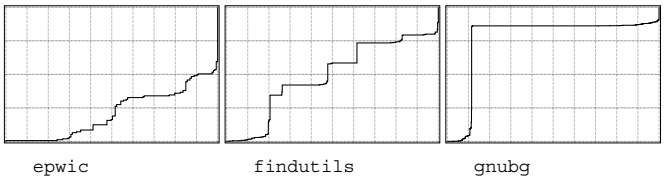


Fig. 1. Example MSGs for the visual classification (`epwic`: *low*, `findutils`: *medium*, `gnubg`: *high*)

As can be seen in Table I, overall we found 5 programs to be low, 11 medium, and 13 highly clusterized.

B. Measuring clusterization

Beyond visual interpretation, we will need an exact numerical expression (metric) of the level of clusterization and its relative change for two reasons. First, this way an automatic classification of programs could be made with the help of appropriate thresholds. Second, the metrics can be applied for the analysis of linchpins and measuring the effect of their removal (discussed in subsequent sections).

The obvious choice of metric to be used in these kinds of experiments is based on Binkley and Harman's work [2], who measured the area under MSG and used the change of this metric to analyze linchpins (we also rely on this metric and denote it by AREA in the following). The apparent weakness of AREA is that it increases if all dependence sets are increased by the same amount, although intuitively clusterization should not be different in such cases. Programs with no dependence clusters can have both small and large dependence sets, and vice versa.

[1]Note, that the same dependence set size may incidentally correspond to different sets, however this is very unlikely except for the smallest sets.

We experimented with alternative metrics to express clusterization in programs that are independent of the actual dependence set sizes and could reflect this property, these are presented below in a more formal way.

We define the measures so that they are comparable to each other and yield a value in the interval $[0, 1]$. For all metrics, 0 is set to mean that the level of clusterization is close to none, while 1 means that clusterization is maximal.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of procedures in a program X (for simplicity, we assume $n \geq 2$). The SEA relation of program X is a binary relation defined on its set of procedures, *i. e.* $SEA \subseteq P \times P$. With $\overline{n} = \{0, 1, \ldots, n\}$ we give the following auxiliary definitions:

$SEA(p)$ is the dependence set of a procedure $p \in P$ defined as

$$SEA(p) = \{q \in P \mid SEA(p, q)\}$$

The weight function w gives the sizes of the dependence sets as

$$w : P \to \overline{n}, \quad w(p) = |SEA(p)|$$

Formally, the clusterization of a program based on SEA dependences is in fact a partitioning of the procedure set P (not being transitive, the SEA relation itself does not exhibit partitions). We define two kinds of clusters, first by assigning two procedures to the same cluster (partition) if they have the same dependence set, *i. e.* they have the same SEA image:

$$\mathcal{I} = \big\{\, \{q \in P \mid SEA(q) = SEA(p)\} \;\big|\; p \in P \big\}$$

The second kind of cluster is defined by considering only the sizes (weights) of the dependence sets of the procedures:

$$\mathcal{S} = \big\{\, \{q \in P \mid w(q) = w(p)\} \;\big|\; p \in P \big\}$$

For any $c \in \mathcal{S}$ the weight of its members are equal, so we can assign the same weight to cluster c itself. Clearly $SEA(q) = SEA(p)$ implies $w(q) = w(p)$, so $\mathcal{I}$ is a refinement of $\mathcal{S}$. This also means that the weight function can be extended naturally to $\mathcal{I}$ as well.

Now we can define the different clusterization metrics as follows (the metrics always have to be interpreted in the context of a given program).

Consistently with earlier descriptions, AREA has three equivalent definitions:

$$\text{AREA} = \frac{1}{n^2} \sum_{p \in P} w(p) = \frac{1}{n^2} \sum_{c \in \mathcal{I}} |c| \cdot w(c) = \frac{1}{n^2} \sum_{c \in \mathcal{S}} |c| \cdot w(c)$$

Our next metric is based on an analogy of *entropy* and measures the "(dis)order" in the system of dependence sets in terms of their sizes. We consider a program more clusterized in this respect if there is a greater number of equal-sized dependence sets, *i. e.* when the entropy is lower (note, that this inverse relationship is required to obtain comparable metric intervals with the other metrics). Our entropy-based clusterization measure is formally defined as:

$$\text{ENTR} = 1 - \frac{\sum_{c \in \mathcal{S}} f(c) \cdot \log_2 f(c)}{\log_2 1/n}, \text{ where } f(c) = \frac{|c|}{n}$$

Finally, our two metrics referred to as *regularity* metrics are based on the number of partitions. The idea is that the fewer partitions there are, the larger their size must be, so there have to be more large clusters among them. Inversely, more partitions have to take more "regular" different sizes hence they will represent low clusterization. This metric has two variants, the first is based on $\mathcal{S}$, the other (extended, REGX) is based on $\mathcal{I}$.

$$\text{REGU} = \frac{n - |\mathcal{S}|}{n - 1} \qquad \text{REGX} = \frac{n - |\mathcal{I}|}{n - 1}$$

As noted earlier, all metrics are normalized (*i. e.* their value is a real number from the interval $[0, 1]$), which is useful to be able to compare the metrics of programs with different size to each other.

We computed all four metrics for all of our moderate size subject programs and compared the rankings of the procedures based on these individual metrics to the visual classification of the programs. These values are shown in the last four columns of Table I (to ease interpretation, the gray areas inside the small rectangles are set to be proportional to the metric values). Note that the ordering of the programs in this table was done based on the visual ranking first, then on the number of procedures inside each rank group.

Visual clusterization created three groups with 5 (low), 11 (medium), and 13 (high) elements, respectively. We would expect an ideal clusterization metric to yield values in such a way that the 5 smallest would be assigned to the "low" level, the middle 11 would be assigned to "medium", and the largest 13 to the "high" level group. Based on these criteria, the clusterization metrics can be characterized by counting how many programs they fail to assign to the group given by visual ranking. The counts are as follows: AREA $\to$ 10, ENTR $\to$ 7, REGU $\to$ 15, REGX $\to$ 8. The differences in these counts can also be observed by visual inspection of the metric values. It can clearly be seen that AREA and REGU are significantly worse than the other two metrics, while the difference is not so great regarding ENTR and REGX. ENTR is more precise on low and medium clusterization levels, while REGX performs better on highly clustered programs (see the last four columns of Table I).

Based on the above observations we will use ENTR and REGX to measure the degree of clusterization in the rest of the paper, and will mostly rely on ENTR where low or medium clusterization is concerned and use the other for high clusterization.

C. Dependence clusters in the big programs

So far, we have been dealing only with the moderate size programs, but our dataset contains two big programs as well, which need more thorough investigation. The MSGs for these two programs, GCC and WebKit, can be seen in Figure 2.

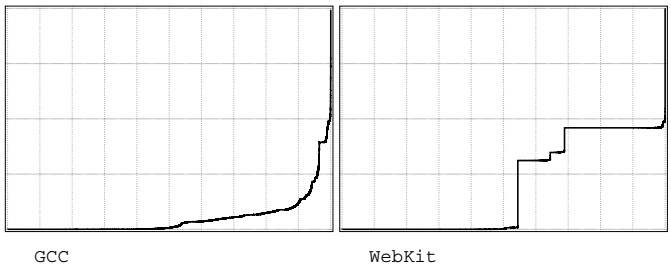


Fig. 2. MSGs for GCC and WebKit

The differences between the two programs are clear. GCC belongs to the low level clusterization category, while WebKit exhibits some clusterization (it would belong to the medium category in the visual ranking). The ENTR values are 0.4347 for GCC and 0.6980 for WebKit, while REGX is 0.3134 and 0.3552, respectively, which supports our initial (visual) classification for these two systems. While ENTR shows a notable difference, in the case of REGX it is not so significant, which may also reflect our finding from above that ENTR was better for low or medium clusterization.

It would be interesting if we could find any properties of these systems that justify their classification in terms of dependence clusters. In other words, what makes GCC not having significant dependence clusters as opposed to WebKit? In previous work [15], we analyzed the structure of source code and the dependences in WebKit in a slightly different context. After consulting with some key WebKit developers and showing them the members of the clusters, we came to the conclusion that clustarization is related to architectural concepts in the system.

We speculate that the most notable difference between the two systems in this respect is that while WebKit is essentially a library consisting of highly coupled elements for the distinct functional areas, GCC is a complex application but with much clear behavioural paths that are independent of each other. In WebKit, most complex functionalities are implemented in a set of highly interacting procedures (for example, webpage rendering is performed by several hundred procedures calling each other recursively). On the other hand, GCC implements functionalities like compiler optimization passes that are more isolated from each other. In addition, the two systems are written in different programming paradigms (C vs. C++) which may influence their internal structure. More detailed analysis of the causes for this difference remains for future work.

In the remaining parts of this paper we will be concerned with the identification of linchpins, however for programs of this size only the heuristic approaches are feasible. Hence we will subsequently use WebKit only to verify the effect of our heuristic methods, while GCC will play no role from now on.

V. LINCHPIN DETERMINATION

First, we identified the linchpins for our moderate size programs using the brute-force method that enumerates all procedures. As the biggest challenge in this topic is how to locate linchpins using more efficient methods, in the next experiment we investigated approximate heuristic methods for this task and compared their results to the exact results of the brute-force method. Since we could not apply the brute-force method to our large program WebKit, we verified the successfulness of the heuristic method on it in the final step.

A. Linchpin identification by brute-force

The simplest way to identify a possible linchpin in a program is to remove procedures one by one and see which one brings the biggest gain according to some metric. In the following, *gain* will mean the amount the respective metric is reduced in percentage: $\frac{m-m'}{m}$ [%], m being the original metric value and m' the value after linchpin removal.[2]

Specifically, we computed all SEA dependence sets for a program by ignoring the candidate procedure and all of its dependences. We compared the ENTR and REGX metrics of the reduced versions of the program to the corresponding metrics of the original program. This calculation was then repeated for all procedures in the program with these two metrics. For simplicity, we will present our results for REGX in the cases when the results were similar for both metrics, and will note explicitly in other situations. For the purposes of the remaining discussion, the procedure that caused the biggest reduction in the REGX metric was considered to be the linchpin.

TABLE II
REGX THRESHOLDS FOR LINCHPIN PROCEDURES

	Minimum gain	Maximum gain	Typical gain
Low clusterization	5%	20%	—
Medium clusterization	18%	55%	20%
High clusterization	13%	99%	37%

Table II summarizes results of linchpin determination for different clusterization classes of moderate sized programs. It shows how much reduction in the REGX clusterization value could be attained in the worst case (*minimum gain*) for a given clusterization class, and similarly how much was the *maximum gain*. It also indicates how much reduction could be attained if the few outlier programs with least gain—4 in the medium and 3 in the high class—are ignored (*typical gain*).

One would expect that programs with low clusterization do not contain linchpins, *i. e.* there is no procedure whose removal significantly reduces the already low clusterization. This was not entirely supported by our findings, as there were cases when as much as 20% gain could be achieved. This is not negligible, but as noted earlier, REGX performs better at high clusterization levels, so results for the low clusterization level are not entirely relevant. However, the achieveable gain is definitely larger for the medium and high classes, and gains for highly clusterized programs vary widely.

In Table III, we listed the results of linchpin calculation for moderate size programs with high clusterization. To get these

[2]Note, that we do not actually *refactor* linchpins and get equivalent programs but we merely remove the procedures in order to identify them.

results we had to compute over 15 million SEA sets altogether, but this was possible to complete in hours on an average server machine.

TABLE III
LINCHPINS FOR MODERATE SIZE PROGRAMS WITH HIGH CLUSTERIZATION

Program	REGX gain	Procedure name
barcode	57%	`Barcode_Encode`
bc	37%	`dc_func`
compress	37%	`compress`
copia	99%	`scegli`
ctags	13%	`createTagsForFile`
ed	53%	`exec_command`
ftpd	43%	`parser`
gnubg	52%	`HandleCommand`
gnuchess	53%	`main`
go	14%	`get_reasons_for_moves`
indent	47%	`indent_main_loop`
sudoku	41%	`rsolve`
userv	22%	`parser`

The second column shows the REGX gain after linchpin removal, which was quite significant (at least 37%) in almost all of the cases, 43.7% on average for this class of programs. The last column of the table shows the names of the respective procedures identified (which, except for `compress`, `gnubg` and `go`, were the same for ENTR as well). It is interesting to observe that, as names themselves suggest and a manual analysis of the programs confirms, most of the identified procedures indeed have central role in the programs. It is an open question, however, how many of these procedures could be deemed responsible for avoidable dependence clusters, in other words, dependence pollution [2]. Expectedly, procedures acting as the main procedures could not be easily refactored.

B. Heuristic determination of linchpins

We estimated that the brute-force method to determine the potential linchpin for the WebKit system would take about 70 years to complete using our strongest servers. So, obviously, we must find alternative methods to find (or at least approximate) the linchpins to enable practical application of dependence cluster related research.

The existence of dependence clusters and any related linchpins are determined by the structure of the dependences under investigation (SEA and the underlying ICCFG program representation in our case). Therefore, it is to be expected that by investigating the topology of the underlying dependence graph one could gain insight into what makes a program point a potential linchpin.

The problem does not have an obvious solution, so we wanted to investigate whether local properties of the dependence graph nodes (procedures) could be leveraged to approximate lichpins. We used the following heuristic metrics as potential indicators: NOI (Number of Outgoing Invocations from the procedure), NII (Number of Incoming Invocations to the procedure), sum of the former two (SOI=NOI+NII), and their product (POI=NOI·NII). We tried the sum and the product because we expected that in linchpin formation both incoming and outgoing dependences could be important.

To compare the actual linchpins identified by the brute-force method to the performance of the heuristic metrics, we related two values for each procedure in the programs: a clusterization metric (ENTR or REGX) after removing the procedure and one of the heuristic metrics (NOI, NII, SOI, POI) associated with the procedure. Then we used Pearson and Kendall correlation checks between the corresponding vectors of these values. We do not provide detailed data for these measurements because they all pointed out the same best heuristic estimation.

TABLE IV
PEARSON CORRELATION BETWEEN HEURISTIC METRICS AND THE ENTR AND REGX METRIC. UNDERLINED NUMBERS INDICATE STRONGEST CORRELATION IN THE CORRESPONDING BLOCK.

	ENTR				REGX			
Program	**NOI**	**NII**	**SOI**	**POI**	**NOI**	**NII**	**SOI**	**POI**
a2ps	-0.27	0.03	-0.16	-0.04	-0.57	-0.01	-0.39	-0.40
acct	-0.67	0.21	-0.52	-0.53	-0.67	0.13	-0.57	-0.46
barcode	-0.59	0.07	-0.55	-0.65	-0.71	0.06	-0.66	-0.74
bc	-0.72	0.04	-0.56	-0.57	-0.75	0.05	-0.58	-0.59
byacc	-0.11	-0.01	-0.08	-0.20	-0.72	0.05	-0.42	-0.40
compress	-0.72	0.04	-0.63	-0.49	-0.89	-0.09	-0.85	-0.63
copia	-0.72	-0.66	-0.98	-1.00	-0.70	-0.68	-0.98	-1.00
ctags	-0.42	0.03	-0.18	-0.23	-0.53	0.04	-0.23	-0.24
diffutils	-0.42	-0.02	-0.36	-0.51	-0.66	-0.02	-0.56	-0.57
ed	-0.67	0.03	-0.49	-0.56	-0.82	0.04	-0.59	-0.62
epwic	0.28	0.12	0.32	0.32	-0.50	-0.02	-0.48	-0.32
espresso	-0.55	0.03	-0.33	-0.46	-0.70	0.04	-0.42	-0.43
findutils	-0.25	0.07	-0.20	-0.04	-0.34	0.07	-0.29	-0.01
flex	-0.79	0.07	-0.70	-0.54	-0.88	0.08	-0.78	-0.62
ftpd	-0.74	0.03	-0.53	-0.40	-0.78	0.02	-0.57	-0.42
gnubg	-0.66	-0.07	-0.55	-0.68	-0.69	-0.07	-0.57	-0.71
gnuchess	-0.54	0.07	-0.47	-0.31	-0.55	0.06	-0.48	-0.29
gnugo	-0.45	0.04	-0.06	0.01	-0.53	-0.01	-0.13	-0.05
go	-0.49	0.03	-0.16	-0.31	-0.58	0.04	-0.18	-0.33
indent	-0.64	0.04	-0.45	-0.17	-0.69	0.05	-0.48	-0.16
lambda	0.30	0.53	0.50	0.58	-0.61	-0.49	-0.64	-0.57
li	-0.07	-0.17	-0.18	-0.18	-0.09	-0.15	-0.17	-0.18
nascar	-0.13	-0.18	-0.23	-0.41	-0.77	0.15	-0.76	-0.33
sudoku	-0.69	0.22	-0.26	-0.40	-0.79	0.20	-0.35	-0.52
termutils	-0.35	0.18	-0.21	-0.13	-0.46	0.17	-0.33	-0.20
tile	0.48	0.46	0.62	0.63	-0.27	-0.18	-0.29	-0.28
time	0.70	-0.29	0.47	0.70	-0.55	0.08	-0.47	-0.12
userv	-0.49	0.02	-0.35	-0.40	-0.57	0.04	-0.39	-0.39
wdiff	0.04	-0.23	-0.02	-0.50	-0.89	0.18	-0.89	-0.66
average	-0.36	0.03	-0.25	-0.26	-0.63	-0.01	-0.50	-0.42
strongest	17	0	1	11	23	0	2	4

In Table IV, we show Pearson correlation results for all programs. We marked the strongest correlation values for each program underlined; the last row shows the average correlation values for each metric. It can clearly be seen that the NOI metric (Number of Outgoing Invocations) is the best estimator for both ENTR and REGX. The best values are negative in the NOI columns, which means that for the procedures of a program there is a high correlation between a high NOI value and a low clusterization value resulting from the removal of that procedure. In other words, the higher NOI value a procedure has, the more likely it is that its removal would decrease the clusterization considerably, *i. e.* the more likely it is that the procedure is a linchpin.

In the case of ENTR and REGX metrics, in 59% and 79% of the cases NOI showed the strongest correlation; the average correlation was -0.36 and -0.63 (with standard deviations 0.4 and 0.18), respectively. The second best was POI showing strongest correlation in 38% and 14% of the programs with average correlation values -0.26 and -0.42. NII performed

poorly, which was surprising because we expected NOI and NII will perform similarly. The promising results for NOI are strengthened by the fact that *the highest NOI value predicts a linchpin correctly in most of the cases*: in the highly clustered group in 12 out of 13 programs, in the medium group in 7 out of 11 programs the procedure with the highest NOI value turned out to be a linchpin. What causes NOI to be the best estimator is not yet clear, we are going to investigate this question in the future.

As a statistical test to support our choice for NOI, we make a null-hypothesis that the other three metrics are at least equally good. For instance, consider NOI and NII with ENTR measure. Out of the 28 programs, in 22 instances NOI has stronger correlation than NII. Using Chernoff's bound we get that the probability of NII being at least as good as NOI is at most 0.01035. Chernoff's bound shows that NOI is in fact better with a probability of at least 0.9235 than any of the other three with respect to any of the considered measures.

Another interesting observation we made about the data is that for smaller programs the agreement between the NOI metric and both clusterization metrics was slightly better, suggesting that this heuristic will perform better for smaller programs. Figure 3 shows how the correlation values between NOI and ENTR as well as NOI and REGX change as a function of program size. Only programs with high clusterization are shown because in the other cases the relationship was not so evident. Particularly, from left to right, we can see the average correlation values for the programs ordered increasingly by their number of procedures. Although not drastically, but a tendency of worsening correlation can clearly be observed. This could also indicate the need for combined identification of linchpins as outlined at the end of this section. The exact causes of this phenomenon are not clear yet, they are probably related to the different topologies of small and bigger programs.

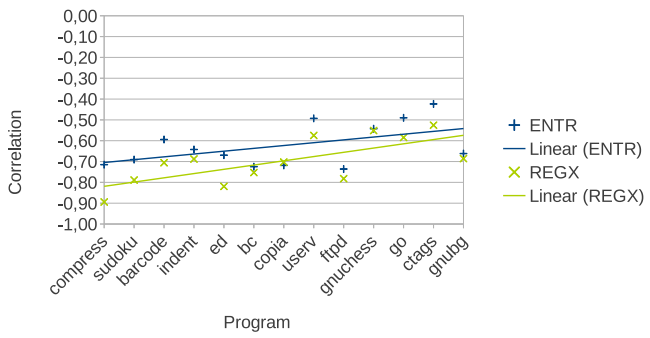


Fig. 3. Correlation change with program size of highly clusterized programs

Once we got these results about the best linchpin estimator heuristic metric, we applied it to WebKit to see whether we can achieve significant ENTR or REGX metric reduction and hence potentially find linchpins in that system too. In the first instance we calculated the NOI metrics for WebKit and applied the filtered dependence set calculation excluding the first 10 procedures with highest NOI values individually, thus obtaining a set of 10 clusterization reduction values. Unfortunately, after this experiment we could not observe any notable improvement in clusterization: even the largest ENTR and REGX reductions were negligible and visual inspection could not reveal anything either. Then we tried the other heuristic metrics as well in a similar way, but we got even worse results, so we decided to continue the research with the combined exclusion of procedures as detailed below.

C. Reducing clusterization by sets of linchpins

Linchpin identification and removal can be rephrased in graph theoretical terms as well. Given a graph G on an n element vertex set V, we say that $S \subset V$ is a separator set, if $G \setminus S$ contains only small connected components. If G does not contain certain substructures (large excluded minors, to be precise), then it must contain a small separator set. Still, in many cases the size of the separator set grows with n, usually it is about $O(\sqrt{n})$ [20].

While clusters in a SEA graph (or other graphs associated with a program) are more complex than connected components (for instance, most dependence graphs including SEA are not transitive), it is easy to see an analogy between the two kinds of problems. We think that analogously to separation of graphs, one cannot always expect to find a single linchpin vertex whose removal can significantly decrease clusterization. Rather, one should look for a hopefully small subset of vertices that somehow glue together the graph, and deleting them results in small clusters.

Graph theory also tells that not every graph has a small separator, for example, one has to delete a large number of vertices from a so called expander graph in order to make it disconnected. We expect that the graphs associated with programs are not expander graphs, and therefore the deletion of a relatively small subset can reduce clusterization. Finding such a linchpin set effectively seems to be challenging.

To verify this theoretical concept, we performed empirical measurements with sets of linchpins as opposed to only one on some representative programs from our moderate size subjects. In this series of measurements, three programs from the medium level clusterization group (`findutils`, `termutils`, `nascar`), and three other from the high clusterization group (`go`, `ed`, `sudoku`) were selected. Moreover, care was taken to include programs with different sizes in both groups. We took the first 10 procedures of a program with the highest NOI values, performed the calculation of the dependence sets while ignoring the first 0, 1, 2, ..., 10 procedures together and investigated the resulting clusterization metrics. We were interested to see whether there was indeed one linchpin that brought significantly more gain than the followers, or were the differences not so significant between these first 10 candidates.

Figure 4 shows the ENTR metric for the cumulative removal of the first k procedures (k-element linchpin sets), where the different k values are represented on the horizontal axis.

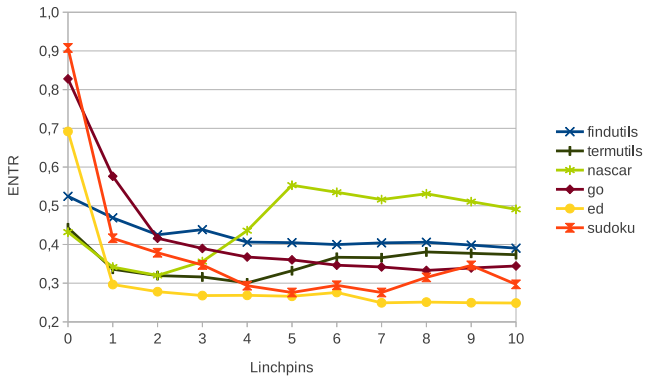


Fig. 4. Changing of the gains for the first 10 linchpins

One can observe that at most the sets with the first three procedures are notable and require further investigation. It can also be observed that for the three programs of medium level clusterization, the overall decrease in the metric is less than for the other group, as expected. More interestingly, it seems that regardless of the level of clusterization, program size plays an important role in the rate of decrease. Consider programs `sudoku` and the ten times larger `go`, for instance. We can see that for `sudoku` a significant decrease of clusterization occurs right after the first procedure, while for `go` even the second procedure contributes to the decrease significantly. As an example from the other group, the same effect can also be observed for `nascar` and `findutils`. The anomalous behavoiur of increasing clusterization can be expected in certain cases, the effect is more pronounced in the case of `nascar` due to its small size.

D. Linchpins in WebKit

Findings from above suggest that in many cases, especially for large programs, not only one program element (procedure) could be responsible alone for the formation of dependence clusters but a set of program elements together.

We performed similar experiments with the WebKit system in the hope to identify linchpin sets that significantly reduce clusterization. We expected that it would need more than only 2-3 procedures to achieve the same effect but we did not know how many. We tried removing several first procedures with the biggest NOI values but could not observe significant change in the clusterization up to until we removed about the first 200-250 procedures together. Removing 256 methods with the highest NOI values resulted in ENTR reduced by 7.2%, and REGX metric value reduced by 19.1% (AREA was reduced by 32.2%). In other words, the dependence sets collapsed this way, also changing dependence cluster formations, which can be observed in Figure 5.

Here, we can compare the original dependence clusters and the ones after removing these procedures (note, that in both cases the total number of procedures are represented on both axes, which is in the second case smaller by a comparatively

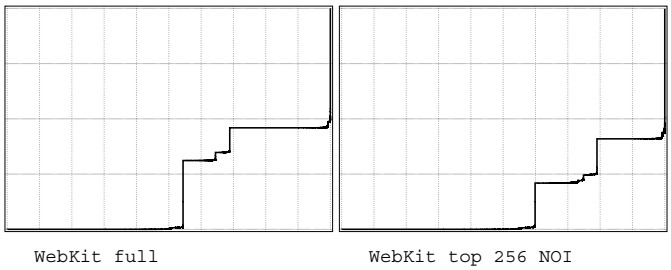


Fig. 5. WebKit MSGs before and after removing top 256 NOI procedures

neglible amount, hence the graphs are still comparable to each other). Although we cannot state that the clusters completely disappeared, the change is significant. To check if this result can indeed be attributed to the special role of the procedures with the top 256 NOI, we performed validation tests with three sets of randomly chosen 256 procedures. We found that randomly filtering procedures does not bring any improvement to the clusterization, change in ENTR was 0.04%, in REGX it was 0.25% on average.

It remains for future work to analyze in depth these linchpin sets and their effect on clusterization. From preliminary investigation, it seems that the clusters did not really disappear, they just changed their structure. Some of the procedures with high NOI could represent some very general connecting procedures, which do not really bear significant functionality (for example, we noticed a procedure that consisted of only a big switch statement and a huge number of method calls). When removing such procedures, the core dependences responsible for the main functionalities will remain, although the overall size will be smaller.

VI. Threats to Validity

We believe that the range of subject programs we used is representative for C/C++ as it ranges over various sizes and the domains are different. However, it would be important to see whether these findings can be generalized to other languages and types of dependences. The imprecision of our SEA-based dependences could slightly distort the results due to dependences that could have been avoided using a more precise method. However, as previous research showed [10], such false dependences are expectedly tolerable. We did not investigate if similar results would have been obtained using different, for instance, slice-based clusters.

We generalized our findings regarding the heuristic method based on the NOI metric to the WebKit system. We could not verify how good this heuristic actually performs on this system as we do not know the exact linchpin data, we could only state that some improvement has been obtained.

Our findings related to the presence of linchpin sets instead of individual linchpins showed that this is more probable with bigger programs. However, this highly depends on the system itself so this observation should be generalized with caution.

VII. Discussion and conclusions

This paper presented an empirical investigation of SEA-based dependence clusters. We may now answer the research questions set forth in this paper, however, our contributions to the better understanding of dependence clusters raised a number of additional questions.

Answering RQ1, we found that dependence clusters occur frequently in programs regardless of their domain and size, however there are also programs which exhibit very little clusterization. We gave precise definitions for four clusterization metrics and used them throughout to evaluate the results of our experiments. The results so far enable us to expect that the clusterization of programs can be measured with greater precision in the future. Additionally, we plan to experiment with more complex metrics that are less sensitive to the cluster and dependence set sizes.

RQ2 dealt with identifying one or more linchpins in programs. We were able to identify linchpin procedures using the brute-force method for all moderate size programs, but it is still to be explored in which cases we must seek for multiple linchpins and not only one. What we found is that as the program size increases it is less probable that only one program element is responsible for the formation of clusters. Note, that in this work we did not systematically investigate to decide if a dependence cluster reflects a dependence pollution [2] or not, and if the associated linchpins could be actually refactored, which is one of our future research directions.

Exhaustive exploration of possible linchpin combinations is computationally even more hopeless than for a single case. Hence additional, more sophisticated heuristic methods should be explored. Specifically, analyzing certain properties of the dependence graphs in terms of the graphs' topology is promising. Answering RQ3, we found that a specific metric based on graph structure, the Number of Outgoing Invocations for a procedure (NOI) is quite a good heuristic estimator for the linchpin, and the highest NOI value indeed predicts linchpins correctly in most of the cases, so in the short term we plan to investigate what made NOI the best in these experiments. Investigating other more sophisticated but still efficient methods is among our future plans as well, including the use of machine learning classifiers based on graph topology properties. Involving the hierarchical structure of the programs (packages, classes and methods) could also be promising.

We are still looking for the causes of the dependence clusters in WebKit. Although our heuristics gave interesting insights into the structure of these clusters, further investigation of the internals of this software is required. For this task we will consult WebKit developers, who already noticed that there are some elements in the identified clusters which they cannot explain. We need to investigate whether these are the consequence of the imprecision of the SEA algorithm or they represent some more hidden dependences in the system.

Acknowledgements

The authors would like to thank Zoltán Herczeg, László Langó, Csaba Osztrogonác, John Taylor and Béla Vancsics for their supporting work in this research. This research was supported by the Hungarian national grant GOP-1.1.1-11-2011-0049.

References

[1] D. Binkley, "Source code analysis: A road map," in *Proceedings of 2007 Future of Software Engineering (FOSE'07)*. IEEE Computer Society, 2007, pp. 104–119.

[2] D. Binkley and M. Harman, "Locating dependence clusters and dependence pollution," in *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 177–186.

[3] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proceedings of the 33rd ACM SIGSOFT International Conference on Software Engineering (ICSE)*, 2011, pp. 746–765.

[4] D. Binkley and M. Harman, "Identifying 'linchpin vertices' that cause large dependence clusters," in *Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*, 2009, pp. 89–98.

[5] D. Binkley, M. Harman, Y. Hassoun, S. Islam, and Z. Li, "Assessing the impact of global variables on program dependence and dependence clusters," *Journal of Systems and Software*, vol. 83, no. 1, pp. 96–107, 2010.

[6] M. Harman, D. Binkley, K. Gallagher, N. Gold, and J. Krinke, "Dependence clusters in source code," *ACM Transactions on Programming Languages and Systems*, vol. 32, no. 1, pp. 1–33, Nov. 2009.

[7] S. Black, S. Counsell, T. Hall, and D. Bowes, "Fault analysis in OSS based on program slicing metrics," in *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE Computer Society, 2009, pp. 3–10.

[8] "GCC, the GNU Compiler Collection," http://gcc.gnu.org/, last visited: 2013-05-08.

[9] "The WebKit open source project," http://www.webkit.org/, last visited: 2013-05-08.

[10] J. Jász, Á. Beszédes, T. Gyimóthy, and V. Rajlich, "Static Execute After/Before as a replacement of traditional software dependencies," in *Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM'08)*, 2008, pp. 137–146.

[11] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, 1984.

[12] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–61, 1990.

[13] Á. Beszédes, T. Gergely, J. Jász, G. Tóth, T. Gyimóthy, and V. Rajlich, "Computation of Static Execute After relation with applications to software maintenance," in *Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07)*, 2007, pp. 295–304.

[14] Á. Hajnal and I. Forgács, "A demand-driven approach to slicing legacy COBOL systems," *Journal of Software: Evolution and Process*, vol. 24, no. 1, pp. 67–82, Jan. 2012.

[15] L. Schrettner, J. Jász, T. Gergely, Á. Beszédes, and T. Gyimóthy, "Impact analysis in the presence of dependence clusters using Static Execute After in WebKit," in *Proceedings of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'12)*, Sep. 2012, pp. 24–33.

[16] S. Islam, J. Krinke, and D. Binkley, "Dependence cluster visualization," in *Proceedings of the 5th international symposium on Software visualization (SOFTVIS'10)*, 2010, pp. 93–102.

[17] Á. Beszédes, T. Gergely, L. Schrettner, J. Jász, L. Langó, and T. Gyimóthy, "Code coverage-based regression test selection and prioritization in WebKit," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*, 2012, pp. 46–55.

[18] "Homepage of GrammaTech's CodeSurfer," http://www.grammatech.com/research/technologies/codesurfer, GrammaTech, Inc., last visited: 2013-05-08.

[19] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy, "Columbus – reverse engineering tool and schema for C++," in *Proceedings of the International Conference on Software Maintenance*, 2002, pp. 172–181.

[20] K. Kawarabayashi and B. Reed, "A separator theorem in minor-closed classes," in *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, Oct., pp. 153–162.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

Characterization and Assessment of the Linux Configuration Complexity

Ahmad Jbara and Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel

***Abstract*—The Linux kernel is configured for specific uses by manipulations of the source code during the compilation process. These manipulations are performed by the C pre-processor (CPP), based on in-line directives. Such directives, and the interleaving of multiple versions of the code that they allow, may cause difficulties in code comprehension. To better understand the effects of CPP, we perform a deep analysis of the configurability of the Linux kernel. We found significant inconsistencies between the source code and the configuration control system. Focusing on the thousands of config options appearing in the source code, we found that their distribution is heavy-tailed, with some options having more than a thousand instances in the code. Such wide use seems to imply a massive coupling between different parts of the system. However, we argue that employing a purely syntactic analysis is insufficient. By involving semantic considerations, we find that in reality the coupling induced by the very frequent options is limited. Moreover, even at the syntactic level the adverse effects of CPP are limited, as there is little nesting and the expressions controlling conditional compilation are usually very simple. But it could be even better if the configuration system undergoes a clean up. On the other hand, we found that the code controlled by CPP is very heterogeneous and may exhibit intimate mingling with non-variable code. As a result the applicability of alternative mechanisms such as aspects is hard to envision.**

I. Introduction

While much attention in the wider software engineering community is (rightfully) directed towards other aspects of systems development and evolution, such as specification, design, and requirements engineering, it is the source code that contains the only precise description of the behaviour of a system. Thus ensuring that the source code is comprehensible is of prime importance. An often overlooked aspect of code comprehension is the use of pre-processors, which support various manipulations of the source code during the compilation process. The injection of pre-processor directives, and the inclusion of multiple alternative versions of the code, may be expected to exacerbate the code comprehension problem. To see if this is indeed the case, we performed a detailed analysis of the configuration control mechanisms used in the Linux kernel. In a nutshell, our findings indicate that at least in this system the situation is not so bad and the code remains largely comprehensible. Nevertheless, the configuration system could use a massive cleanup.

CPP (the C pre-processor) is a commonly used tool for expressing software variability, whereby different executables are built from common source code. It works by using directives for conditional compilation, so that during the build process the compiler can decide whether or not to compile certain code fragments, or select from among multiple versions of the same basic functionality. The conditional compilation directives typically use preprocessor constant definitions, which may be derived from configuration option values that were set at an earlier stage.

Despite its popularity and strength, CPP has been identified as problematic. In particular, the interleaving of flow control and conditional compilation directives, as well as its lexical nature, may make the code harder to understand and maintain [16], [8], [4], [20], [15], [1]. Even the CPP reference manual identifies many pitfalls, especially when using it for macro definitions [17].

This situation has spurred recent interest in how CPP is used and in possible alternatives [13], [1], [9], [10], [19]. For example, Liebig et al. have analyzed the use of CPP in 40 large open-source projects, and provide various statistics characterizing its use [9]. While work such as this provides a wide picture of CPP usage, it might miss on the details. For example, not every symbolic constant that appears in a conditional directive is relevant to variability management. We therefore set out to complement previous work with an in-depth analysis of one of the projects, namely the Linux kernel. Linux makes heavy use of conditional compilations due to the need for special customization to support different architectures and features. The version we analyzed was Linux kernel 2.6.32.3.

To study variability we focus on configuration options, which are the means for expressing the select feature set that should be included in a specific build. Importantly, we consider the entire set of config options of all architectures. Our first finding is that it is not so easy to identify the relevant config options, as there are significant inconsistencies between the source code and the configuration control system. Thus one must be careful regarding the goal of the study: if it is the control of configurations and variability, the data might be different than if it is the effect of configurability on the complexity of the source code.

After selecting the data we want to focus on, we study the use of config options in the source code. We found a few thousands of options. These options have a skewed distribution such that the 20/80 rule holds. Thus some options, members in the significant 20%, are used more than a thousand times,

An abbreviated preliminary version of this work was presented in the PhD forum of ICPC 2012.

978-1-4673-5739-5/13/$31.00

seeming to imply massive coupling between diverse parts of the kernel. However, such a conclusion is based on purely syntactic analysis. By adding semantic considerations, we found that the imposed coupling is in fact minor. Moreover, recall that a large number of options, the least significant 80%, are used only a few times, so their contribution to the code coupling is also limited.

Despite the relatively low coupling, CPP nevertheless does have adverse effects on the code. Our measurements showed that the average (geometric) scattering degree of config options is relatively high as well as the number of code variations within a file. Another issue is therefore the possibility of replacing CPP with other mechanisms. This is found to be problematic due to the intimate mingling of code snippets using CPP, and the heterogeneity of the variable code blocks, so mechanized alternatives such as aspects are questionable.

Section II describes the CPP tool, the Linux configuration process, and related work. Section III then lists our research questions. The bulk of our study is reported in Section IV, which characterizes config options, and Section V, which presents metrics for the complexity introduced by CPP. We discuss the results and summarize conclusions in section VI.

II. Background

A. The C Language Preprocessor

Preprocessing by CPP is the first phase in compiling C programs. CPP is a powerful tool for managing configuration and portability of software, and also provides useful features such as header inclusion and macro definition [17].

In our work the most important feature of CPP is the support for *conditional compilation*. This allows a single software base to be used to compile many variants. In each variant, some fragments of the original source code are included and others are excluded, based on conditional directives. The #ifdef directive includes the controlled code in the compilation if its argument is a defined CPP constant. The complementary #ifndef directive includes the fragment if the constant is *not* defined. The #if directive resembles the if statement of the C language: the controlled code is included iff the expression evaluates to non-zero. The expression may consist of CPP constant values, tests of whether they are defined, and logical operations. The #else and #elif directives can be used to provide alternatives to #if, #ifdef, and #ifndef.

When a CPP constant controls a configurable feature of the system we call it a *config option*. In the Linux system the config options, by convention, have a CONFIG_ prefix. When we talk about specific occurrences in the code we call them *config instance*s.

CPP constants are defined internally by an explicit use of the #define directive within the source code, or externally by flags to the compiler. For example, the *gcc* compiler uses the -D option to define a new CPP constant and the -U option to undefine it.

The Emacs tool [6] enables programmers to navigate the conditional directives, allowing them to view the code with its macros expanded, and to emulate the different configurations which are controlled by preprocessor variables.

```
1 menu "Processor type and features"
2 source "kernel/time/Kconfig"
3 config SMP
4   bool "Symmetric multi-processing support"
5   ---help---
6   This enables support for systems with more than one CPU.
```

Listing 1. Example of the definition of the SMP config option in a Kconfig file.

```
1 obj-$(CONFIG_GENERIC_ISA_DMA)+=dma.o
2 obj-$(CONFIG_USE_GENERIC_SMP_HELPERS)+=smp.o
3 ifneq ($(CONFIG_SMP),y)
4 obj-y += up.o
5 endif
6 obj-$(CONFIG_SMP) += spinlock.o
```

Listing 2. Examples of using config options in a makefile to control the set of files to compile.

B. The Linux Configuration Process

The Linux kernel may be configured to run on many different platforms and provide diverse sets of features by using config options [19]. We present the config options from two different points of view: developers, who define config options and integrate them into the code, and users, who use the config options to customize the kernel for their needs.

Developer View of Config Options. Initially, each config option is defined in the *Kconfig* system. The *Kconfig* system is a set of text files placed in kernel source directories where variability is needed. These files' names start with Kconfig. Their contents are definitions of config options. An example is shown in Listing 1.

After defining the config options, developers integrate them in the *Kbuild* system or in the source code files. The *Kbuild* system is a collection of makefiles which are responsible for building the system [3]. In makefiles, config options are used to control the compiling process; an example is shown in Listing 2. In source code the config options are used as constants in conditional compilation directives to control the inclusion or exclusion of code fragments. Examples are shown below in Listings 3 and 4.

As time passes and new versions of the system come into being some of the configuration options become redundant and should be removed from the system files. As we show below this is not always done.

User View of Config Options. In order to build the Linux kernel one must first specify the desired configuration. This is done by invoking the make tool with one of three variations: make config, make menuconfig, or make xconfig. Initially, the config tool reads the *Kconfig* system files to extract the menus, config options, and dependencies between the different config options. The extracted config options are presented to the user who is asked to select the desired options according to his needs. The difference between the three config tools is the user interface. The make config is a forward-only version

that enables configuration from scratch. menuconfig displays a menu and enables the user to selectively set the configuration, so there is no need to pass all the options one by one. The third version is GUI-based.

Once done, the config tool generates the .config file which is placed in the top directory of the kernel source. This file contains all the configuration options that were set by the user. It is a text file with a line for each option. Each line is a key-value pair with the format CONFIG_*key*=*value*. The *key* is the name of the option, and *value* is y to indicate that a component will be built into the kernel, or m to indicate that it will be built as a loadable module [2]. Options that were not set are commented-out with # or deleted from the file.

The next stage is building the Kernel with the make command. At the very beginning of the build process the autoconf.h file is created. This file contains CPP definitions for the config options that were included in the .config file. To make these definitions available during compilation of the source code files the compiler uses its -include option. A few source code files explicitly use the inclusion mechanism of the preprocessor.

To shorten the configuration process and make it efficient the kernel source is provided with a preset configuration for each of the architectures that the kernel supports. These default configurations are kept in the arch/* subdirectories, and are called defconfig*X*, where *X* typically indicates the architecture and the developer who created the file. To utilize these defaults one should rename the default configuration file to be .config prior to the configuration process and place the new .config file in the top level directory, or pass the name of the default configuration file as an argument to the make tool.

C. Related Work

The use of CPP has attracted significant interest in recent years, Mostly related to configurability and the creation of product lines. We are specifically concerned with the resulting complexity and cognitive load on developers.

Early work also considered the ill-effects of using CPP. Spencer and Collyer [16] claim that careless use of #ifdefs is usually considered a mistake. They presented alternatives to conditional compilations, applied to their C News Package, and reduce the use of CPP by using clean interfaces and information hiding. Krone and Snelting [8] also claim that the use of conditional directives makes the code hard to understand even for experienced programmers. They suggested a visual tool which infers the configuration structure of source code, and makes it easy to discover violations of software engineering principles such as high cohesion and low coupling.

Favre [4] stated that heavy use of CPP directives can lead to unreadable programs, and makes maintenance and tool building hard. Vidács and Beszédes [20] also believe that heavy use of preprocessor directives causes problems of code comprehension due to the gap between what the programmer sees and what the compiler gets. They suggested using a tool, CANPP, for producing preprocessor schemas that can be used for information extraction such as original source, preprocessed files, and intermediate states.

The impaired readability and reduced reusability of conditional directives were also the motivation for developing C-CLR [15]. They identified the macro conditional redefinition and composition of multiple configuration options as problematic. The C-CLR tool improves readability by enabling users to perform configuration-specific navigation and enables reusability by automated identification of equivalent blocks.

Sutton and Maletic [18] presented a common configuration architecture for managing portability among three packages which were examined in their study. They also introduce configuration management patterns which they observed, including naming conventions, replaceable and parameterized inclusion, and compiler abstractions.

Based on [16] the authors of ASTEC [12] claim that macros are difficult to analyze and are error-prone. They present an alternative language which eliminates many of the potential errors. This is a syntactic language, as opposed to CPP which is purely lexical. It preserves the configurations of the program for analysis tools, while CPP produces a one-configuration program. Also, it enables tools to check errors before macros are expanded.

Liebig et al. [9], [10] studied the configurability of 40 open-source projects, including Linux. They found that 23% of the code is variable, there is no correlation between a system's size and the complexity of variable code, variable code is mostly heterogeneous which makes the use of AOP inapplicable, and the #ifdefs are mostly used in a high level granularity, enclosing entire entities such as functions and control statements.

Linux is the third-largest software system analyzed by Liebig et al. [9], both in terms of lines of code and in terms of the number of CPP constants (the two bigger ones are also operating systems: OpenSolaris and FreeBSD). Reynolds et al. [13] also studied Linux, focusing on the config options of one architecture (*i386*). In contrast, we consider the entire set of config options of all architectures. Tartler et al. [19] studied the consistency of using configuration options in Linux using an automated tool, and found 147 confirmed bugs.

Czarnecki et al. also studied the configuration system of Linux and transformed it to a feature model for benchmarking purposes [14]. They examined only the Kconfig files of one specific architecture (*x86*). Diettrich et al. suggested an approach for extracting implementation variability from the Linux build system, as it contains more than 65% of all config options [3]. In contrast, we examine *all* configuration contexts (source code, default configuration, makefiles, and Kconfig) of the whole system (all architectures and subsystems), with the goal of evaluating the impact of CPP on comprehensibility. Our study thus includes many observations not made previously by others.

III. Research Questions

Our ultimate goal is to gain some insights about the increased complexity of the code that results from configuration

variability management with CPP. Considering two of the studies cited above raises some important methodological questions. The work of Liebig et al. [9], [10] consists of a wide survey of 40 large-scale open-source projects, comprising 30 million lines of code. By necessity, such a study is based on automated tools and a high-level view of average metric values. Tartler et al. [19] focus on only one system, namely Linux, and show that this system suffers from bugs resulting from inconsistent use of configuration options. This leads us to the following specific questions:

1) Can the *relevant* configuration options indeed be identified by straightforward lexical means?
2) Are all config options equally important, and are average values generally representative?
3) Can the effect of config options be evaluated by syntactic analysis alone?
4) Do config options affect the whole codebase in a uniform manner?
5) Does conditional compilation based on config options introduce significant complexity to the code?

Thus we wish to take the reservations of Tartler et al. into account, and perform a more detailed study than that performed by Liebig et al. This is enabled by focusing on a single system, and considering complete distributions rather than averages.

IV. Characterization of the Linux Kernel Configuration

We are specifically interested in variability that is part of the system's design, namely explicit support for different configurations. Studying such variability in a system like Linux has two possible points of departure: either collect information on all the CPP constants used to control conditional compilation, or else start with all the known configuration options. As we show below, neither is completely satisfactory: there are many CPP constants that do not reflect real variability, and there are many config options that do not appear in the code. We therefore actually want the intersection of the two. But even this is not enough, as the code may include definitions of derived CPP constants that depend on other config options.

A. Source Configurability

As noted above, configurability is implemented in the source code using conditional compilation. To study its effect on the code we must first identify the CPP constants that are part of the configuration control mechanism. We therefore started by extracting all the CPP constants from all the #ifdef expressions[1] in the Linux source.

We found 10,988 different constants that appear in #ifdefs, of which 4731 start with CONFIG_ and may therefore represent config options (but some are false positives, as we show below). In this we use Linux-specific knowledge, and depart from Liebig et al. who considered all CPP constants. We claim this is important for reliable results concerning code variability, and thus that variability cannot be studied using only syntactic means. For example, 4330 of the non-config constants are defined by #define directives in the code itself, which means that they typically do not contribute to variability and always provide the same path.

[1]Here and in the sequel we use #ifdef to also mean #ifndef, #if, and #elif.

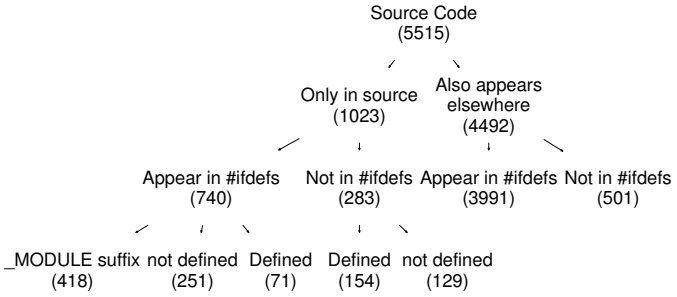


Fig. 1. Distribution of CONFIG_ options in the source code.

However, we found that some of these defined constants actually depend on configuration options in one of two ways: they are either defined as an alias of a config option, or else their very definition occurs within a code fragment that is only compiled conditioned on a config option. This means that their definition is *derived* from the config options. We found 23 thousand such derived definitions, but only 1009 of them are relevant because they are subsequently used in #ifdefs. This means that in total there are actually 5740 constants that may reflect configurability, and not 4731 as a merely syntactic analysis suggests.

Another large group of constants is due to the CPP inclusion mechanism: it is customary to avoid double-inclusion of header files by protecting such files with an #ifdef based on a constant defined in the same file. We ignore these constants and #ifdefs as they are idiomatic and do not reflect variability. The rest of the constants are not of real interest. Many of them deal with debug issues, while others are not defined at all (even not in makefiles).

B. Inconsistent Use of Config Options

As we mentioned above Linux configuration options may appear in four different contexts. The config options are initially born in the *Kconfig* files, the settings in the *defconfig* files are derived from the Kconfig files, and the options in *source code* files or *makefiles* are a subset of the total config options. In an ideal world all these sources of the config options would be synchronized. In practice, they are not.

We identified potential config options as follows. In the Kconfig files, they are introduced by the config keyword (see Listing 1). In the other files they are string constants that start with CONFIG_. When we checked all the header and implementation files (C and assembly) of the source code we found 5,515 such string constants (Figure 1). In the Kconfig system files we found 9,342 options, and in the defconfig files we found 8,696 options. Finally, we got 6,325 config options in makefiles. Altogether we have 11,303 unique config options from all these sources. These results and the logical relations between them are presented in Figure 2.

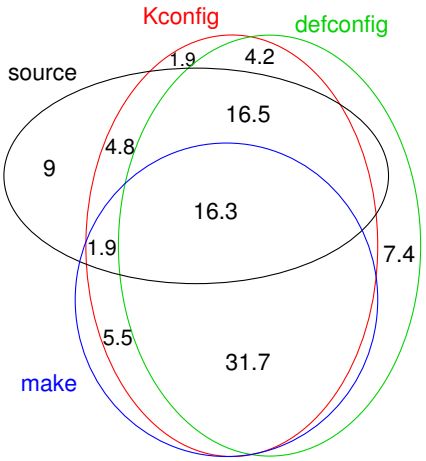


Fig. 2. The overlap relations between the different sources of the config options clearly indicate they are inconsistent. Values are percent of the total number; values less than 1% not shown.

Obviously, just less than half of the options we have found are used in the source code. This is not unreasonable because an additional 37.5% are used in makefiles to control which source files are included in the build process. Strangely, 6.1% of the options in the Kconfig system do not appear at all in the source or the makefiles. These options are therefore effectively no-ops, with no effect on the configuration. Stranger yet, 7.4% appear only in defconfig files. This contradicts the assumption that the defconfig files are derived from the Kconfig files, and may reflect legacy options in architectures that are not well-supported any more.

In the context of our interest in code quality, the biggest anomaly in Figure 2 is that around 9% of the total config options occur only in the source code and not in any of the configuration files as expected. Further examination of the source code helped to explain this and revealed several different sub-categories of options (left branch of Figure 1).

First, we found that more than 40% of these options have a _MODULE suffix. These are derived from options that were defined in the Kconfig files without this suffix. The suffix is appended during the build process, when creating the autoconf.h file, if the user configured an option to be compiled as a module. So, for a given option *X* in the Kconfig system, we may see both CONFIG_*X* and CONFIG_*X*_MODULE in the source code.

Second, we found that 22% of these options are defined by the #define directive in the source code, but not in the configuration process. In other words, these CPP constants are false positives that do not reflect true configurability (unless they are derived from true config options).

Next, around 25% of these options seem not to be defined at all, although they are in fact used by conditional compilation directives. We believe that most such options are leftovers from previous versions that should have been removed. A small number may be bugs, where a config option name was misspelled [19].

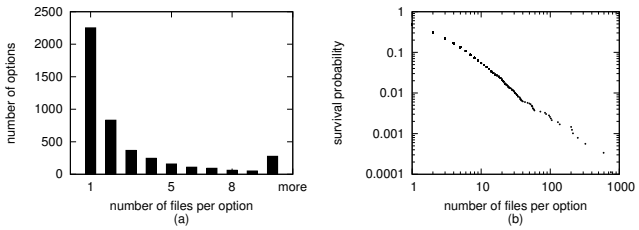


Fig. 3. a) Distribution of config option number of files. b) LLCD plot showing the distribution has a heavy tail.

Finally, most of the remaining CONFIG_ strings were defined and used at the C language level with no relation to the preprocessor at all. For example, elements of an enum data type might have names that start with CONFIG_. These have nothing to do with our research and we ignore them.

The conclusion from all the above is that the configuration process settings are not expressed cleanly in kernel code which is loaded with redundant options as well as misleading naming and usages. Such behavior soils the code and as a result makes the work of developers harder. On the basis of this conclusion we gain a significant insight: *A blind syntactic analysis of a large software system may miss its goal*. Some of the previous analyses of Linux may have had this problem. Our results corroborate and extend those of Tartler et al. [19] who specifically study the inconsistency of using config options.

Our goal is to eventually study the effect of conditional compilation directives based on config options on code comprehension. We will therefore focus on the real options that have a real influence on configurability. These options are those that occur in the source code #ifdefs as well as in the Kconfig files. To this set we will add the derived constants which were presented earlier. There are 4,440 options that are shared by Kconfig and the source code, out of which 3,941 are used in ifdefs. When adding to the 1,009 derived constants that also appear in #ifdefs we get 4,950 config options in total.

Note that config options with the _MODULE suffix are not included because each time they are used in the code their counterpart options, those without _MODULE, are also used in the same expression. Thus they do not add any independent configurability. Config options that are used only in makefiles are also not included, because they do not affect the difficulty of comprehending the code. From now on whenever we refer to config options we mean the set of real configuration options as it was defined here.

C. Heavy-Tailed Distribution of Config Options

Figure 3(a) shows how the configuration options distribute over the files of the source code. More than 2,700 options are used in one file only; these are the more specific options. On the other hand 285 options are used in more than 10 files; these are the cross-cutting options. This latter category of options may cause difficulties in maintenance because the developer must potentially think about many files when processing a single configuration option. In the most extreme case, we

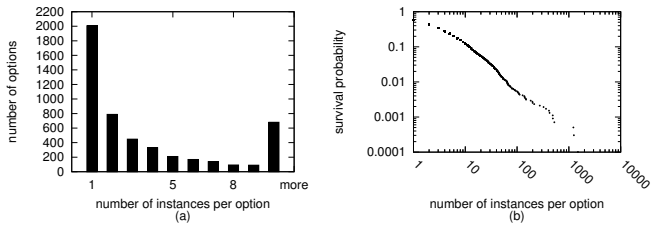


Fig. 4. a) Distribution of config option number of instances. b) LLCD plot showing the distribution has a heavy tail.

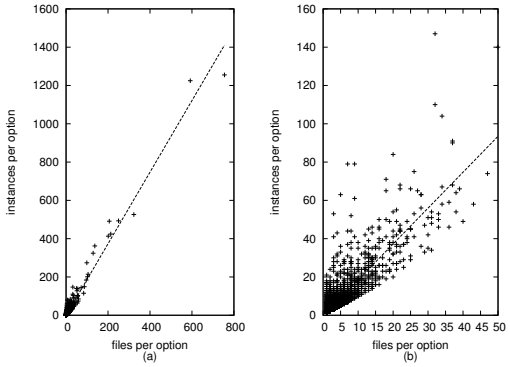


Fig. 5. a) Correlation between config options instances and the number of files they occur in. b) zoom in on the cluttered area.

Option	Files
PM	753
SMP	591
DEBUG	591
PROC_FS	322
PCI	250
COMPAT	213
64BIT	206
MMU	201
X86_64	137
X86_32	129
BITS_PER_LONG	129
NET_POLL_CONT.	105
SYSCTL	102

TABLE I
CONFIG OPTIONS THAT APPEARED IN THE MOST FILES.

Option	Instances
DEBUG	1498
PM	1246
SMP	1221
PROC_FS	525
PCI	488
64BIT	490
COMPAT	424
MMU	413
X86_64	362
X86_32	324
PPC64	274
BITS_PER_LONG	200
NET_POLL_CONT.	214

TABLE II
CONFIG OPTIONS THAT HAD THE MOST INSTANCES.

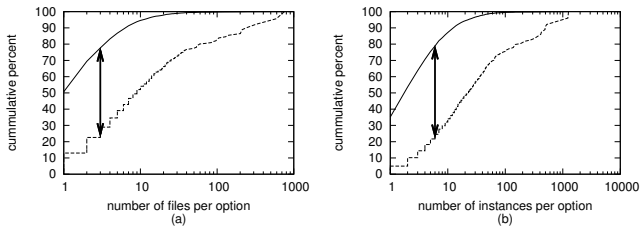


Fig. 6. The "count" (upper) and "mass" (lower) distributions of option occurrence. a) Mass-count disparity plot of number of options per file. b) Mass-count disparity plot of number of instances per option.

found that the most frequent option is used in 753 different files.

The histogram in Figure 3(a) indicates that the distribution of config options across files is highly skewed. We therefore checked whether this is a heavy-tailed distribution. To do so we use an LLCD plot, as shown in Figure 3(b). This shows that the survival function of the distribution is approximately linear in log-log axes, which means that it decays according to a power law and is indeed heavy-tailed. The tail index, which is given by the slope of this line, is approximately 1.3. In heavy-tailed distributions the tail index ranges between 0 and 2, and smaller tail indexes indicate a heavier tail.

Next we considered the number of instances of each config option (note that a config option can have several instances in the same file). The results are illustrated in Figure 4(a). As may be expected this also shows that we have many config options with a small number of instances and few config options with a large number of instances. And again, the distribution of the config options in terms of instances is found to be heavy-tailed as illustrated in Figure 4(b) using an LLCD plot.

To relate these two distributions, we found that the number of files and the number of instances of a config option are correlated with a correlation coefficient of 0.97. It is easy to see this in Figure 5. In addition, the same config options appear at the top in both cases, and almost in the same order. These highly-used config options are listed in Tables I and II. They can be classified into three main groups:

- Related to cross cutting operating system features, such as power management (PM) or SMP support.
- Related to a specific operating system feature, such as the /proc file system.
- Related to a specific architecture or device, such as Intel X86, IBM PPC, or the PCI bus.

Skewed distributions may also be characterized by their mass-count disparity [5]. In our context this means that a small number of configuration options represent the majority of the files and instances, while at the same time most of the config options together account for only a small fraction of the files and instances. Such a phenomenon, when it exists, helps to focus on those items that have the most impact.

In Figure 6 we show the disparity between the mass (files or instances) and count distributions. In both cases the joint ratio is close to the 20/80 rule, which means that around 20% of the configuration options are responsible (present in) for 80% of the different files and instances, while the other 80% of the options lead to only 20% of the files and instances.

The same results and insights were obtained when we investigated the concentration of config options in different files. We counted the number of different configuration options that appear in each file as well as the number of instances. The results show that 4,169 files contain only one option while 237 files contain at least 10 different options. When examining the instances of the options we get that 2,496 files contain only one instance, and 858 files contain at least 10 instances. In both cases we found that the distributions are heavy-tailed, with tail indices of 1.4 and 1.7 respectively. The files in the tail are hot-spots of configurability.

D. Syntactic vs. Semantic Analysis

Our work so far, and previous work as well (e.g. [9], [13]), was performed at the syntactic level. However, we

```
1  static netdev_tx_t eexp_xmit(struct sk_buff *buf, struct
       net_device *dev)
2  {
3    short length = buf->len;
4  #ifdef CONFIG_SMP
5    struct net_local *lp = netdev_priv(dev);
6    unsigned long flags;
7  #endif
8  #if NET_DEBUG > 6
9   printk(KERN_DEBUG "%s: eexp_xmit()\n", dev->name);
10 #endif
11   if (buf->len < ETH_ZLEN) {
12     if (skb_padto(buf, ETH_ZLEN))
13       return NETDEV_TX_OK;
14     length = ETH_ZLEN;
15  }
16  disable_irq(dev->irq);
17 #ifdef CONFIG_SMP
18       spin_lock_irqsave(&lp->lock, flags);
19 #endif
20 {
21    unsigned short *data = (unsigned short *)buf->data;
22    dev->stats.tx_bytes += length;
23    eexp_hw_tx_pio(dev,data,length);
24 }
25    dev_kfree_skb(buf);
26 #ifdef CONFIG_SMP
27    spin_unlock_irqrestore(&lp->lock, flags);
28 #endif
29    enable_irq(dev->irq);
30    return NETDEV_TX_OK;
31 }
```

Listing 3. The blocks of the CONFIG_SMP option are orthogonal to the rest of the code in the function (drivers/net/eexpress.c).

```
1  static int show_stat(struct kmem_cache *s, char *buf,
       enum stat_item si)
2  {
3   unsigned long sum = 0;
4   int cpu;
5   int len;
6   int *data=kmalloc(nr_cpu_ids*sizeof(int),GFP_KERNEL);
7   if (!data)
8               return -ENOMEM;
9   for_each_online_cpu(cpu) {
10   unsigned x = get_cpu_slab(s, cpu)->stat[si];
11   data[cpu] = x;
12                   sum += x;
13  }
14  len = sprintf(buf, "%lu", sum);
15 #ifdef CONFIG_SMP
16  for_each_online_cpu(cpu) {
17   if (data[cpu]&&len<PAGE_SIZE-20)
18    len += sprintf(buf + len, " C%d=%u", cpu, data[cpu]);
19  }
20 #endif
21  kfree(data);
22  return len + sprintf(buf + len, "\n");
23 }
```

Listing 4. The blocks of the CONFIG_SMP option are coupled with the rest code of the function (mm/slub.c).

claim that syntactic measures are not sufficient. By looking more closely at the conditional blocks, one may reveal semantic relationships between blocks that are governed by configuration options and other free blocks. Such interactions, may have a significant impact on the feasibility of applying aspects technology to reduce tangled code when handling cross-cutting concerns [7]. Indeed, Adams et al. [1] have shown that conditional compilation can be partially refactored into aspects.

To be more concrete we present code snippets to show that semantic involvement is required when analyzing code in the context of config options. We focus on a few config options and investigate them in different semantic contexts.

Orthogonality versus Coupling. We start with two basic cases: optional blocks that are orthogonal to the surrounding functionality and optional blocks that have a tight connection with the rest of the code, namely coupled code. The key point here is that the two behaviors occur for the same config option. This means that a single config option, which is considered a concern in AOP terminology, behaves differently regarding its connectivity to other concerns where it appears.

Listing 3 shows a function that has three blocks that are controlled by the CONFIG_SMP option. In line 20 the function acquires the lock and disables interrupts on the local processor while saving the state of the interrupts in flags. The corresponding unlock is called in line 30. Lines 5 and 6 define two variables which are needed only for the sake of these locking/unlocking operations. This is a classical case where aspects would be feasible especially with the knowledge that this function's aim is to transmit a packet and this concern is orthogonal to the CONFIG_SMP one. To gain such an insight one should delve into the code.

However, when examining Listing 4 the local variable len is defined and initialized outside the CONFIG_SMP block. Then, it is used twice and changed once within the CONFIG_SMP and finally used again outside this block. Moreover, the pointer buf is changed three times: before, within, and after the CONFIG_SMP block. One more point to mention is that buf is a parameter which is passed by address, so any change in its value will be returned back to the caller function. These tight dependencies between the function body, the config option block, and the caller function imply a semantic connection and logical dependency to the other parts of the function. Thus it is hard to extract out the block of the CONFIG_SMP automatically.

This function is even a bit more complex than was described. Line 12 calls a function with a body which is controlled by CONFIG_SMP. Also, lines 11 and 21 call the same macro where in line 11 the macro is free while in line 21 it is controlled by the #ifdef. This macro, in the file where it is defined, is controlled by CONFIG_SMP. This means that in fact lines 11–16 are controlled by the CONFIG_SMP option but this is not visible.

Do Many Instances Imply Coupling? The CONFIG_SMP option appears in 591 different files with 1,221 instances within those files. Seemingly, it is a hard case of a cross-cutting concern due to its wide scattering and the potential

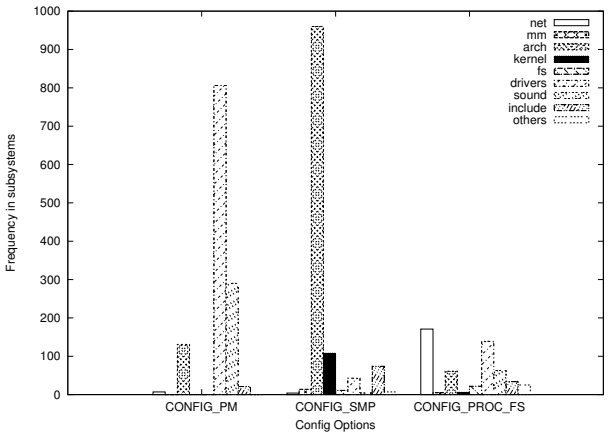


Fig. 7. Distribution of top config options over different subsystems.

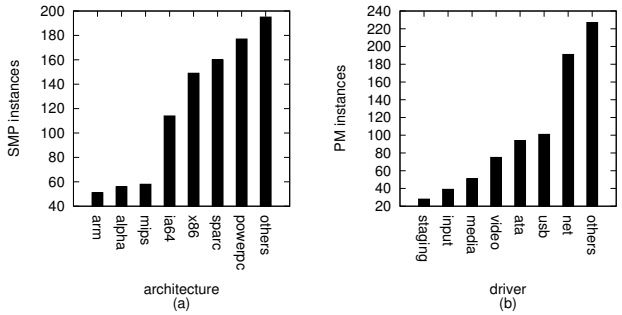


Fig. 8. a) Distribution of the CONFIG_SMP over architectures. b) Distribution of the CONFIG_PM over drivers.

coupling that this may imply. We argue that it is not as bad as it sounds.

The Linux Kernel is composed of subsystems residing in distinct subdirectories. These subsystems include, among others, the initialization code (init), memory management (mm), file systems (fs), and support for diverse architectures and devices (arch and drivers). Each subsystem (directory) is, to a large extent, an independent module. This means that despite the fact that the same config option may appear in multiple subsystems, there is not much coupling between the subsystems and one can treat each subsystem separately.

Other config options are logically related to specific subsystems, so they are largely localized in those subsystems. Thus even if they appear in other subsystems, they create only slight coupling with them. To demonstrate this, we looked at 3 of the most frequent options: CONFIG_PM, CONFIG_SMP, and CONFIG_PROC_FS. Figure 7 shows how these options distribute over the different subsystems of the Linux Kernel. CONFIG_PM occurs significantly in the drivers and sound modules while in the other modules it does not. Similarly, almost all the instances of CONFIG_SMP occur in the arch directory. When looking at the CONFIG_PROC_FS, about 60% of its instances occur in the net and drivers subsystems.

Actually, these results are not surprising. The designation of the *SMP* option is to enable *symmetric multi-processing support*. So, one expects a heavy use of this option in the directory where all the architectures are defined. In the case of CONFIG_PM most instances reside in the drivers directory due to the fact that power management is accomplished on the system components and peripherals. These examples again demonstrate the fact that syntactic analysis should be complemented by its semantic counterpart.

Moreover, Figure 8(a) shows how CONFIG_SMP distributes over the different architectures, and Figure 8(b) shows how CONFIG_PM distributes over the different drivers. Ultimately, one of these orthogonal architectures as well as a combination of the appropriate orthogonal drivers are built for each variant of the system. This means that the real number of instances of a specific config option that a developer needs to tackle at the same time is not the total number, but only those related to the specific architecture or driver he works on.

Due to the modularity of the Linux system and these insights we argue that the coupling between the different entities in the context of each config option is minimal.

V. Code Metrics

To get insights about the complexity resulting from CPP and examine the feasibility of alternative techniques we studied various code metrics. Here we focus on the dominant subset of the configuration options. These are the most frequent config options, which constitute approximately 20% of the different config options and account for approximately 80% of the instances.

Metric measurements with a normal distribution are well described by the arithmetic mean and the standard deviation. But when metrics have a skewed distribution, commonly with a low mean and a large variance, it is more appropriate to use the geometric mean and the multiplicative standard deviation [11]. The geometric mean, denoted by $\bar{X}^*$, is defined as the nth root of the product of n positive numbers: $\bar{X}^* = e^{(\frac{1}{n}\sum_{i=1}^{n} logX_i)}$. The formula is presented this way to allow computation and avoid overflows when the product is large. In contrast to the arithmetic mean, which is dominated by the large values, the geometric mean responds equally to changes in large and small values. The multiplicative standard deviation, denoted by $S(X)^*$, is essentially the average of the quotients of the samples and the mean: $S(X)^* = e^{\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(\log \frac{X_i}{\bar{X}^*})^2}}$. On the basis of these two descriptive values, the distribution is characterized by the range $\bar{X}^* \ ^{\times}\!/ \ S(X)^*$ (that is, from $\bar{X}^*/S(X)^*$ to $\bar{X}^* \times S(X)^*$).

In using full distributions and noting their skewed nature we again depart from previous practices. Looking at the data of [9] we see that in practically all cases the standard deviation is (much) larger than the mean. This indicates that the distributions are skewed, and that the arithmetic mean and standard deviation do not provide a good summary of the data.

The metrics we checked are the following.

Scattering Degree of Config Options. The scattering degree metric, denoted by SD, was defined by [9] as the number of instances of CPP constants in different expressions. This

metric measures the spread of configuration options, and high values are expected to reflect difficulty in comprehending the code and managing the configurations. In practice they actually used the average number of instances per config option. We have shown the distribution in Figure 4. Focusing on the dominant options, we found that the geometric mean of the scattering degree is 15.2, with multiplicative standard deviation of 3. This means that the bulk of the scattering degrees of the dominant config options are covered by the interval 5 to 45.6.

Tangling Degree of Config Options. The tangling degree metric, denoted by TD, was defined by [9] as the number of different CPP constants that occur in an expression. This metric measures the mixing of different configuration options within the expressions of #ifdefs. To evaluate this metric we measure the occurrences of the dominant configuration options in the CPP expressions. We found that the geometric mean is 1.035 with deviation of 1.185. This means that usually we have only one config option and the expressions are quite simple.

Conditional Blocks. The code blocks which are controlled by #ifdefs are classified in the literature as homogeneous or heterogeneous. This is important because alternatives such as aspects are only applicable to homogeneous blocks. To compare the blocks in different #ifdefs we squeeze whitespaces and concatenate the lines to one string and then make the comparison. We found that 92% of the conditional blocks of the config options of the dominant subset are heterogeneous. In particular, the options CONFIG_PM and CONFIG_SMP each have more than one thousand heterogeneous blocks.

As such comparisons are very stringent we also made some manual checks. We manually looked at one hundred blocks of the CONFIG_PM option and got the same results. Moreover, we expected that the automatic comparison will have problems with long code blocks. Our manual work revealed that in fact the automatic comparisons identified identical blocks of 10 lines. This probably indicates that a copy-paste mechanism was used by the developers wherever the same functionality was needed. But we cannot infer that this is the general case and more work should be done here.

Complexity of Expressions. We found 68,524 lines of conditional statements in CPP directives. More than 36% of them reference at least one configuration option of the dominant set. We examined these expressions and counted the number of logical operators in each of them. The results are presented in Figure 9(b). We found that about 79.2% use #ifdef or #ifndef, and therefore do not really have any expression. About 18.3% use the #if construct, and the #elif construct captures the remaining 2.3%.

In the #if and #elif expressions, we found that about 53.3% were compound with an average of 1.2 logical operators. The or operator dominates with more than 72% of the operator instances, the and operators captures about 25%, and the not operator is almost non-existent. However, it should be noted that about half of the or occurrences have operands with a _MODULE suffix. This observation shows that the use of or partially follows a simple predefined pattern that is supposed to reduce its complexity.

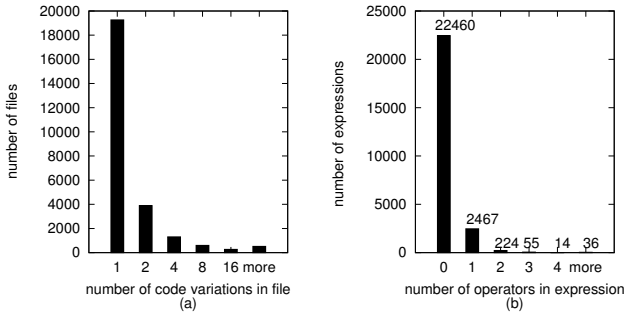


Fig. 9. a) Distribution of code variation in source files. b) Complexity of the expressions in #ifs.

Code Variations. When looking at the code base, the developer must consider all the different combinations in which the code may be built in the future. These combinations are concatenations of code segments that are created on the basis of the evaluations of the preprocessor conditionals. It is obvious that the number of the potential variations has a significant impact on the code understanding. To get insights about the code understanding complexity we counted the number of code variations for each of the source code files. As a first step we only took into account basic conditionals (#ifdef, #ifndef) that contain one configuration option. We have noted earlier that #ifdef and #ifndef constitute more than 79% of the conditionals of interest, so they may be expected to reflect the full picture. The results are shown in Figure 9(a). While obviously most files have few variations, some have an extremely large number of variations, with a maximal value of 316,659,348,799,488.

Nesting. One more aspect to look at is the nesting of conditionals. Initially we checked the nesting of general conditionals that are not related only to configuration options. Figure 10(a) describes the results. The average depth of conditional statements is 1.08. Afterward, we measured the depth of nesting for conditionals which contain at least one configuration option. The results are presented in Figure 10(b). The average depth of conditional statement in the configuration context increased slightly to 1.10, which is still low.

The metrics presented here provide evidence for the existence of many config options scattered around the code. These are bad news due to potential coupling that such characteristics may impose on the code. The good news are that these scattered config options are used in a simple manner. This stems from the low value of the tangling metric and the simple structure (low number of logical operators) of the expressions of the conditional constructs. Moreover, the intensive use of the simple form of the conditional construct as opposed to its composite form is additional evidence for simplicity.

VI. Discussion and Conclusions

The CPP has been identified over the years as a potentially harmful tool, in particular when it is used to implement variability in a large scale system like Linux. In this study we performed a detailed characterization of the Linux kernel

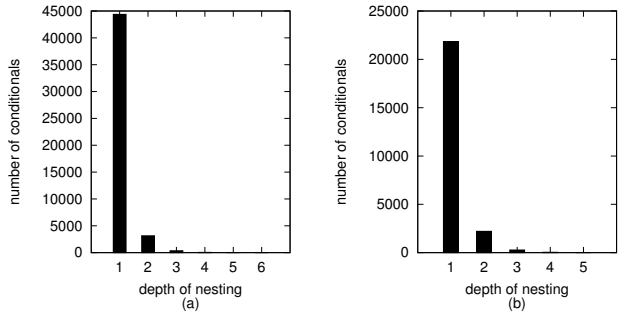


Fig. 10. Depth of the nesting of conditional constructs. a) General Conditionals. b) Conditionals with at least one config option. .

configuration as done using CPP, and presented different metrics in order to better understand CPP's usage and effect.

The approach taken in most previous work was breadth-first, and provided a survey of CPP usage in many systems. Such an approach necessarily comes at the expense of deeper analysis of each system. We therefore complemented this by using a depth-first approach on one particular system, namely the Linux kernel. This allowed us to employ domain-specific knowledge and semantic analysis, and led to various insights that would not be possible in a more general study.

In order to study the variability built into the system, we focus on Linux's configuration options. Our first finding is that it is not trivial to identify the effect of config options on the code, and that their usage is inconsistent. For example, we found more than one thousand options that appear only in the source code, and more than a thousand CPP constants that are derived from config options even though they are not themselves direct config options.

We found that the distribution of the config options is skewed (even heavy-tailed), and manifests the mass-count disparity phenomenon. These realizations are useful because they indicate the existence of a relatively small but prominent group of cross-cutting options. This helped us to focus on this subset when assessing their impact on the code. The skewness of the distribution also indicates that one should be careful to use the right descriptive metrics: geometric mean and multiplicative standard deviation.

Overall we found nearly 5000 real config options. This is worrying because it suggests extensive use of the CPP across the code, with adverse consequences for code comprehension. However, a large portion of these options have only a few instances, so their effect is actually very localized. The small fraction of the options that have many instances appear to create only slight coupling, due to the modularity of the system. The logical expressions used to control conditional compilation are typically very simple, and there is very little nesting. All these findings indicate that variability management with CPP does not cause excessive code degradation. However, the blocks that are controlled by these options were classified as heterogeneous which is bad due to the difficulty of using alternative techniques. Moreover, the code is soiled with garbage and misleading config options so an initiation of a clean up process is vital.

Our work suffers from several threats to validity. The complexity imposed by CPP usage can be measured using additional metrics, such as McCabe's cyclomatic complexity. Another issue is whether to focus on the most prominent config options, or to always consider all of them. Finally, we do not know whether these characterizations are true for other systems except Linux. Nevertheless, we think that this case study is important in its own right even if specific findings do not generalize. We think that it is valuable to characterize more systems in the domain of operating systems and then move to other domains. It is also interesting to study the evolution of CPP usage across the many versions of the Linux kernel itself.

References

[1] B. Adams, H. Tromp, W. D. Meuter, and A. E. Hassan., "*Can we refactor conditional compilation into aspects?*" In 8th *Intl. Conf. Aspect-Oriented Softw. Dev.*, pp. 243–254, 2009.

[2] J.-M. de Goyeneche and E. A. F. de Sousa, "*Loadable kernel modules*". *IEEE Softw.* **16(1)**, pp. 65–71, Jan/Feb 1999.

[3] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, "*A robust approach for variability extraction from the Linux build system*". In 16th *Proc. Intl. Software Product Line Conf.*, 2012.

[4] J.-M. Favre, "*The CPP paradox*". In 9th *European Workshop on Softw. Maintenance*, 1995.

[5] D. G. Feitelson, "*Metrics for mass-count disparity*". In 14th *Modeling, Anal. & Simulation of Comput. & Telecomm. Syst.*, pp. 61–68, Sep 2006.

[6] "*The GNU project emacs homepage*". www.gnu.org/software/emacs/emacs.html.

[7] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "*Aspect-oriented programming*". In 11th *European Conf. Object-Oriented Prog.*, pp. 220–242, 1997.

[8] M. Krone and G. Snelting, "*On the inference of configuration structures from source code*". In 16th *Intl. Conf. Softw. Eng.*, pp. 49–57, 1994.

[9] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "*An analysis of the variability in forty processor-based software product lines*". In 32nd *Intl. Conf. Softw. Eng.*, pp. 105–114, 2010.

[10] J. Liebig, C. Kästner, and S. Apel, "*Analyzing the discipline of preprocessor annotations in 30 million lines of C code*". In *Intl. Conf. Aspect-Oriented Softw. Dev.*, pp. 191–202, Mar 2011.

[11] E. Limpert, W. A. Stahel, and M. Abbt, "*Log-normal distributions across the sciences: Keys and clues*". *BioScience* **51(5)**, pp. 341–352, May 2001.

[12] B. McCloskey and E. Brewer, "*ASTEC: A new approach to refactoring C*". *SIGSOFT Software Engineering Notes* 2005.

[13] A. Reynolds, M. E. Fiuczynski, and R. Grimm, "*On the feasibility of an AOSD approach to Linux kernel e xtensions*". In *Proc AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Softw.*, 2008.

[14] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "*The variability model of the Linux kernel*". In 4th *Workshop on Variability Modeling of Software-Intensive Systems*, 2010.

[15] N. Singh, C. Gibbs, and Y. Coady, "*C-CLR: A tool for navigating highly configurable system software*". In 6th *Workshop on Aspects, Components, and Patterns for Infrastructure Softw.*, 2008.

[16] H. Spencer and G. Collyer, "*#ifdef considered harmful, or portability experience with C news*". In *Proc. USENIX Technical Conf.*, 1992.

[17] R. M. Stallman and Z. Weinberg, "*The C preprocessor*". GNU project, Free software foundation, 2010.

[18] A. Sutton and J. Maletic., "*How we manage portability and configuration with the C preprocessor*". In *Intl. Conf. Softw. Maintenance*, pp. 275–284, 2007.

[19] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "*Feature consistency in compile-time–configurable system software: Facing the Linux 10,000 feature problem*". In 6th *EuroSys*, pp. 47–60, Apr 2011.

[20] L. Vidács and A. Beszédes, "*Opening up the C/C++ preprocessor black box*". In 8th *Symp. prog. lang. & softw. tools*, 2003.

Criticality of Defects in Cyclic Dependent Components

Tosin Daniel Oyetoyan[1], Reidar Conradi[1]
[1]Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway
{tosindo, conradi, dcruzes}@idi.ntnu.no

Daniela Soares Cruzes[1,2]
[2]SINTEF
Trondheim, Norway
danielac@sintef.no

***Abstract*—(*Background*) Software defects that most likely will turn into system and/or business failures are termed critical by most stakeholders. Thus, having some warnings of the most probable location of such critical defects in a software system is crucial. Software complexity (e.g. coupling) has long been established to be associated with the number of defects. However, what is really challenging is not in the number but identifying the most severe defects that impact reliability. (*Research Goal*) Do cyclic related components account for a clear majority of the critical defects in software systems? (*Approach*) We have empirically evaluated two non-trivial systems. One commercial Smart Grid system developed with C# and an open source messaging and integrated pattern server developed with Java. By using cycle metrics, we mined the components into cyclic-related and non-cyclic related groups. Lastly, we evaluated the statistical significance of critical defects and severe defect-prone components (SDCs) in both groups. (*Results*) In these two systems, results demonstrated convincingly, that components in cyclic relationships account for a significant and the most critical defects and SDCs. (*Discussion and Conclusion*) We further identified a segment of a system with cyclic complexity that consist almost all of the critical defects and SDCs that impact on system's reliability. Such critical defects and the affected components should be focused for increased testing and refactoring possibilities.**

Keywords—defect severity; dependency cycles; defect distribution; defect-prone components; software reliability; empirical study

I. INTRODUCTION

According to [1], software reliability is the probability that software will not cause a system to fail (i.e. behave incorrectly) for a specified time under specified conditions. A system failure may be the result of a software fault/defect [1]. Moreover as noted by [2], software does not "wear out" after some period of proper operation as hardware components do. In addition, defects in software systems may not be apparent over time but when they are exposed, they act like a hidden bomb [2].

There are many cases of system failures due to software defects. For example [2]: The "STS-126 Shuttle Software Anomaly-2008"; The "Air Traffic Control Communication Loss – Los Angeles 2004"; The "Widespread Power Outage in the Northeast in Northern Ohio – 2003"; the "Ariane 5 Failure Forty Seconds After Lift-Off – 1996". In all these cases, the failures were caused by defects that we could classify to be of critical severity because of their impact on these systems. Critical defects are not limited to system and/or hardware failures. They may also be associated with many business failures. Many examples exist, for instance [3]; recently, a "Faster Payment System" at Lloyds bank, meant to speed up payment, was hit by critical defects and ironically delayed wage and bill payments and caused duplicate charges for PayPal users. Similarly, a trading software glitch was caused by critical defects that resulted in a $461.1million loss for Knight Capital last year [3].

Many of today's software systems are overly complex and indeed highly interconnected. The higher the complexity of a system, the more difficult it is to maintain and the higher the risk of accidental and unexpected failures [4]. One area of such software complexity is dependency cycles that are formed by direct or indirect decisions during software development and evolution. Dependency cycles among components are notorious for extremely increasing coupling complexity among interconnected components [5, 6]. Despite numerous claims that cycles inhibit software quality attributes such as extensibility, understandability, testability, reusability, build-ability, maintainability and reliability [7-9], evidence shows that they are widespread in real life software systems [9-13]. Intuitively, we expect that since cycles increase coupling complexities among components [5, 6], then it should have a positive correlation with the most defects. In fact, by performing an empirical study on six software systems, we confirmed this conjecture of higher defect profiles for cyclic-related components in all the six systems [14].

On the other hand, the study by Adams [15] showed that removing large number of defects may have a trivial effect on reliability. As pointed out in [15, 16], the most number of latent defects lead to very rare failure in practice while the vast majority of observed failures are caused by a relatively tiny number of defects [17, 18]. Therefore showing that it is not the number of defects, rather their severity that matters. A critical severity defect usually points to a fatal error that results into system, hardware and/or business failures whereas low severity defects mostly points to some cosmetic issues.

Since cycles reduce our cognitive ability to reason about interconnected components, we conjecture that the most severe defects detected in such systems may have an "undisclosed" relationship with the cyclic complexity. Our hypothesis in this study therefore, is that, components in cyclic relationships have higher likelihood of containing significant number of highest severity defects and severe defective components than those not in any cyclic relationships.

The remainder of the work is structured as follows: In section II, we lay the background for our study. In section III, we detail our empirical setup. Section IV presents the

978-1-4673-5739-5/13/$31.00 ©2013 IEEE

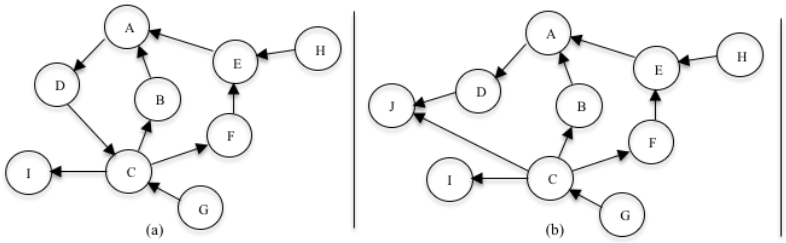


Fig. 1. (a - b) - Cyclic Dependencies and propagation effect on components in a software network [14]

results of this work while section V discusses our findings. In section VI, we draw out the threats to the validity of the results. Lastly, in section VII, we give the conclusion with a note on future studies.

II. BACKGROUND

In a software system, a component *X* is said to have dependency on another component *Y* if *X* requires *Y* to compile or function correctly [19]. For formal representation of a dependency graph for an object-oriented (OO) program, we borrow two definitions from [12] and state these as follows:

Definition 1. An edge labeled digraph G = (V, L, E) is a directed graph, where V = $\{V_1, ..., V_n\}$ is a finite set of nodes, L = $\{L_1, ..., L_k\}$ is a finite set of labels, and $E \subseteq V \times V \times L$ is the set of labeled edges.

Definition 2. The object relation diagram (ORD) for an OO program P is an edge-labeled directed graph (digraph) ORD = (V, L, E), where V is the set of nodes representing the object classes in P, L = {*I*, *Ag*, *As*} is the set of edge labels representing the relationships (Inheritance, Aggregation and Association) between the classes and E = $E_I \cup E_{Ag} \cup E_{AS}$ is the set of edges.

We concern ourselves in this study with dependency at the physical level, that is, both files (top-level classes) and packages since we can infer strong dependencies from physical relationship [8]. As illustrated in Figure 1a, cyclic dependency is formed when components depend on one another in a circular manner. This relationship covers both direct and indirect connection between components. Cyclic relationships increase coupling complexities and thus have the potential to propagate defects in a network [20].

A hypothetical case as depicted in Fig. 1 (a-b) demonstrates such effect. From Fig. 1(a), assume that component **I** contains some defects. We can further assume that the rest of the components **A – H** will have a certain probability to inherit the defect from **I**, since they are directly and indirectly dependent on **I**. To reduce the likelihood of defect propagation e.g. in Fig. 1(b), let us say that, a new component **J** is created so that components **D** and **C** depend on **J** directly thereby breaking the cyclic effect. By performing such a refactoring, the effect of possible defect propagation is reduced to only component **G**.

For the purpose of this paper, we use some of the terms defined in [14]: Assume a component c ∈ System P then:

Component's Children: Components that are directly and transitively dependent on c. E.g. in Fig. 1(a), All the components except component **I** are directly or transitively dependent on **A**. Components **G**, and **H** have no children. We use **TChildren** for both direct and transitive children and **DChildren** for direct children. For example, **DChildren(A)** = {B, E}.

Component's Parent: All components that c is both directly and transitively dependent upon. We use **TParent** for both direct and transitive parents and **DParent** for direct parents. For instance, **TParent(G)** = {A, B, C, D, E, F, I} and **DParent(G)** = {C}.

Component In-Cycle: Component c is said to be in cycle, if it has at least one parent that is the same as one of its children. E.g. **B** is in cycle because its parent **A** is also one of its children.

Component Depend on In-Cycle Component: Component c is said to depend on another in-cycle component if at least one of its direct parents is in cycle. **G** and **H** are examples of components that depend on **In-Cycle** components **C** and **E** respectively.

Associated Defect: Two components have associated defect if a specific defect affect both components. We use defect ID to track associated defect between components.

Some metrics such as CBO, RFC [21], CyclicClassCoupling[1] and dwReach [22, 23] are of interest but not useful for our purpose since they cannot classify whether a component is involved in cyclic components indirectly. Therefore, we describe the cyclic metrics and notations [14] relevant for our study. Consider a set of components, C in an object-oriented system. For each component c ∈ C:

- **Component In-Cycle:**
  inCycle: boolean
  $\exists p : p \subseteq (TParent(c) \wedge TChildren(c))$
  $\{\forall c. inCycle(c) \leftrightarrow p \neq \emptyset\}$
  where:
  $inCycle(c)$ *denotes* **c** *to be in a cyclic dependency*
- **Depend On Cycle:**
  depOnCycle: boolean
  $\exists x : x \in DParent(c)$
  $\forall c. depOnCycle(c) \leftrightarrow (\neg inCycle(c) \wedge inCycle(x))$
  where:
  $depOnCycle(c)$ *denotes c depends on inCycle component x that is a direct parent of c*

Cycles among components have been claimed to be detrimental to understandability [7], production [8, 24], marketing [8], development [8, 24], usability [8, 24], testability [8], integration testing [10-12, 19, 25], reusability [24], extensibility [9] and reliability [8].

[1] This metric counts the number of direct cyclic connections between two classes. For instance, C_1 depends on C_2 and C_2 depends on C_1

Although, it has been stated and implied, to date, it appears that only one study [13] has performed an elaborate empirical study of cycles on many software systems at the class level. The result shows that almost all the 78 Java applications they analyzed contain large and complex cyclic structures among their classes.

In a recent study [14], we established that components in cyclic relationships, either directly or indirectly, have significantly more defect-prone components than those not in any cyclic relationships. The four hypotheses we tested on six different and non-trivial systems confirm that:

1. Components in cycles have higher likelihood of defect-proneness than those not in cyclic relationships.
2. The higher number of defective components is concentrated in cyclic dependent components.
3. Defective components in cyclic relationships account for the clear majority of defects in the systems investigated.
4. The defect density of cyclic related components is sometimes higher than those in non-cyclic relationships.

However, as found in [15, 16], this is not sufficient to focus testing resources. We are compelled to find out if this majority of defects and defect-prone components are also the majority in both critical defects and severe defective components.

Zhou and Leung [26], Shatnawi and Li [27] and Singh et al. [28] demonstrated that object-oriented design metrics could predict defect-proneness of classes based on defect severity. Bhattacharya et al. [29] on the other hand, revealed that graph based metrics are capable of predicting defect severity, maintenance effort and number of defects at both the function and module levels. In similar direction like these studies but not concerned with prediction of defect-proneness of components, Menzies and Marcus [30], Lamkanfi et al. [31], Iliev et al. [32] and Yang et al. [33] have all focused efforts on models that could assign severity levels to defect reports.

These studies focused on (a) predicting defect-prone components based on their severity of defects using both the OO design and graph metrics and (b) predicting the severity of reported defects and not the affected components. Our work differs from these efforts in the sense that: None of these studies analyzed defect severity using cyclic complexity. In our study we identified cyclic dependent components as a set within software components that consists the majority of critical defects and defect-prone components with such critical defects.

This study extends our previous study [14] and the findings are aimed to add significance to the need to collect cycle metrics and focus on defect-prone cyclic related components with critical defects for increased testing activities and refactoring possibilities.

III. EMPIRICAL SETUP

Our goal in this work is to determine the severity of defects in the cyclic dependent components of the systems under study. As explained in [7, 8, 24], cyclic dependencies are better studied at physical design levels such as the source file (compilation unit) and package levels (Organizational and deployable units), since this type of dependencies is stronger than logical dependencies [8]. In addition, previous empirical studies [13] on cyclic dependency have performed analysis at the file levels. Furthermore, when developers resolve defects, they usually log the changes at the file level and thus have file to defect mapping. Based on the above reasons, we identify relationships and dependencies at the compilation units (top-level classes for Java) and at the package level.

We perform our evaluation in two ways: First, we use the set of metrics built around cyclic dependency relationships proposed in our previous study [14] to mine software components and classify them into two groups, "Cyclic" and "Non-Cyclic". Second, we statistically evaluate the severity of defects from cyclic related components and non-cyclic related components.

A. Systems under study

We choose two systems primarily because of their criticality to the environments where they operate. First, we analyze an industrial Smart Grid application, a type of system of systems (SoS) applications. Our motivation for the choice of this case study is that, as a critical infrastructure, the availability and reliability of the Smart Grid is crucial to its safety and security. Smart Grid represents the injection of Information and Communication Technology (ICT) infrastructure to the electricity grid to allow for bi-directional flow of energy and information [34].

The system under study is a distribution management system designed to monitor and plan the Grid operations. It provides real-time operational support by continuously receiving status data from the power grid. The system has been in development for about six years and we have analyzed six post releases (field and operational) of this application. It is mostly developed with C# programming language with .NET framework. As listed in Table I, it has a size of approximately 341KLOC and contains 1203 class files and 2142 classes as of version 4.2.4.

Furthermore, we choose Apache-ActiveMQ[2], a very powerful and open source messaging and enterprise integration pattern server. Our motivation for this choice is that systems that provide integration platform for other applications are very critical and form the backbone for these applications. The security and reliability of the applications running on this platform depend mostly on it. We consider this system to be non-trivial. As listed in Table I, the CORE module we analyzed as at release 5.7.0 (penultimate release) has about 136.22KLOC, containing 1517 class files and 1665 classes.

B. Research Hypotheses

The hypotheses investigated in this study are as follows:

- *H_A: Cyclic dependent components have significantly higher number of highest severity defects than non-cyclic components.*
- *H_B: Cyclic dependent components have significantly higher severe defect-prone components than non-cyclic components.*

A ***severe defect-prone component*** (SDC) is defined as a component within the top 25% with the highest severity defects.

To test our hypotheses, either a t-test or non-parametric

[2] http://activemq.apache.org/

TABLE I. SOFTWARE SOURCE CODE AND DEFECT DATA

Release	Date	#Pkg	#Class-File	#Class	KLOC	#Defective Class-File	#Defects
Apache-ActiveMQ							
5.7.0	Nov 22 2012	82	1517	1665	136.22	35	68
5.6.0	Jun 15 2012	83	1505	1649	133.25	88	102
5.5.1	Oct 16 2011	78	1331	1472	118.27	54	76
5.5.0	Apr 01 2011	78	1331	1472	118.27	115	105
5.4.2	Dec 02 2010	77	1258	1393	113.01	80	66
5.4.1	Sept 21 2010	77	1256	1386	112.20	79	63
CommApp							
4.2.4	Nov 14 2012	191	1203	2142	341.83	29	14
4.2.2	Oct 12 2012	191	1199	2134	339.78	49	18
4.1	Aug 17 2012	171	1002	1884	316.22	60	42
4.0.1SP4	Apr 11 2012	141	904	1650	286.99	69	29
4.0.1SP2	Mar 26 2012	142	903	1645	285.89	46	28
4.0	Oct 14 2011	133	849	1546	266.11	137	143

TABLE II. % OF DEFECTS MAPPED FROM DTS TO SVN

Apache-ActiveMQ		CommApp	
Release	%Bugs	Release	%Bugs
5.7.0	85.3	**4.2.4**	71.4
5.6.0	93.1	**4.2.2**	83.3
5.5.1	77.6	**4.1**	85.7
5.5.0	82.9	**4.0.1SP4**	69.0
5.4.2	80.3	**4.0.1SP2**	64.0
5.4.1	85.7	**4.0**	51.7

test [4] such as Wilcoxon signed rank test will be applicable depending on whether our data sample is normally distributed or not. Lastly, we test the difference in mean between both groups for significant difference that is greater than zero. Three categories are identified for both groups based on our hypotheses:

1. Number of critical severity defects recorded in each group
2. Number of defect-prone components with critical severity defects.
3. Number of severe defect-prone components in each group.

For these three categories, we test the hypothesis (1-tailed significance test):

- H_0: $\mu_C \leq \mu_{NC}$ (The mean of cyclic group is significantly less than or equal to the non-cyclic group)
- H_1: $\mu_C > \mu_{NC}$ (The mean of cyclic group is significantly higher than the non-cyclic group)

C. Data collection

We have collected data for six releases of two important applications. We describe in each subsection the details of our approach: (1) to collect the data from the defect repository, (2) to map the class files to the defects, (3) to aggregate the defect counts at the class-file and the package level, (4) of ranking components by severity of defect and (5) of dependency data collection

1) Defects collection from the defect tracking system (DTS)

We have collected defect data from both the HP-QC DTS and JIRA DTS. A Defect repository gives typically a high level overview of a problem report. For example, typical attributes of the HP-QC defect tracking system (QC-DTS) are the Defect ID, severity of the defect, the type of defect, date defect is detected, the module containing the defect, the version where defect is detected, and the date the defect is fixed. Our first step is to determine the defects that affect each version of the system. In the HP-QC, we use "Detected in Version(s)" and in Apache JIRA DTS, we use the "Affects version" field to filter all bugs that affect a particular version of the system. A certain defect may keep re-occurring and span several versions of a system (persistent defects [35]). We include such defects in all the versions they affect. Next, we filtered out "duplicate", "Not a problem", "Invalid", "Enhancement" and "Task" cases from the resolution/Defect Type field.

2) Method to map class files to defects

Version repository, on the other hand, is a configuration management system used by the developers to manage source code versions. The version system provides historical data about the actual class file that is changed and/or added as a result of corrective action (defect fixes), adaptive, preventive and perfective actions [36]. Thus, the SVN/CVS provides a detailed granularity level to know which source file(s) in the module(s) are changed to fix a reported defect. A common way to figure out what operation is performed on the source file is to look at the message field of the SVN commit. When developers provide this information with the bug number and/or useful keywords (e.g. bug or fix), it is possible to map the reported defect with the actual source file(s) [37, 38]. In some cases, not all bug commits in the version repository contain the bug number or useful keyword in the message field. In the past, researchers have approached this situation by mapping from defect repository to the version repository [38, 39].

We have used both approaches to map defect from the HP-QC and JIRA DTSs to the code changes. The resolution date allows us to map some of the untagged commits in the version system to the resolved bugs. Overall, for the six releases of each system, we mapped an average of 84.2% for Apache-ActiveMQ and 71% for CommApp (see Table II). From these percentages, we consider only defects that are associated with source files of the analyzed modules and ignore defects for non-source files, test source files and source files of other modules not analyzed. Consequently, the reported defect figures in the results section are fractions of the mapped percentages.

3) Aggregating number of defects per class file and per package

In a release, it is possible that multiple reported bugs be associated to one class file. The unique defect ID is thus appropriate to compute the number of defects fixes that

affect a class file and a package. From the mapped change data, we look up each file and determine the total of defects per file by counting the number of unique defect ID in this release. At the package level, we aggregate the unique defect IDs for each class file in the package. As demonstrated in Fig. 2, based on the defect ID, File1, File2 and File3 have 2 defects each and Pkg 1 has a total of 3 defects although it contains 3 files with 2 defects each. The unique defect-ID shows that for pkg1, only 3 defects are fixed.

4) Ranking of components by defect severity

The HP-QC DTS of CommApp uses four values to describe the severity of each recorded defect while in JIRA five values are used. The severity is determined based on the impact of the defect on the system and the business. From our observation of the message logs in both DTSs, defect severities with "blocker" and/or "critical" values relate strongly to reliability, performance and/or security issues in the system. For instance, in CommApp, an example is: "*Database running at 100% CPU*". An example in ActiveMQ is: "*Network bridges can deadlock when memory limit exceeded*". For HP-QC, a defect can be ***critical***, ***major***, ***average*** or ***minor***. In JIRA, a defect can be ***blocker***, ***critical***, ***major***, ***minor*** or ***trivial***. Because of the few numbers of ***blockers*** and ***critical*** defects in the defect data for Apache-ActiveMQ, we decided to merge these two to form only ***critical*** category. We transform the severity scales in both defect-tracking systems to map ***critical***, ***high***, ***medium*** and ***low***.

We want to be able to rank the components based on their number of the highest severity defects. This presents the possibility to be able to evaluate ***severe defect-prone components*** (SDC). Unlike other studies [26-28] that have developed multiple models to predict two or three categories of defect severity, we can only devise an approach to have a single ranking of component based on its most severe defects. We describe this method in this section since we believe it can find practical use by other researchers and practitioners.

We keep in mind that a component can have many defects and therefore contain different severity values (i.e. different severities distributed over a component). For instance, a component can have 3 defects in this order {Critical=1, High=1, Medium=0, Low=1}. To rank according to the highest severity of defects requires that we make some transformation to give the highest weights to components according to their most severe defects.

We describe the transformation process we use for this purpose:

1. Given ***n*** number of components and ***m*** number of defect severities, we form an ***mxn*** matrix, where the column elements in the matrix stand for the severity values of a component in their order of severity.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

2. We form a new matrix B as follows; for each column element, starting from the first element, replace all elements below with zero iff the element above is greater than zero.

$$\forall a_{i,j}\ if\ a_{i,j} > 0, a_{k,j} = 0, for\ k = i+1, \ldots, m$$
$$i \in [1 : m-1]$$

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \cdots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \cdots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{m1} & b_{m2} & b_{m3} & \cdots & b_{mn} \end{bmatrix}$$

3. Form a weight row vector W of ***1xm*** dimension containing the sum of the maximum element of each row below the k[th] row in *B*. The last column element in W is kept as 0:

$$w_k = \sum_{i=k+1}^{m} \max_{j=1,\cdots,n} (b_{i,j}), for\ k = 1, 2, \cdots, m-1$$

$$W = [w_1, w_2, \cdots, w_{m-1}, 0]$$

4. Form a new ***mxn*** matrix W_D, where *W* is the diagonal elements and all other elements are zero

$$W_D = \begin{bmatrix} w_1 & 0 & 0 & \cdots & 0 \\ 0 & w_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

5. Form matrix C by dividing each element in B by itself

$$C = \begin{bmatrix} \frac{b_{11}}{b_{11}} & \frac{b_{12}}{b_{12}} & \frac{b_{13}}{b_{13}} & \cdots & \frac{b_{1n}}{b_{1n}} \\ \frac{b_{21}}{b_{21}} & \frac{b_{22}}{b_{22}} & \frac{b_{23}}{b_{23}} & \cdots & \frac{b_{2n}}{b_{2n}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{b_{m1}}{b_{m1}} & \frac{b_{m2}}{b_{m2}} & \frac{b_{m3}}{b_{m3}} & \cdots & \frac{b_{mn}}{b_{mn}} \end{bmatrix}$$

6. Lastly, compute $D = W_D * C + B$

For example, with components; $\mathbf{c_1}$: {Critical=2, Major=1, Average=0, Minor=1}, $\mathbf{c_2}$: {Critical=0, Major=1, Average=3, Minor=0}, $\mathbf{c_3}$: {Critical=0, Major=3, Average=0, Minor=0} and $\mathbf{c_4}$: {Critical=0, Major=0, Average=0, Minor=1} gives matrix:

$$A = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 1 & 1 & 3 & 0 \\ 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Following the transformation steps II-VI yields matrices:

$$B = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} W_D = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$D = \begin{bmatrix} 6 & 0 & 0 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From the matrix D's result, $\mathbf{c_1}$ has the highest weight of 6, followed by $\mathbf{c_3}$ with a weight 4, then $\mathbf{c_2}$ with a weight 2 and lastly $\mathbf{c_4}$ with a weight of 1.

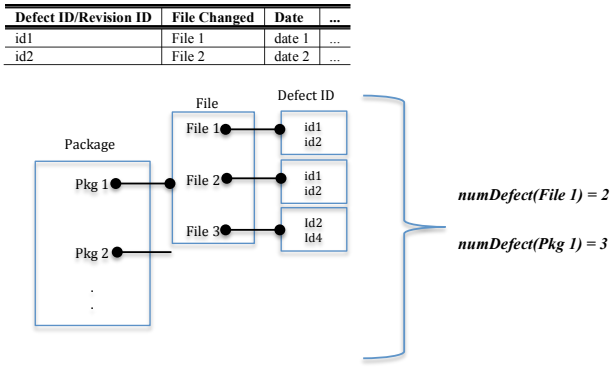


Fig. 2. Aggregating defect count at the package and file level [14]

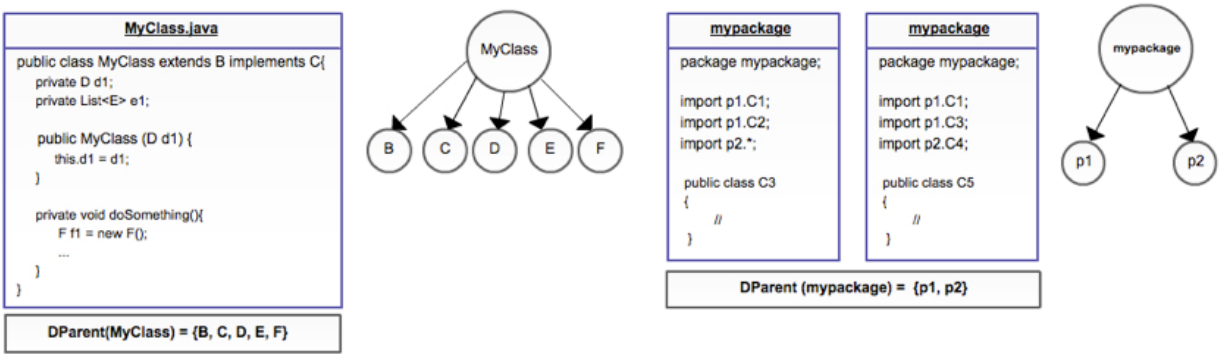


Fig. 3. (a) Class source data (b) Dependency Graph for Class (c) Package source data (d) Dependency Graph for package [14]

5) Dependency data collection

We have developed a small Java tool to extract source files dependency data [14]. The source files are downloaded from the version repository. Organizational rules in Java source file are substantially different from C# source file. A Java source file has a one-one mapping from file to top-level class and it is not allowed to define another top-level class in a Java file. In addition, the top-level class must have the same name as its enclosing file. Also, there is a one-zero or one-one mapping from file to package; a maximum of one package can be defined in a Java file. Finally, a Java class may contain nested classes (one to many relation). In C#, multiple relations are possible. A file can contain many top-level classes and many top-level namespaces can also be defined in a file. It is also possible that a class contains nested classes and a namespace can equally contain nested namespaces. Unlike Java file, the file name does not need to match any of the classes defined in it, although, good practices suggest to have filename as the same as a top-level class.

Since the compilation unit for both Java and C# is the source file and we are considering dependencies at the physical level [8], we decide for the following:

1. A dependency on any class in a source file implies a dependency on the source file.
2. The cyclic metric for a class is computed using dependencies that cross compilation units (source files). We skip cycles that are formed among classes within a source file.

We use the "USES" relations [8], which we have defined earlier as **DParent** and apply them to the two systems. We ignore all external library types (e.g., .NET and Java API) that developers have no access to their source codes since it is practically impossible for these external classes to form cycles with internal application's classes. Fig. 3 shows an example of the actual dependencies for **MyClass** and **mypackage** components. In order to collect other nodes (classes) to which **MyClass** is connected to requires that we scan the text of **MyClass**. The edge between **MyClass** and other **DParent(MyClass)** nodes is a directed path (without label, L) from **MyClass** to each node in the **DParent** set (Fig. 3a-b). In the case of **mypackage** (Fig. 3c-d), the **DParent(mypackage)**, is a set of unique imported packages and is processed from the collected class data.

IV. RESULTS

Table III lists the distribution of defect severity for the entire systems. Table IV lists the distribution of defects in each group. The total number of defects ($\mathbf{N_D}$), the number of defects in *in-cycle* group ($\mathbf{IC_D}$), the number of defects in the *depend-on-cycle* group ($\mathbf{DC_D}$), the total number of defects in both groups (i.e. $\mathbf{C_D = IC_D \cup DC_D}$) and the number of defects in the *non-cyclic* group ($\mathbf{NC_D}$) for both class-files and packages. In Table V, we report the results of mining the components (class-files and packages) into *in-cycle* (**IC**), *depend-on-cycle* (**DC**) and *non-cyclic* (**NC**) categories. In Table VI, we list for each severity value and SDC, the percentages of *cyclic-related defect-prone components* ($\mathbf{C_{DPC}}$), *non-cyclic defect-prone components* ($\mathbf{NC_{DPC}}$), the number of defects in *cyclic* group accounted for by *cyclic defect-prone components* ($\mathbf{C_D}$), the number of defects in *non-cyclic* group accounted for by *non-cyclic defect-prone components* ($\mathbf{NC_D}$), and the difference between the defects in *cyclic* group and *non-cyclic* group (i.e. $\mathbf{C_D\text{-}NC_D}$ and $\mathbf{NC_D\text{-}C_D}$).

Table VII reports the hypotheses tests of SDC, defect-prone components with critical severity defects and the number of defects with critical severity in *cyclic* and *non-cyclic* groups.

A. Distribution of Defect Severity

Table III lists the average distribution of defect severity in the six releases of the two systems. In Apache-ActiveMQ, 8% of defects are **Critical** (Blocker + Critical) defects and are distributed in 8% of defect-prone components[3] (DPC). 75% of defects are **High** (Major) severity defects and are spread across 78% of DPC, while 15% of total defects are **Medium** (Minor) severity defects and are distributed in 13% of DPC. Lastly, 2% of all defects are **Low** (Trivial) severity defects and are spread across 1% of DPC. In CommApp, 12% of defects are **Critical** (critical) severity defects and are distributed in 25% of DPC. 45% of defects are **High** (high) severity defects and are spread across 42% of DPC. 36% of defects are **Medium** (average) severity defects and are distributed in 27% of DPC and lastly, 7% of defects are **Low** (low) severity defects and are distributed in 6% of DPC.

Fig. 4 illustrates how much of defect-prone components affected by critical severity defects are accounted for when using the largest-first or the smallest-first[4] prioritization approaches [40]. For both systems, this distribution shows

[3] A defect-prone component as used in this study is defined as components with one or several defects

[4] Largest-first approach assumes that larger components are more defect-prone and therefore ranks components in the order of their highest number of defects while the smallest-first approach, however, assumes that smaller components are relatively more defect-prone

critical severity defects to spread across both DPC with large size and number of defects and those with small size and number of defects. At the top 25%, we could only account for less than 45% of DPC with critical severity defects. This number is, of course, not desirable for critical applications. Even at the top 75%, we are still unable to account for all the DPC with critical severity defects (since the percentage identified is approximately 80%).

In conclusion of this section, we can caution that models that target top k% may not uncover a significant number of defect-prone components affected by critical severity defects. At least this is confirmed in these two applications. There is a need for more studies in prediction methods that focus further on the severity of defects rather than number of defects. This is an additional motivation for us to conduct an investigation into how much of critical severity defects and defect-prone components with critical severity defects are contained in cyclic dependent components.

TABLE III. % (AVERAGE) OF DISTRIBUTION OF DEFECT SEVERITY

Severity	Apache-ActiveMQ		CommApp	
	DPC	#Defect	DPC	#Defect
Critical	8	8	25	12
High	78	75	42	45
Medium	13	15	27	36
Low	1	2	6	7

B. *Distribution of defect and DPC in cyclic and non-cyclic groups*

Tables IV and V list the average of cyclic data and their defect profiles for the two systems we investigated. On the average, in Apache-ActiveMQ, there are 1366.3 class-files out of which 75.2 are defective. 32.5% of class-files are in-cycle (**IC**) while 15.8% of class-files are dependent on other components in cycles (**DC**). 51.7% are not involved in any cyclic relationships (**NC**). Of the total defective class-files, both in-cycle and depend-on-cycle components account for 78.3%, while non-cyclic components account for 21.7%. The system contains an average of 44 defects, of which the **IC** group accounts for 84.5%, the **DC**, 17% and the **NC** 19.8%.

At the package level, there are average of 79.2 packages with 27.3 defective ones. 68.6% of packages are in-cycle while none is dependent on any in-cycle components. 31.4% of packages are not in any cyclic relationships. Furthermore, packages in-cycle account for 89% of the total defective components while non-cyclic packages account for 11%. Out of the total average of 44 defects in the system, **IC** group accounts for 93.2% while **NC** group accounts for 10.7%. As can be seen from these statistics, the cyclic related components in Apache-ActiveMQ account for the clear majority of defective components and number of defects at both the class-file and package levels.

For CommApp, the average class-files totaled 1010, out of which only 65 are defective from 25.7 defects. 24.8% of these class-files are in-cycle and account for 57.4% of defect-prone components and 80.5% of total defects. The **DC** group contains 28.8% of the class-files and account for 26.5% of defect-prone components and 51.4% of the defects. Lastly, the **NC** group has 46.4% of the total class-files and account for 16.2% of defect-prone components and 30% of total defects.

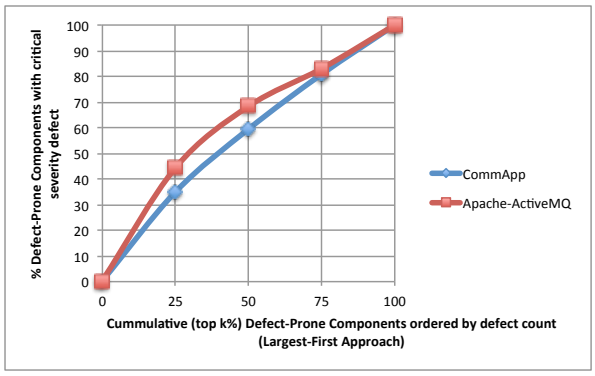


Fig. 4. % of DPC with **critical** defects identified at the top k% of the class-files DPC over six releases

At the package level, the CommApp contains an average of 161.5 packages of which 20.5 turned defective. 12.6% of these packages are in-cycle and account for 31.7% of defect-prone components and 68.9% of total defects. The **DC** group contains 40.7% of the packages and account for 37.6% of defect-prone components and 57.6% of the defects. Lastly, the **NC** group has 46.6% of the total packages and account for 30.7% of defect-prone components and 31.9% of total defects. As observed from these statistics, the cyclic related components in CommApp also account for the clear majority of both defect-prone components and the number of defects.

C. *Distribution of critical defects and SDC in cyclic and non-cyclic groups*

We now investigate if this majority in both defect-prone components and number of defects are also the clear majority in the number of critical defects and severe defect-prone components. As listed in Table VI, in Apache-ActiveMQ, the cyclic group of class-files contains 90.4% of SDC[5], that is, defect-prone components in the top 25% based on their number of critical defects while the non-cyclic group has 9.6%. Furthermore, the total percentage of the SDC defects[6] in cyclic group is 96.1% while that of NC group is 10.2%. Also, the cyclic group accounts for 90.3% of the defect-prone components with critical severity defects while the non-cyclic group accounts for 9.7%. At the package level, 97.7% of SDC are in cyclic relationships while 2.3% of SDC are not in cyclic relationships. SDC defects in the cyclic group account for 95% while 5.2% are recorded for non-cyclic group. In addition, all the defect-prone packages with critical severity defects are in cyclic relationships and they account for all the critical severity defects in this system.

In the case of CommApp, the cyclic group of class-files consist 88.6% of SDC and this number accounts for 94.7% of SDC defects. Furthermore, the cyclic group accounts for all (100%) the critical severity defects and contains 82.2% of defect-prone components affected by the critical severity defects. At the package level, cyclic group comprises 65.6% of SDC and accounts for 80.8% of SDC defects. Also, the cyclic group accounts for all (100%) of critical severity defects and contains 75% of defect-prone packages affected by critical severity defects.

[5] According to the ranking algorithm and the percentile figure used (Top 25% = 75th percentile), SDC might contain **high** severity defects in addition to **critical** severity defects.

[6] Note that defects in SDC are ranked as the highest severity defects.

Table VII lists the results of the hypotheses tests. Regarding the first hypothesis H_A, the p-values of 1-tailed test for the two systems and for both types of components (class-files and packages) are less than 0.05. Therefore, we reject the null hypothesis and confirm that the number of critical severity defects in the cyclic defect-prone components is significantly higher than those in the non-cyclic defect-prone components. Regarding H_B, we reject the null hypothesis for H_{B1} at both the class-file and package levels and affirm that defect-prone components with critical defects are significantly higher in the cyclic group than non-cyclic group. The null hypothesis H_{B2} is rejected for all cases except for package-level result for CommApp.

V. Findings and Discussion

First, as demonstrated in the distribution data of Fig. 4, defect-prone components affected by critical severity defects are spread across DPCs. In other words, prioritizing testing activities using the "largest-first" or "smallest-first" [40] approach is not optimal to discover such "first class" candidates that should be prioritized in critical systems. Furthermore, we revealed that all critical severity defects (100%) are located in the packages that are in cyclic relationships. Likewise, between 95% and 100% of critical severity defects can be discovered in the class-files that are in cyclic relationships.

When we look at the defect-prone components affected by those critical defects, we discovered that for cyclic related components, between 82.2% and 90.3% are class-files in cyclic relationships and between 75% and 100% are packages that have cyclic relationships. In addition to these,

TABLE IV. Summary of Components #defects[7] (Averaged over six releases)

System		Class-File				Package			
	N_D	IC_D	DC_D	C_D	NC_D	IC_D	DC_D	C_D	NC_D
ActiveMQ	44	37.2	7.5	39.8	8.7	41	0	41	4.7
CommApp	25.7	20.7	13.2	24.3	7.7	17.7	14.8	23.7	8.2

TABLE V. Summary of cyclic data (Averaged over six releases)

System	Class-File								Package							
	N	N_{DPC}	IC	IC_{DPC}	DC	DC_{DPC}	NC	NC_{DPC}	N	N_{DPC}	IC	IC_{DPC}	DC	DC_{DPC}	NC	NC_{DPC}
ActiveMQ	1366	75.2	444.7	50.4	215.3	8.5	706.3	16.3	79.2	27.3	54.3	24.3	0	0	24.8	3
CommApp	1010	65	250.5	37.3	290.6	17.2	468.8	10.5	162	20.5	20.3	6.5	65.8	7.7	75.3	6.3

TABLE VI. Average % of components in cyclic[8] and non-cyclic groups and grouped by defect severity

Metric	Apache-ActiveMQ						CommApp					
	C_{DPC}	NC_{DPC}	C_D	NC_D	C_D-NC_D	NC_D-C_D	C_{DPC}	NC_{DPC}	C_D	NC_D	C_D-NC_D	NC_D-C_D
Class-Files												
SDC (25%)*	**90.4**	**9.6**	**96.1**	**10.2**	**90**	**4**	**88.6**	**11.4**	**94.7**	**34**	**66**	**5.3**
Critical*	**90.3**	**9.7**	**95.1**	**14.3**	**86**	**4.8**	**82.2**	**17.8**	**100**	**55.6**	**44.4**	**Ø**
High	78.6	21.4	89.4	21.1	79	10.6	84.9	15.1	91.4	35.7	64.3	8.6
Average	75.8	24.2	92.3	18	82.1	7.7	84.5	15.5	96.3	18.2	81.8	3.6
Minor	100	0	100	0	100	Ø	90.6	9.4	100	9.1	91	Ø
Package												
SDC (25%)*	**97.7**	**2.3**	**95**	**5.2**	**94.8**	**5.2**	**65.6**	**34.4**	**80.8**	**61.5**	**38.5**	**19.2**
Critical*	**100**	**0**	**100**	**0**	**100**	**Ø**	**75**	**25**	**100**	**50**	**50**	**Ø**
High	88.4	11.6	94	11	89	6	62.6	37.4	87	40	60	12.9
Average	87.9	12.1	84	15.4	84.6	15.4	82	18	96.3	18.2	81.8	3.6
Minor	100	0	100	0	100	Ø	89.5	10.5	91.1	18.2	81.8	9.1

* Both categories that are focused in this study

TABLE VII. 1-tailed test for comparing highest severity defects and defect-prone components in cyclic and non-cyclic groups

System	CLASS	PACKAGE
	H_A : Test of Number of critical defects in Cyclic vs. Non-Cyclic groups	
	p-value (C)	**p-value (C)**
ActiveMQ	0.0113*	0.0012*
CommApp	0.0214*	0.0299*
	H_{B1} : Test of Number of Defect-prone components with critical defects in Cyclic vs. Non-Cyclic groups	
	p-value (C)	**p-value (C)**
ActiveMQ	0.0337*	0.0052*
CommApp	0.0295*	0.0125*
	H_{B2} : Test of SDC (Top 25%) in Cyclic vs. Non-Cyclic groups	
	p-value (C)	**p-value (C)**
ActiveMQ	0.0051*	0.0001*
CommApp	0.0003*	0.0586

* Significant at $\propto = 0.05$

[7] It is important to note that defects can overlap in both categories since a defect can spread to many components

[8] Note that **$C_{DPC} = (IC_{DPC} \cup DC_{DPC})$ and $C_D = (IC_D \cup DC_D)$**

when we rank components according to the number of their highest severity defects, we found that between 88.6% and 90.4% of defect-prone components (class-files) with the highest severity defects are in cyclic relationships. Also, between 65.6% and 97.7% of packages that are defect-prone and with the highest severity defects are in cyclic relationships.

The set differences $\mathbf{C_D}$-$\mathbf{NC_D}$ and $\mathbf{NC_D}$-$\mathbf{C_D}$ present useful perception into identifying defects that are unique to each group. The findings in this study show that with certainty, we can confirm that 86% (class-files) and 100% (packages) of critical severity defects originate from the cyclic group in Apache-ActiveMQ while 4.8% (class-files) of critical severity defects originate from the non-cyclic group. For CommApp, we are sure that 44.4% (class-file) and 50% (packages) of critical severity defects originate from the cyclic group whereas no critical severity defect can be said with certainty to originate from the non-cyclic group (The set difference NC_D-C_D = null).

One major contribution of this work is that we are able to partition a software dataset into sub sets that allows a maintenance engineer and software testers to look for defect and most especially critical severity defects in the right places. For instance, it is far more efficient to look for highest severity defects in 50% or less of a system´s components than the whole 100%. The cyclic related components in our study range between 48.3% and 53.6% of the class-files and within this range, we can discover between 95.1% and 100% of critical severity defects. Several empirical results already revealed that defect distribution in software systems is skewed and followed the Pareto rule (20-80) [16, 17]. The challenge is higher when dealing with large and complex systems with thousands of components but extremely few defect-prone components. In such situations, prediction models have lower chance of good performance. It is even worse when fewer of those components are associated with critical severity defects, which is the case in many software systems. Finding them can be analogous to looking for a needle in a haystack (see Fig. 4). To reinforce this point, the study in [26] confirmed that prediction based on low severity defect performed better than prediction on high severity defects.

The results in this study are useful to employ for focusing testing resources and refactoring possibilities in both industry and the academia. Many studies [9-13] of dependency cycle in software systems suggest that cycles are inherent in real-life systems and appear like a menace we have to live with. We confirm our conjecture in these two systems that dependency cycles contain the highest severity defects. We rush to say that we make no claim to this pattern in all systems, however, we believe this is a step forward to encourage further studies and to understanding dependency cycles, defect-prone components and defect severity.

VI. Threats to Validity

We have performed analysis and evaluation of an industrial Smart Grid system and a messaging and enterprise integration pattern server. Therefore, we cannot claim that this kind of pattern or related will be visible in other systems and other domains. As it is with most case studies, we cannot generalize these results across all systems. Further studies will be necessary to compare results across several systems.

Our study is based on static coupling measurements and not dynamic coupling measurements [41]; as such actual coupling among classes at runtime may not be completely captured. This imprecision can occur due to polymorphism, dynamic binding and dead code in the software. This as it may, static code analysis has been found to be practically useful and less expensive to collect [5, 6, 21, 23, 42-44]. In addition, we collect coupling types that are not only based on method invocation. We do not think the data collected based on static code analysis can bias our result in any significant manner.

For this study, we have relied on the defects logged in the defect tracking systems of each application. Our approach of defect data extraction is similar to what other researchers have used in the past [37-39]. Nevertheless, common threats are whether defects logged in the DTS are accurately tagged in the respective code changes in the version systems. In addition, we cannot be sure if all defects are logged in the DTS. Also, there could be cases that the message log of the file that consists a change is not tagged with the bug numbers of the resolved defect. Furthermore, there could be cases of typographical error in the recording of the bug number in the version systems [39] and lastly, it is still possible that duplication will occur.

The recording of defect severity in many defect-tracking systems has been argued to be subjective [45]. We cannot exclude the possibilities of subjective severity records in the DTSs that we have used. However being critical applications and from our investigation of the repositories, most records that impact on reliability, performance and/or security point to the highest severity values (blocker/critical). These are, essentially, the focus in our analysis and therefore, we can rely on the quality of the data to a great degree.

VII. Conclusions

We have empirically investigated if defects with the highest criticality and the components impacted by such defects are mostly concentrated in cyclic dependent components. Our findings based on the two non-trivial systems we investigated revealed that DPC with critical defects are spread across the systems. Furthermore, we confirmed our conjecture that cyclic dependent components account for almost all of the critical severity defects and most severe defect-prone components.

Empirical analysis shows that in three out of four cases, all the highest severity defects are found in components that are involved directly or indirectly in cyclic relationships and in the 4th case, over 95% of the highest severity defects are discovered in the cyclic related group. The results in this study have practical use in allocating testing resources to a subset of systems with the highest likelihood of containing the most critical defects. Furthermore, it provides reasoning for refactoring and/or reengineering of especially defect-prone cyclic dependent components with critical defects.

Lastly, it shows a subset of software systems that can be further explored for improved prediction models based on defect severity. As future studies, we aim to conduct a large empirical study of critical systems with well-maintained repositories to understand if the findings in this study relate to a general pattern in systems with cyclic relationships. Furthermore, dataset imbalance as discussed in Hall et al. [45] is a threat to prediction models' performance. We

speculate that we can explore the results of dividing the datasets into these categories to build better models that can predict defect-proneness of components based on defect severity.

REFERENCES

[1] *IEEE Recommended Practice on Software Reliability.* IEEE STD 1633-2008, 2008: p. c1-72.

[2] Lilley, S., *Critical Software: Good Design Built Right.* NASA System Failure Case Studies, 2012. **6**(2).

[3] Leo, K. *Why banks are likely to face more software glitches in 2013*. [Web] 2013 April 24, 2013]; Available from: http://www.bbc.co.uk/news/technology-21280943.

[4] Fenton, N.E. and S.L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*. 2nd ed. 1997, Boston: PWS Publishing Press.

[5] Briand, L.C., et al., *Predicting fault-prone classes with design measures in object-oriented systems.* Ninth International Symposium on Software Reliability Engineering, Proceedings, 1998: p. 334-343.

[6] Briand, L.C., J. Wuest, and H. Lounis, *Replicated Case Studies for Investigating Quality Factorsin Object-Oriented Designs.* Empirical Softw. Engg., 2001. **6**(1): p. 11-58.

[7] Fowler, M., *Reducing coupling.* Software, IEEE, 2001. **18**(4): p. 102-104.

[8] Lakos, J., *Large-scale C++ software design*. 1996, Redwood City, CA: Addison-Wesley Longman.

[9] Parnas, D.L., *Designing Software for Ease of Extension and Contraction.* IEEE Transactions on Software Engineering, 1979. **SE-5**(2): p. 128-138.

[10] Briand, L.C., Y. Labiche, and W. Yihong. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. in Proc. 12th International Symposium on Software Reliability Engineering, (ISSRE) 2001.

[11] Hanh, V.L., et al., *Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies.* Proc. 15th European Conf. Object-Oriented Programming (ECOOP), 2001: p. 381-401.

[12] Kung, D., et al., *On Regression Testing of Object-Oriented Programs.* Journal of Systems Software, 1996. **32**(1): p. 21-40.

[13] Melton, H. and E. Tempero, *An empirical study of cycles among classes in Java.* Empirical Software Engineering, 2007. **12**(4): p. 389-415.

[14] Oyetoyan, T.D., D.S. Cruzes, and R. Conradi, *A Study of Cyclic Dependencies on Defect Profile of Software Components.* Journal of Systems and Software, 2013. (in press)

[15] Adams, E.N., *Optimizing Preventive Service of Software Products.* IBM Journal of Research and Development, 1984. **28**(1): p. 2-14.

[16] Fenton, N.E. and N. Ohlsson, *Quantitative analysis of faults and failures in a complex software system.* IEEE Transactions on Software Engineering, 2000. **26**(8): p. 797-814.

[17] Boehm, B. and V.R. Basili, *Software Defect Reduction Top 10 List.* Computer, 2001. **34**(1): p. 135-137.

[18] Ebert, C., et al., Defect Detection and Quality Improvement, in Best Practices in Software Measurement. 2005, Springer Berlin Heidelberg. p. 133-156.

[19] Jungmayr, S. Identifying test-critical dependencies. in Software Maintenance. 2002.

[20] Abreu, F.B.E. and W. Melo, *Evaluating the impact of Object-Oriented design on software quality.* Proceedings of the 3rd International Software Metrics Symposium, 1996: p. 90-99.

[21] Chidamber, S.R. and C.F. Kemerer, *A Metrics Suite for Object-Oriented Design.* IEEE Transactions on Software Engineering, 1994. **20**(6): p. 476-493.

[22] Nagappan, N. and T. Bhat, *Technologies for Code Failure Proneness Estimation*, 2007, Microsoft Corporation: USA.

[23] Zimmermann, T. and N. Nagappan, *Predicting Defects using Network Analysis on Dependency Graphs.* 2008 30th International Conference on Software Engineering: (ICSE), Vols 1 and 2, 2008: p. 530-539.

[24] Martin, R.C., *Granularity, C++*, 1996. p. 57-62.

[25] Briand, L.C., Y. Labiche, and W. Yihong, *An investigation of graph-based class integration test order strategies.* Software Engineering, IEEE Transactions on, 2003. **29**(7): p. 594-607.

[26] Zhou, Y.M. and H.T. Leung, *Empirical analysis of object-oriented design metrics for predicting high and low severity faults.* IEEE Transactions on Software Engineering, 2006. **32**(10): p. 771-789.

[27] Shatnawi, R. and W. Li, The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. Journal of Systems and Software, 2008. **81**(11): p. 1868-1882.

[28] Singh, Y., A. Kaur, and R. Malhotra, *Empirical validation of object-oriented metrics for predicting fault proneness models.* Software Quality Journal, 2010. **18**(1): p. 3-35.

[29] Bhattacharya, P., et al., *Graph-Based Analysis and Prediction for Software Evolution.* 2012 34th International Conference on Software Engineering (ICSE), 2012: p. 419-429.

[30] Menzies, T. and A. Marcus, *Automated Severity Assessment of Software Defect Reports.* 2008 IEEE International Conference on Software Maintenance, 2008: p. 346-355.

[31] Lamkanfi, A., et al., *Comparing Mining Algorithms for Predicting the Severity of a Reported Bug.* 2011 15th European Conference on Software Maintenance and Reengineering (CSMR), 2011: p. 249-258.

[32] Iliev, M., et al. Automated prediction of defect severity based on codifying design knowledge using ontologies. in First International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2012.

[33] Yang, C.-Z., et al. An Empirical Study on Improving Severity Prediction of Defect Reports Using Feature Selection. in Software Engineering Conference (APSEC), 19th Asia-Pacific. 2012.

[34] Rahimi, F. and A. Ipakchi, *Demand Response as a Market Resource Under the Smart Grid Paradigm.* Smart Grid, IEEE Transactions on, 2010. **1**(1): p. 82-88.

[35] Li, Z.D., et al., Characteristics of multiple-component defects and architectural hotspots: a large system case study. Empirical Software Engineering, 2011. **16**(5): p. 667-702.

[36] Gupta, A., et al., *An examination of change profiles in reusable and non-reusable software systems.* Journal of Software Maintenance and Evolution-Research and Practice, 2010. **22**(5): p. 359-380.

[37] S'liwerski, J., T. Zimmermann, and A. Zeller, When do changes induce fixes?, in Proceedings of the 2005 International Workshop on Mining Software Repositories 2005, ACM: St. Louis, Missouri. p.1-5.

[38] Schroeter, A., T. Zimmermann, and A. Zeller, Predicting component failures at design time, in Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering 2006, ACM: Rio de Janeiro, Brazil. p. 18-27.

[39] C'ubranic, D., *Project History as a Group Memory: Learning From the Past.* , in *PhD Thesis* 2004, University of British Columbia: Canada.

[40] Koru, A.G., et al., *Theory of relative defect proneness.* Empirical Software Engineering, 2008. **13**(5): p. 473-498.

[41] Arisholm, E., L.C. Briand, and A. Foyen, *Dynamic coupling measurement for object-oriented software.* IEEE Transactions on Software Engineering, 2004. **30**(8): p. 491-506.

[42] Basili, V.R., L.C. Briand, and W.L. Melo, *A validation of object-oriented design metrics as quality indicators.* IEEE Transactions on Software Engineering, 1996. **22**(10): p. 751-761.

[43] Zimmerman, T., et al. An Empirical Study on the Relation between Dependency Neighborhoods and Failures. in IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST). 2011.

[44] Xu, J., D. Ho, and L.F. Capretz, *An Empirical Validation of Object-Oriented Design Metrics for Fault Prediction.* Journal of Computer Science, 2008. **4**(7): p. 571- 577.

[45] Hall, T., et al., *A Systematic Review of Fault Prediction Performance in Software Engineering.* IEEE Transactions on Software Engineering, 2011. **99**(PrePrints).

Code Clustering Workbench

Annervaz K M, Vikrant Kaulgud, Janardan Misra, Shubhashis Sengupta, Gary Titus, Azmat Munshi
Accenture Technology Labs
Bangalore, India
Email: {annervaz.k.m, vikrant.kaulgud, janardan.misra, shubhashis.sengupta, gary.titus, azmat.munshi }@accenture.com

***Abstract*—Source code clustering is an important technique used in software development and maintenance to understand the modular structure of code. An array of algorithms are available for clustering like simulated annealing based search. Source code have different kinds of features such as structural or textual features. The collection of these different types of source code features and computation of relevant feature metrics is a difficult task. Further, the clustering algorithms can run on metrics based on different types of source code features or their combinations. This flexibility makes it non-trivial to test effectiveness of clustering algorithms on a source code. In this paper, we present a highly configurable clustering workbench that allows the user to collect the various source code features and then to select the code features used for clustering, the clustering algorithm and its various parameters. Clustering quality metrics are computed. They allow comparison of algorithm output based on different combinations of code-features and algorithms. We also present the specific contribution made in multi-dimensional feature analysis and clustering. The tool hides the algorithm complexity from the user, thus allowing complete focus on understanding the 'effect' of the configuration choices. We have also applied this tool in real-life maintenance projects, where the users found it useful to tweak the clustering techniques for the source-code peculiarities.**

***Index Terms*—Clustering, Source Code Analysis, Semantic Indexing, Experimental Workbench.**

I. Introduction and Motivation

Source code is a vital artifact in a software development environment as it accurately describes the application behavior and functionality. Particularly in case of Java applications, due to the use of several technical services, design patterns, and coding standards, source code is difficult to comprehend than it's higher-level abstractions such as class diagrams or sequence diagrams. The difficulty is even more pronounced when comprehending legacy code. *Source code analysis* helps developers extract and reconstruct higher-level abstractions of the application. It also helps developers locate functional features in code. Using these abstractions and feature location, the task of code comprehension is greatly simplified.

In our interactions with developers in maintenance projects, we identified extracting functionally-cohesive clusters from code as a key code-comprehension requirement. This allows developers to understand the modular nature of a complex application and focus on the modules that contain the relevant functionality. Clustering, however, is a complex process. It requires the use of different techniques and algorithms: feature extractors (e.g., AST Parsers, dependency graph generator), clustering algorithm (e.g., graph partitioning algorithm), and a visualization framework. Unless one uses an end-to-end tool chain, it is quite difficult to plumb disparate tools required for clustering. Further, tweaking the techniques for optimal results can become difficult, especially for novice users. Even for expert users, setting the multiple parameters becomes a tedious activity.

978-1-4673-5739-5/13/$31.00

The challenges of selecting, configuring and plumbing disparate source code analysis techniques and tools is the motivation for the Code Clustering Workbench (CCW). The objective of designing CCW is to provide a **configurable** tool that integrates best-of-breed analysis techniques for code clustering. CCW allows fine-grained configuration of each step of code clustering - feature extraction, class-class similarity generation, clustering algorithm tuning, cluster refinement, and visualization support. The tool outputs all intermediate analysis outputs in convenient CSV and XML formats. The tool is based on Python and Java. CCW can be executed on a Windows and Linux/Unix environments having standard Python and Java installations. We have used the current prototype in real-life maintenance projects as well as on-going study of optimizing clustering of large legacy code.

In this paper, we share the architecture of CCW, the configuration options we found desirable from a power user's perspective and the intermediate analysis outputs generated. Section II describes the CCW tool architecture in detail. Section III highlights the key configuration parameters. We briefly describe the planned work in Section IV.

We have provided a video of the tool's installation and usage at http://youtu.be/vP552YRyz-s . Furthermore, to give readers a better understanding of the tool's output, we have provided the results of CCW's analysis on XERCES2110(a reference Java application) at https://sites.google.com/site/ccwsampledata/ .

II. CCW Tool Architecture

A high level schematic architecture of the tool is given in the Figure 1. CCW expects following inputs: Java Source Code, Java Byte Code and Textual descriptions for Functional Components (optional). Based on the user specified configuration, CCW generates the following outputs: Extracted Features (for each class), Similarity Measures (for every pair of classes), and Functional Component to Cluster Mapping.

CCW provides support for collecting and analyzing mainly three kinds of features of Java source code: Textual Features, Code Features and Structural Features. The subsequent sections describe how data is collected and analysed for

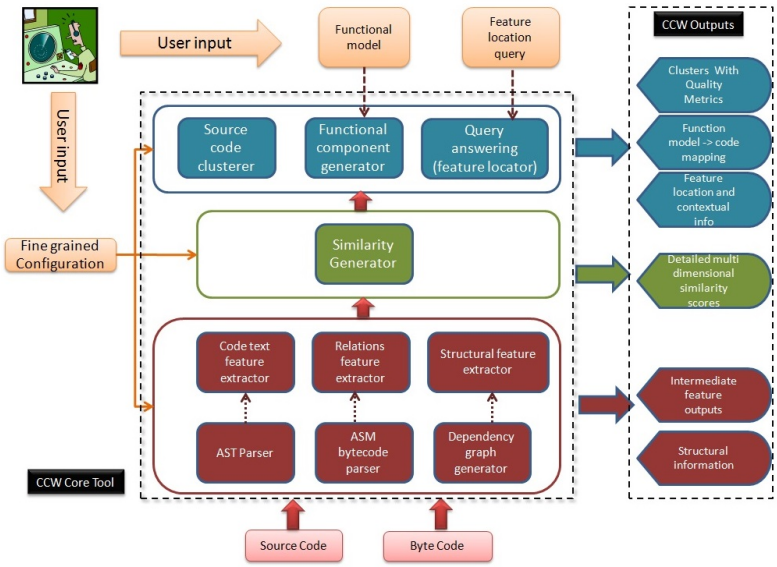


Fig. 1. Basic Tool Architecture

these features. A detailed, algorithm-level description of these sections is presented in the on Java code clustering by Misra et al. [1] [2]

A. Collecting and Analyzing Textual Features

Textual Features includes tokens extracted from code comments and identifiers used in the source code. For example, comment string "This ControllerClass will schedule processes" is split into words {'this', 'controllerClass','will', 'schedule', 'processes'}. The tokenizer extracts list of tokens from these words as {'this', 'controller', 'Class','will', 'schedule', 'processes'}. Next the reserved words and stop words are removed from the lists generated. Word stemming is applied to these tokens. In the example, final list of tokens will be {'control', 'schedule', 'process'} Textual features are directly extracted by parsing the source code using Python regular expressions.

For each Java class we collect the tokens and form the word co-occurrence vector. Word co-occurrence vector for a particular class represents the frequency of occurrence of each word in that class' source code. After populating this data we form the word co-occurrence matrix C, where $C[c, w]$, denotes the frequency of occurrence of word 'w' in class 'c'. This data is printed in a '.csv' file named ***WordsFrequencyTable*** in CCW's output folder. After populating this data, we tabulate the number of different classes in which each word appears. This data is printed in a '.csv' file named ***WordsOccuringDocumentCounts***. After this step, the TFIDF matrix is computed, by applying the following transformation to word co-occurrence matrix, $C[c, w] = C[c, w] \times \log(\frac{d}{n})$ where n is the number of classes where token w appears and d is the total number of classes present in the application. The TFIDF table is printed in a '.csv' file named ***TFIDFTable***.

In the standard Vector Space Model(VSM), we compute the similarity between two classes as the cosine similarity between the corresponding vectors of the classes in the TFIDF table. If LSI is enabled in the options, we compute the LSI space by applying SVD on co-occurrence matrix and following [3] and choosing $\max\{30, (d * t)^{0.2}\}$ as the reduced dimension of LSI space, where d is the number of classes and t is the number of different tokens altogether. Then we compute the cosine similarity of the corresponding vectors in the LSI space to find the textual similarity between two classes. The computed textual similarity for every pair of classes is printed under the column *TextualSimilarity* in the output '.csv' file named ***AllSimilarityMeasures***

1) Word Stemming Options and Stop Words: Tool provides *four options* for stemming algorithms. The supported algorithms are,

1) Porter2: Standard Porter2 algorithm
2) Custom: Rule based stemmer, which includes basic stemming rules of plurals, 'ing', and past tense form of English language
3) LookUp: Stemming based on the stem-list published by [4] consisting of more than 21000 words and their stems based upon extensive analysis of more than 9000 open source Java projects.
4) Lemmatize: Open NLTK based Lemmatizer [5].

Stop words that need to be removed from textual analysis are taken as an input in the configuration file.

B. Collecting and Analyzing Code Features

In code features, we include three types of features for each class in the Java source code: Class-Name, Inheritance and Interface-Implementation Relations, and Packaging structure. The extraction of class-names is straightforward. The inheritance and implementation relations are populated for each class, as a list, using a Java AST parser [6]. The packaging

information for each class is read from the source file using Python regular expression. After processing each class, we get the required class-name, inheritance and interface-implementation relations list and package-name as different lists. We break the data in these lists and extract *concept words* out of it.

For example, a Class FinanceControlManager contained in package 'com.atl.application.controlManager', is represented by concept tokens {'com', 'atl', 'application', 'control', 'manager'} and class-name would result into a list of concept words as {'Finance', 'Control', 'Manager'}.

Inheritance / interface-implementation relationship list is also split to a list of concept words. For example consider, *Class ClientAnalytics implements Business extends ClientScheduler*. The concept word list from inheritance/implementation relation for the class ClientAnalytics would be {'Business', 'Analytics', 'Client', 'Scheduler'}.

These concept tokens are printed in a '.csv' file named ***ConceptWords***. The similarity between concept words is computed using an adapted version of standard Jaccard Similarity [7]. First we calculate an Inverse Document Frequency (IDF) for each concept word. For every word w, this is equal to $\log(\frac{d}{n})$, where d is the total number of classes and n is the number of classes where the word w appears. Now, for two sets of concept words for class A and class B, we compute the intersection and union of these sets. Then we calculate the sum of the IDFs of the concept words occurring in the intersection of sets divided by the sum of the IDFs of the concept words occurring in the union of sets. This value we treat as the similarity measure between class A and B. The similarity between each code feature is computed separately for each class-pair. The computed code feature similarities for every pair of classes are printed under the columns *JaccardSimilarityOnClassNames, JaccardSimilarityOnRelations, JaccardSimilarityOnPackageStructure* in the output '.csv' file named ***AllSimilarityMeasures***.

C. Collecting and Analyzing Structural Features

The collection of structural relations (i.e. dependency information) is done using an open source tool *DepFinder*. CCW processes the dependency information generated by DepFinder to populate a table containing IDFs of various functions and classes in the source code. For every function f or class c this is equal to $\log(\frac{d}{n})$, where d is the total number of classes and n is the number of classes where the function f or class c is being used. This data is printed in '.csv' files named ***FunctionOrVariableIDFs, ClassIDFs***.

After this we calculate the structural similarity between two classes A and B, by adding a value equivalent to IDF of A, if A is being used in Class B, to the structural similarity measure between A and B or by adding a value equivalent to IDF of A.f, if A.f is being used in class B. To bring the value range of structural similarity between 0 and 1, the information-theoretic weighed structural similarity between all pair of classes is normalized by dividing a similarity score with the maximum value in the structural similarity set. This data is printed under the column *StructuralSimilarity* in the output '.csv' file named ***AllSimilarityMeasures***.

D. Combining Similarity Measures

After coming up with the various similarity measures between all pair of classes as described in the previous sections, we combine them in a weighted fashion. The weights for combining is read from the configuration. This data is printed under the column *CombinedSimilarity* in the output '.csv' file named ***AllSimilarityMeasures***.

E. Clustering Based On Modularization Quality

CCW represents the source code as a weighted undirected graph, where the nodes represent the classes and edges represent an existence of similarity between classes. The edge weight presented the combined similarity between classes. We search and find the partition of this graph, which maximizes Modularization Quality(MQ) metric [8]. For a particular graph partition this is defined as the sum of intra member edge weights divided by the sum of intra and inter member edges, summed over all the individual members (clusters) of that particular partition. The problem of finding the optimal graph partition which maximizes this quantity is a known NP-Hard problem[9].We have implemented *four search algorithms* to find a sub optimal graph partition which gives good MQ: Next Good Neighbour, Next Best Neighbour, Simulated Annealing and Genetic Algorithm.

The first three graph partitioning algorithms use a global neighbourhood search. For a graph partition, a neighbouring partition is defined as another partition, which is generated by the movement of one node in the graph from a particular member cluster of the partition to another cluster. 'Next Good Neighbour' search starts with a heuristically generated graph partition and then in each iteration proceeds to search for any other neighbourhood partition, which has a MQ higher than the current partition. It continues the search in the next iteration from that neighbouring partition till it can find a neighbour which has a higher MQ than itself. The 'Next Best Neighbour' search also starts with heuristically generated graph partition but it proceeds to find that neighbouring partition, which has the highest MQ among all the neighbouring partitions of the current partition and continues this search in the next iteration from that neighbouring partition till it cannot find any neighbour which has MQ higher than itself. Simulated Annealing based search is also a neighbourhood search, starting with a heuristically generated graph partition. However, the key difference is that, lower MQ valued neighbouring partitions may be selected in the search process with some probability. The probability of choosing such a neighbouring partition with a lower MQ is higher in the initial iterations of the search and will subsequently reduce. Simulated Annealing details are present in [10]. The Simulated Annealing constant is a user configurable parameter in the tool. The user can also configure the number of iterations (i.e., steps required for the selection and movement of a node from one partition to its neighbour)

the search has to proceed and stop, giving out the best partition found so far.

Heuristically generated initial graph partitions mentioned in the previous paragraph for all the three search algorithms are connected components of the graph or non-intersecting cliques of the graph, which are generated by considering only those edges having weight more than x percentile (a configurable parameter to the tool, e.g., 0.97) of the total edge weights of the graph. This seed partition for the search algorithms is printed in a '.xml' file named ***InitialSeedClusters***. Thereafter depending on the algorithm selected by the user we run a search for better MQ valued clusters. Final result is printed in a '.xml' file named ***Clusters***. Alternatively, if the user has enabled JSON format output in the configuration, a **'.json' file named *Clusters*** is generated. This is particularly useful for visualization using libraries like Prefuse. Tool also provides an option to control the search time by configuring the number of max iterations for which the search has to proceed.

1) Border Line Cases and Cluster Interactions: After generating the clusters, we list out the cases where two classes are being included in separate clusters, although their similarity value is greater than x percentile of the total edge weights. This data is printed in a '.xml' file named ***BorderLineCases***.

The method interactions that happen across clusters is also listed out and printed in a '.xml' file named ***ClusterInteractions***. If the user has enabled JSON format output, cluster interactions data will also be printed in the *'.json' file named Clusters* along with the actual cluster information.

F. Hierarchical Clusters

After generating the clusters of classes, we consider each cluster as a single node, construct the graph of clusters, with the similarity measures between them as the normalized sum of the similarity measures of the members of the clusters. Thereafter we apply the same clustering process as described before and generate a *cluster of clusters*. We print this information in a '.xml' file named ***HierarchicalClustersLevel1***. Tool continues this process recursively, until it ends up with a single cluster or MQ value cannot be improved further. At each level L, this data is printed in a '.xml' file named *HierarchicalClustersLevel*L.

G. Functional Component Mapping

In a maintenance scenario, users typically have business knowledge or high-level architecture knowledge. They are aware of the *functional components* that exist in the application. However, for legacy applications, often times, the documentation related to mapping of these functional components to the actual code is missing.

CCW supports mapping of natural language descriptions of the functional components to the code cluster. In order to achieve this, CCW adds up the word co-occurrence vectors of the individual members of the clusters and treat this as the word co-occurrence vectors of the clusters. This data is available in '.csv' files named ***WordFrequencyTableForClusters, TFIDFTableForClusters***. The user inputs the functional components descriptions. CCW forms the word co-occurrence vectors for these natural descriptions. Further, the cosine similarities between the word co-occurrence vectors of the clusters and the functional component descriptions is computed. The values of these similarities are printed in a '.csv' file names ***FunctionalComponentsClustersSimilarities***. The mapping with the highest similarity denotes the code cluster most likely to be associated with a functional component. This mapping from functional components to highest matching cluster are then printed in '.xml' file named ***FunctionalComponentsMapping***.

H. Cluster Quality metrics

The comparison and quality analysis of clusters is a very difficult task. The usual strategy in doing this is in comparison with an existing set of clusters, which is usually called as Golden cluster set. The minimal number of move and join operations that is required to convert the generated clusters to the golden clusters is what is used to calculate a metric called *mojo* [11] distance. An optimal algorithm for the calculation of this metric is described in [11]. We have implemented this algorithm and calculate this for the clusters produced to facilitate ease of analysis of these clusters. We also compute the centroids of various clusters, along with the connectedness of various members with the centroid, and output these data.

III. CCW Configuration

The following are the main configuration options available in the tool and their descriptions. For brevity, we include the key configuration parameters only. A screen shot of the UI capturing the options for clustering, in given in Figure 2.

1) RunMode
   a) Values: Full Run, Feature Extraction, Similarity Measures Generation, Clustering, and Query Answering.
   b) Description: Selects which all functions tool has to perform.
2) HierarchicalClustering
   a) Values: yes, no
   b) Description: Enables or disables hierarchical clustering
3) ApplyLSI
   a) Values: yes, no
   b) Description: Enables or disables Latent Sematic Indexing for textual feature analysis and query answering
4) FunctionalComponentsMapping
   a) Values: yes, no
   b) Description: Enables or disables functional components to cluster mapping
5) TopNForQueryAnswering
   a) Values: Positive Integer Value
   b) Description: Number of top results to be displayed as query answers
6) StemmingAlgorithm
   a) Values: Custom, LookUp, Porter2, Lemmatize
   b) Description: Selects the stemming algorithm to use
7) IdentifyBorderLineCases
   a) Values: yes, no
   b) Description: Generates border line cases in clustering, if enabled

8) GenerateClusterInteractions
   a) Values: yes, no
   b) Description: Generates cluster interactions, if enabled
9) PrintIntermediateClusters
   a) Values: yes, no
   b) Description: Outputs the intermediate clusters in the cluster search phase.
10) ClusterSearchMaxSteps
   a) Values: Positive integer value greater than 100
   b) Description: Number of iterations the cluster search has to proceed.
11) JavaKeyWords, StopWords
   a) Values: ',' separated list of words
   b) Description: Stop words that have to be removed from processing
12) SimilarityWeightage
   a) Values: ',' separated list of five floating point numbers between 0 & 1, which adds to 1
   b) Description: Relative weights of individual similarity measures while calculating the combined similarity measure
13) SimulatedAnnealingConstant
   a) Values: Floating point value between 0 & 1
   b) Description: Constant used in Simulated Annealing algorithm
14) ThresholdToStartApplyingBruteForceSearch
   a) Values: Positive integer value
   b) Description: A threshold below which brute force search will be fired for optimal cluster search.
15) UpperCutOffPercentileForCCClustersGeneration
   a) Values: Floating point value between 0 & 1
   b) Description: x percentile value, for the edge weights that have to be considered in the initial seed cluster generation
16) BorderLineCaseCutOffPercentile
   a) Values: Floating point value between 0 & 1
   b) Description: x percentile value, for the edge weights that have to be considered and identified as a border line case after clustering
17) ClusterSearchAlgorithm
   a) Values: 'NextGoodNeighbour', 'NextBestNeighbour', 'SimulatedAnnealing', 'GeneticAlgorithm'
   b) Description: Search algorithm to be used for cluster search
18) GeneticAlgorithmFitnessCutOff
   a) Values: Positive floating point value greater than 1
   b) Description: Genetic algorithm fitness cut off after which search has to be stopped.
19) CalculateMojoDistance
   a) Values: yes, no
   b) Description: Whether the mojo [11] distance of the generated clusters has to be calculated or not, compared to a set of golden clusters.
20) ClusterOutputFormats
   a) Values: JSON, GraphML, .net, YAML
   b) Description: The formats in which the clusters have to be generated.

A. Tool execution phases

CCW allows *partial execution* as well. For example, a user might be interested in performing only feature extraction, or based features collected earlier the user would like to resume analysis. Following modes are currently supported: Full Run, Feature Extraction, Similarity Measures Generation, Clustering, and Query Answering.

The **'FeatureExtraction**' mode, only extracts and prints this data as required. The **'SimilarityGeneration**' model, analyzes the various features and comes up with the similarity measures between each class pairs. All the individual similarity measures and the combined one are given as outputs for further analysis. Both the data, before information theoretic weighing is applied and after its application are provided at all stages. The combination option provided in terms of relative weights equips the users to analyze the effect of various similarity measures in the final clusters generated. **'Clustering'** mode allows incremental execution of only clustering based on previously computed similarity measures, and optionally functional component mapping. **'Query Answering'** is an experimental feature that is used for semantic code search based on long natural language queries.

B. Comparison with BUNCH tool

BUNCH tool [12] [8] is a major prior work in the clustering based component identification from the source code. We present a brief comparison with BUNCH. While conceptually, BUNCH is a code clustering tool, there are a two key differences between CCW and BUNCH: the manner in which feature extraction is handled, and richness of clustering algorithms and configurability.

The BUNCH tool considers only structural features of source code, where as CCW the capability to analyze a multitude of other source code features as discussed earlier. Our tool provides the flexibility to combine the similarity scores from various features to compute the final similarity score used for clustering. Setting this combination to use only structural similarity will make our tool behave similar to BUNCH. BUNCH does not weigh the features in an information theoretic manner, where as we have the option to do relevant information theoretic weighing for each of the features collected.

In the clustering phase, CCW supports different algorithms for cluster search. CCW provides total flexibility in controlling the behaviour of clustering phase, such as deciding the algorithm to be used for clustering and the various parameters and inputs for each of these algorithms. CCW has rich options such as to start clustering from a set of seed clusters, generate the clusters in various formats, marks the boundary cases, compute the cluster quality metrics, map functional components to clusters, and output the intermediate clusters from the search phase. This richness in capability and complete flexibility given for the user makes CCW a complete workbench for source code clustering.

IV. Future Enhancements

In the present version of CCW, we support four clustering algorithms. We intend optimize the implementation to improve tool performance. We also intend to implement a more refined Genetic Algorithm[13], Spectral Partitioning, and

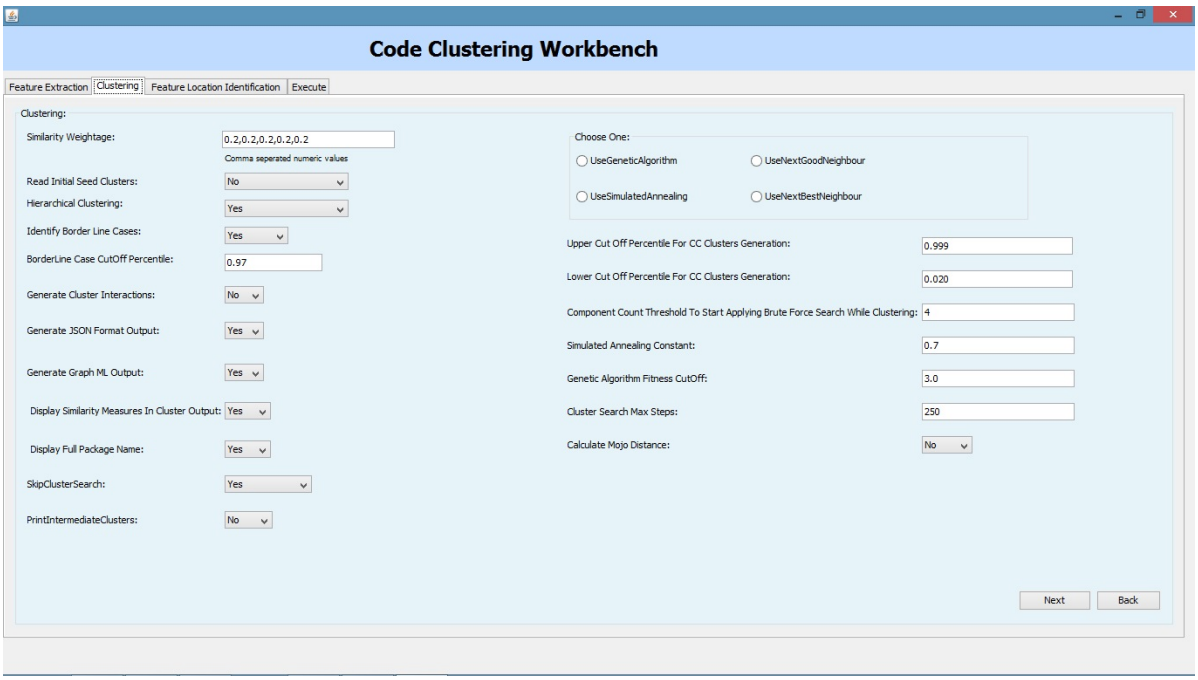


Fig. 2. Configuration UI

iterated Min-cut based algorithms for optimal cluster search. With these additional algorithms, CCW becomes a complete platform for code clustering. It will also help power users to compare and benchmark the performance of various clustering algorithms.

We also intend to provide an input mechanism for specifying reduced dimension for performing LSI. This is a simple addition to the existing implementation. Currently this is taken as a function as mentioned in Section II-A. With this addition, user will have a platform to experiment the effect of LSI reduced dimension parameter [14] on accuracy across various corpora and domains.

V. Conclusion

We presented CCW, a configurable code clustering workbench. It provides users with fine-grained control on various aspects of code clustering. Being written in Python, it is quite easy to deploy our tool in real life projects irrespective of the operating system. We hope that the our tool will help users execute the typical experiments in code clustering with ease and develop insights, from both research and pragmatic project perspectives.

References

[1] J. Misra, K. M. Annervaz, V. Kaulgud, S. Sengupta, and G. Titus, "Software clustering: Unifying syntactic and semantic features," *Reverse Engineering, Working Conference on*, vol. 0, pp. 113–122, 2012.

[2] J. Misra, K. M. Annervaz, V. S. Kaulgud, S. Sengupta, and G. Titus, "Java source-code clustering: unifying syntactic and semantic features." *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–8, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/sigsoft/sigsoft37.html#MisraKKST12

[3] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 230–243, Mar. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2006.10.017

[4] A. Wiese, V. Ho, and E. Hill, "A comparison of stemmers on source code identifiers for software search," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 496–499. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2011.6080817

[5] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. Beijing: O'Reilly, 2009. [Online]. Available: http://www.nltk.org/book

[6] "Java ast parser." [Online]. Available: http://code.google.com/p/javaparser/

[7] M. Deza and E. Deza, *Encyclopedia of distances*. Springer Verlag, 2009.

[8] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 193–208, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1109/TSE.2006.31

[9] U. Brandes, D. Delling, M. Gaertler, R. Grke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," 2008.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[11] Z. Wen and V. Tzerpos, "An optimal algorithm for mojo distance," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, ser. IWPC '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 227–. [Online]. Available: http://dl.acm.org/citation.cfm?id=851042.857040

[12] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *In Proceedings; IEEE International Conference on Software Maintenance*. IEEE Computer Society Press, 1999, p. pages.

[13] J. Kim, I. Hwang, Y.-H. Kim, and B.-R. Moon, "Genetic approaches for graph partitioning: a survey," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. [Online]. Available: http://doi.acm.org/10.1145/2001576.2001642

[14] A. Kontostathis, "Essential dimensions of latent semantic indexing (lsi)," in *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*. [Online]. Available: http://dx.doi.org/10.1109/HICSS.2007.213

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

ForkSim: Generating Software Forks for Evaluating Cross-Project Similarity Analysis Tools

Jeffrey Svajlenko Chanchal K. Roy
University of Saskatchewan, Canada
{jeff.svajlenko, chanchal.roy}@usask.ca

Slawomir Duszynski
Fraunhofer IESE, Kaiserslautern, Germany
slawomir.duszynski@iese.fraunhofer.de

***Abstract*—Software project forking, that is copying an existing project and developing a new independent project from the copy, occurs frequently in software development. Analysing the code similarities between such software projects is useful as developers can use similarity information to merge the forked systems or migrate them towards a reuse approach. Several techniques for detecting cross-project similarities have been proposed. However, no good benchmark to measure their performance is available. We developed ForkSim, a tool for generating datasets of synthetic software forks with known similarities and differences. This allows the performance of cross-project similarity tools to be measured in terms of recall and precision by comparing their output to the known properties of the generated dataset. These datasets can also be used in controlled experiments to evaluate further aspects of the tools, such as usability or visualization concepts. As a demonstration of our tool, we evaluated the performance of the clone detector NiCad for similarity detection across software forks, which showed the high potential of ForkSim.**

I. Introduction

In software development, similar software projects called software variants can emerge in various ways. Although systematic reuse approaches such as software product lines are known to enable considerable effort savings [1], existing projects are frequently just forked and modified to meet the needs of particular clients and users [2]. These variants typically undergo further parallel development and evolution, and reuse techniques are often not explored until after the variants have matured. This leads to an increased maintenance effort as many tasks are duplicated across the variants.

Maintenance effort can be reduced by merging the forks or by adopting a software reuse approach (e.g., software product lines). Berger et al. [3] report that 50% of industrial software product lines developed by participants of their study were created in an extractive way, i.e., by merging already existing products. This indicates a substantial practical demand for cross-project similarity detection approaches that help software developers discover the similarities between their software variants and support decisions on reuse adoption strategy. Several such approaches have been proposed (e.g., [4] [5]).

A current need is a benchmark for evaluating the performance of tools which detect similarity between software variants. Performance is measured in terms of recall and precision. Recall is the ratio of the similar source code shared between the variants that the tool is able to detect and report. Precision is the ratio of the similar source code elements reported by the tool which are not false positives.

978-1-4673-5739-5/13/$31.00

Measuring recall and precision involves executing the tool for a benchmark dataset (or datasets) and analysing the tool's output. Precision can be easily measured by manually validating all (or typically a random subset) of the tool's output. However, measuring recall requires that all similar code amongst the variants of the dataset be foreknown. This makes it very difficult to use datasets from industry or open source (e.g., BSD Unix forks). Building an oracle by manually investigating the dataset for similar code is, due to time required, essentially impossible for datasets large enough to allow meaningful performance evaluation.

To address these difficulties, and to reduce the amount of required manual validation to a minimum, we developed ForkSim, which uses source code mutation and injection to construct datasets of synthetic software forks. The forks are generated such that their similarities and differences at the source level are known. Recall can then be measured automatically by procedurally comparing a tool's output against the dataset's known similarities. Precision can be measured semi-automatically, as reported similarities which match known similarities or differences in the dataset can be automatically judged as true or false positives, respectively. Only the reported similar code not matching known properties needs to be manually investigated.

The forks generated by ForkSim can be used in any research on detecting, visualizing, or understanding code similarity among software products. The generated forks are a good basis for evaluating automated analysis approaches, as well as for performing controlled experiments to investigate how well the specific tool supports users in understanding similar code. ForkSim is publicly available for download[1].

This paper is organized as follows. Related work is discussed in Section II, and software forking in Section III. Section IV outlines ForkSim's fork generation process, Section V outlines the comprehensiveness of the simulation, and Section VI discusses the quality of the generated forks. Section VII outlines ForkSim's use cases, while Section VIII provides a demonstration of its primary use case: tool performance evaluation. Section IX discusses our plans for future experiments with ForkSim, and Section X our future work on ForkSim. Finally, Section XI concludes the paper.

II. Related and Previous Work

Although automatic tool benchmark construction has been proposed in various problem domains [6], to the best of our

[1] http://homepage.usask.ca/~jes518/forksim.html

knowledge there are no other tools which generate software fork datasets for evaluating cross-project similarity analysis tools, nor is there a reference case (e.g., a set of software forks with known properties) which could be used for tool evaluation. The most related work to ours is a manual validation of cross-project similarity analysis results obtained through clone detection by Mende et al. [5]. However, manual result validation has several drawbacks, as discussed in the introduction. ForkSim is unique in that tool evaluation can be mostly automated for datasets generated by ForkSim, as the similarities and differences between the generated software forks are known.

In previous work, we built a framework which automatically evaluates clone detection tools for intra-project clones [7] by generating a dataset of artificial clones using source code fragment mutation and injection [8] [9] [10]. In this work, we use source mutation and injection at a number of source granularities (function, file, directory) to simulate a forking scenario. The result is a set of software variants with known similarities and differences which can be used to measure the performance of cross-project similarity detection tools.

III. Software Forking

In the forking process, a software system is cloned and the individual forks are evolved in parallel. Development activities may be unique to an individual fork, or shared amongst multiple forks. For example, code may be copied from one fork to another. While forks may share source code, the code may contain fork-specific modifications, and may be positioned differently within the individual forks. A fork may itself be forked into additional forks. Table I provides a taxonomy of the types of source code level development activities performed on forks. These development activities describe how a fork may change with respect to its state at the start of the forking process. This taxonomy is based upon our research and development experience, and discussions with software developers.

Existing code originates from pre-fork development, and new code originates from post-fork development. The development activities occur at various code granularities, including: source directory, source file, function, etc. The result of forking and these further development activities are a set of software variants containing source code in the following three categories: (1) code shared amongst all the forks, (2) code shared amongst a proper subset of the forks, and (3) code unique to a specific fork. ForkSim creates datasets of forks resulting from these development activities, and containing source code in these three categories. It does this by simulating a simple forking scenario.

TABLE I
Taxonomy of Fork Development Activities

ID	Development Activity
DA1	New source code is added.
DA2	Existing source code is removed.
DA3	Existing source code is modified and/or evolved.
DA4	Existing source code is moved.
DA5	Source code is copied from another fork. It may be copied into a different position than in the source fork, and it may be modified and/or evolved independently of the source.
DA6	A fork may itself be forked.

IV. Fork Generation

ForkSim's generation process begins with a base subject system, which is duplicated (forked) a specified number of times. Continued development of the individual forks is simulated by repeatedly injecting source code into the forks. Specifically, ForkSim injects a user specified number of functions, source files, and source directories. Instances of source code of these types are mined from a repository of software systems, which ensures the injected code is realistic and varied.

Each of the chosen functions, files and directories are injected into one or more of the forks. The number and particular forks to inject a source artefact into are selected at random. Injection into a single fork creates code unique to that fork, while injection into a subset of the forks creates code shared amongst those forks. Injection locations are selected randomly, but only amongst the code inherited from the base system, i.e., not inside previously injected code. This prevents the injected code from interacting, which makes the generation process much easier to track and thereby simplifies tool evaluation using the generated dataset. When code is injected into multiple forks, the injection location may be kept uniform or varied, given a specified probability. Forks may share code, but that code may be positioned differently within the individual forks.

Before code is injected into a fork it may be mutated, i.e., automatically modified in a defined way, given a specified probability. This causes code shared by the forks to contain differences, simulating that shared code may be modified or evolved independently for the needs of a particular fork.

Files and directories may be renamed before injection, given a specified probability. While forks may share code at the file and directory level, they may not have the same name. For example, they may have been renamed to match conventions used in the fork.

Each injection operation (injection of a function, a file, or a directory) is logged. This includes recording the forks the code was injected into, the injection locations used, and if and how the code was mutated and/or renamed before injection. A copy of the code and any of its mutants are kept separately and referenced by the log. ForkSim also maintains a copy of the original subject system. From the log and the referenced data the directory, file and function code similarities and differences inherited from the original system and introduced by injection, can be procedurally deduced.

Fig. 1 depicts this generation process. On the left side are the inputs: the subject system, source repository and user-specified generation parameters. The subject system duplicates (forks) are modified by the boxed process, which repeats for each of the source files, directories and functions to be introduced. The figure shows an example of this process, in which a randomly selected function from the source repository is introduced into the forks. ForkSim randomly decided to inject this function into three of the four forks using non-uniform injection locations, and to mutate the function before injection into the latter two forks. On the right side are the outputs: the generated fork dataset and the generation log.

ForkSim supports the generation of Java, C and C# fork datasets. It is implemented in Java 7 (main architecture and simulation logic) and TXL [11] (source code analysis, extraction and mutation). ForkSim's generation parameters are summarized in Table II.

Injection. Directory and file injection is accomplished by copying the directory or file into a randomly selected directory in the fork. In order to prevent injected directories from dominating a fork's directory tree, only leaf directories (those containing no subdirectories) are selected for injection. Function injection is performed by selecting a random source file from the fork, and copying the function's content directly after a randomly selected function in the chosen file. The generation process can be parametrized to only select functions for injection which fall within a specified size range measured in lines before mutation.

Mutation. ForkSim uses mutation operators to mutate code before injection. Fifteen mutation operators, named and described in Table III, were implemented in TXL [11]. Each operator applies a single random modification of its defined type to input code. These mutation operators are based upon a comprehensive taxonomy of the types of edits developers make on cloned code [12], which makes them suitable for simulating how developers modify shared code duplicated between forks.

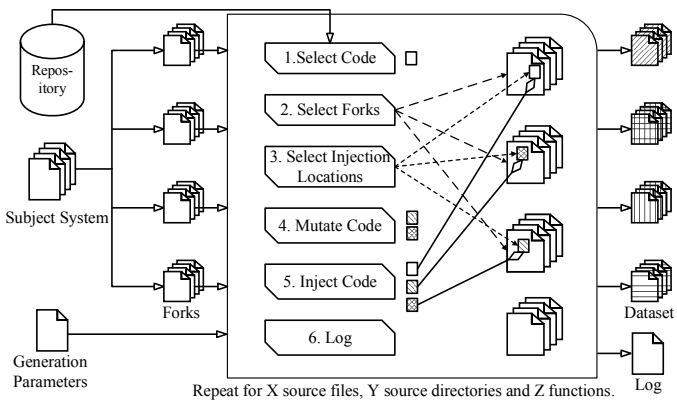


Fig. 1. Fork Generation Process

TABLE II
FORKSIM GENERATION PARAMETERS

Parameter	Description
Subject System	The base system which is forked during the generation process.
Source Repository	A collection of systems from which the source files, directories and functions are mined.
Language	Language of forks to generate (Java, C or C#).
# Forks	Number of forks to generate.
# Files	Number of files to inject.
# Directories	Number of directories to inject.
# Functions	Number of functions to inject.
Function Size	Maximum/Minimum size of functions to inject.
Max Injections	Maximum number of forks to inject a particular function/file/directory into.
Uniform Injection Rate	Probability of uniform injection of a source artefact.
Mutation Rate	Probability of source mutation before injection. Specified separately for function, file and directory injections.
Rename Rate	Probability of renaming before injection.
Max Mutations	Maximum number of mutations to apply to injected code. Specified as a ratio of the code's size in lines.

Files and functions are mutated by applying one of the mutation operators a random number of times before injection. The number of mutations is limited to a specified ratio of the size of the file/function measured in lines after pretty printing. This provides an upper limit on how much simulated development is allowed to occur on a source file or function in a particular fork. Pretty printing (one statement per line, no empty lines, comments removed) the source artefact before measurement ensures the measure is consistently proportional to the amount of actual source code contained. This ratio can be specified separately for files and functions.

A small mutation ratio is recommended (10-15%) as too many changes may cause the variants of a file/function injected into multiple forks to become so dissimilar that they would no longer be a clone. Detection tools would be correct not to report them. ForkSim datasets are only useful if the elements declared as similar are indeed similar.

When directories are injected, each of the source files in the directory may be mutated using the same process as used for injected files. The directory mutation probability parameter defines how likely a file in an injected directory is mutated.

As a principle of mutation analysis, ForkSim does not mix mutation operators. This makes it easier to discover if a similarity analysis tool struggles to detect similar code with particular types of differences. ForkSim cycles through the mutation operators to ensure that each is represented in the generated forks roughly evenly. When it is not possible to apply a given operator to the file or function, another operator is chosen randomly.

Renaming. The probability of a file or directory being renamed before injection is specified separately from that of source code mutation, and both are allowed to occur on the same injection. Renamed source files keep their original extensions.

TABLE III
MUTATION OPERATORS FROM A CODE EDITING TAXONOMY FOR CLONING

ID	Description
mCW_A	Change in whitespace (addition).
mCW_R	Change in whitespace (removal).
mCC_BT	Change in between token (/* */) comments.
mCC_EOL	Change in end of line (//) comments.
mCF_A	Change in formatting (addition of a newline).
mCF_R	Change in formatting (removal of a newline).
mSRI	Systematic renaming of an identifier.
mARI	Arbitrary renaming of a single identifier.
mRL_N	Change in value of a single numeric literal.
mRL_S	Change in value of a single string literal.
mSIL	Small insertion within a line (function parameter).
mSDL	Small deletion within a line (function parameter).
mIL	Insertion of a line.
mDL	Deletion of a line.
mML	Modification of a whole line.

Usage. ForkSim operation is very simple. The user makes a copy of the default generation parameters file, and tweaks it for the dataset they wish to generate. This includes specifying paths to the subject system and source repository to use. Once the parameters file and systems are prepared, the user executes ForkSim and specifies the location of the parameters file and a directory to output the forks and generation log into. Once execution is complete, the forks and generation log are ready for use in experiments involving similarity analysis tools.

V. Simulation of Development Activities

During the generation process, ForkSim simulates all six of the development activities from the forking taxonomy (Table I). The following subsections describe how the code injection scenarios performed during the generation process can be interpreted as the six development activities.

DA1. Any of the code injections can be interpreted as the addition of new code to a fork.

DA2. Code injected into a proper subset of the forks can be interpreted as existing code (pre-fork) which was deleted from the forks it was not injected into.

DA3. Code injected into the forks, with at least one instance mutated, can be interpreted as existing code which was modified/evolved in one or more of the forks, perhaps inconsistently. The code needs not be injected into all of the forks to simulate *DA3*, as the forks missing the code can be interpreted as having lost this shared code due to *DA2*.

DA4. Code injected into the forks, with variation in injection location, can be interpreted as existing code being moved. When the code is not injected into all the forks, the forks missing this existing code can be interpreted as instances of *DA2*.

DA5. Code injected into multiple forks can be interpreted as code implemented in one fork and copied into others. Non-uniform injection, source mutation and renaming simulate that the code may be copied into a different location than in the source, and continued development may occur independently of the source.

DA6. While the generation process creates all of the forks from the same forking point (the base system), the resulting dataset can be interpreted as originating from multiple forking points. Code shared due to injection amongst a subset of the variants can be interpreted as development before a shared forking point, which may not be shared across all the forks. This activity can also be simulated by using the forks as the base system for additional executions of ForkSim.

VI. Discussion

Advantages. The primary advantage of ForkSim is that the user can precisely control the amount and type of similarities and differences among the generated forks. This allows for well-controlled evaluation of tools which analyse forks. Moreover, as the fork generation process is known and logged, the correct and complete information on the actual code similarities and differences between the forks is available. This is not the case when real-world forks are analysed.

Disadvantages. One of the limitations of ForkSim is that the generated variants may have properties that differ from real forks, particularity if aggressive injection settings are used. As injection is a random process, the code-level properties do not represent meaningful development. Also, the distribution of the similar and dissimilar code might differ from real-world forks. However, to the best of our knowledge, there are no systematic studies on the amount and distribution of code similarities and differences in real-world forks. Therefore, we are not able to tune our generation algorithm and its parameters to produce very realistic forks. However, as fork analysis tools likely do not behave differently for less realistic software variants, it is unlikely that this will have a significant effect on tool evaluations using ForkSim-generated fork datasets.

Unknown Similarities. The similarities and differences between the forks inherited from the subject system and intentionally created by injection are exactly known. However, there will be some additional similarities between the forks which are unknown. These include: (1) clones within the original subject system which become similar code within and between the generated forks, (2) unexpected similarity between the functions, files, and directories randomly chosen from the source repository, and (3) unexpected similarity between these chosen source artefacts and the subject system.

Since these similarities are unknown, they are not included in the measurement of a tool's recall for the dataset. However, this is not a disadvantage as we are not interested in evaluating the tools for intra-project similarities. The known similarities are sufficient for measuring cross-project similarities. These similarities, however, must be considered in the measure of a tool's precision.

VII. Use Cases

Cross-Project Similarity Tool Performance Evaluation. ForkSim datasets can be used to measure the recall and precision of tools which detect similarity between software projects. It is especially attuned for tools which focus on similarity detection between software variants (e.g. forks). ForkSim datasets are ideal for this usage scenario as similarities and differences between the generated forks are known. Recall can then be measured automatically, and precision semi-automatically. Recall is evaluated by measuring the ratio of the similar code between the forks, and their relationships, the tool is able to detect and report. The recall measure considers both the similarities created by injection, and the similarities between each of the forks due to the duplication of the base system during the generation process.

How tools report similar code is likely to differ. Therefore, to evaluate recall the dataset's generation log must be mined and converted into the detection tool's output format. This process creates the tool's gold standard, i.e., its ideal and perfect output for the dataset. Recall is then the ratio of the gold standard the tool was able to produce. By building the gold standard procedurally, recall evaluation becomes automatic. The conversion procedure needs only to be written once, and reused for various datasets.

Precision can be evaluated semi-automatically. Any detected similar code which is in the gold standard can be

automatically labelled as true positive. Any reported similarities which match known differences in the dataset can be labelled as false positives. The remaining output requires manual validation to complete the measure of precision. It is sufficient to validate a random subset (large enough to be statistically significant) of the remaining output and to estimate precision from these results.

Tool Usability Study. ForkSim-generated datasets are valuable for performing controlled experiments involving tools which analyse and/or visualize similarities and differences between software variants. The goal of such an experiment can be to measure the level of support for software similarity comprehension the tools provide to their users. The experiment would have the following procedure: first, a dataset of forks with known similarities is generated using ForkSim. Then, the study participants, divided into a few groups, use the tools to analyse the dataset and report their findings. Each user group uses a different tool to solve the same tasks. For example, the participants can be asked a set of questions related to the similarity of the analysed variants, which they should answer by discovering and understanding the similarities using the given tool. The tasks should be designed to evaluate a specific aspect of the tools, e.g., their usability or the appropriateness of the used similarity visualizations. By checking the correctness of the answers and recording the amount of needed effort and/or time, user group performance is quantitatively measured. In this way, the effect of using the different tools is quantified, and the tools can be compared regarding the properties targeted by the tasks, such as tool usability. As discussed in Section VI, ForkSim-generated datasets have both advantages (e.g., precise control of similarity) and disadvantages (e.g., injection of non-related code) as compared to real datasets. Hence, the use of generated datasets is suitable in situations where the stated disadvantages do not influence the experimental goal.

Adaptations. For the purpose of clone management, detecting and studying clone genealogies is another important research topic, and there have been a few genealogy detectors (e.g., [13]). The technology used in ForkSim can easily be adapted to generate software versions rather than software variants for evaluating genealogy detectors' performance. Such an adaptation could also be used to evaluate clone ranking algorithms [14], which use multiple versions of a software system to produce a clone ranking.

VIII. Evaluation

As a demonstration of ForkSim's primary use case, tool evaluation, we evaluated NiCad's performance for similarity detection between software variants. NiCad [15], [16] is one of the state of the art near-miss software clone [7] detectors. While it is designed for single systems, it can be used to detect similarity between forks by executing it for the entire dataset and trimming the intra-project clone results from its output. To evaluate NiCad, we generated a ForkSim dataset of five Java forks. We used JHotDraw54b1 as the subject system and Java6 as the source repository. The generation parameters used are listed in Table IV. The NiCad clone detector is capable of detecting function and block granularity near-miss clones. It uses TXL to parse source elements of these granularities from an input system, and uses a diff-like algorithm to detect clones after these source elements have been normalized to remove irrelevant differences (e.g., formatting, comments, whitespace, identifier names, and more). For use in this experiment, we extended NiCad to support the detection of clones at the file granularity.

Using NiCad, we detected the file and function clones in the dataset. NiCad was set to detect clones 3-5000 lines long, with at most 30% difference. It was configured to pretty print the source, blind rename the identifiers, and normalize the literal values in the dataset before detection. The clones were collected both in clone pair (pairs of similar files or functions) and clone class (set of similar files or functions) formats. Overall, NiCad found 363 file clone classes (16,553 pairs) and 1,831 function clone classes (2,198,636 pairs).

To evaluate NiCad's recall, we converted the known similarities between the forks into file and function clone classes. Each file injected into multiple forks was converted into a file clone class, as were the files contained in directories injected into multiple forks. File clone classes were created for each of the files the forks inherited from the subject system, with the files modified due to function injection trimmed from these classes. Each function injected into multiple forks was converted into a function clone class. Finally, a function clone class was created for each function the forks inherited from the subject system. These clone classes were also converted to clone pair format.

NiCad's recall performance is summarized in Table V. Recall was measured per clone granularity (file or function), and per origin of similarity (file/directory/function injection or original subject system files). As can be seen, NiCad had 100% recall for all sources of file clone classes, and 98-99% for function clone classes. If we consider clone pairs instead of clone classes, we see that the function clone detection is marginally better (+0.1%). These are very promising results for NiCad as a fork similarity analysis tool. These results are specific to the dataset's generation parameters. In future, we plan to evaluate NiCad's recall performance for many datasets with varied parameters; for example, with larger and smaller max mutation values.

Due to time constraints, we did not perform a full precision analysis for this experiment. However, NiCad is known to have high precision [17]. Using known similarities, we were able to

TABLE IV
ForkSim Generation Parameters: NiCad Case Study

Parameter	Value
Subject System	JHotDraw54b1
Source Repository	Java6
Language	Java
# Forks	5
# Files	100
# Directories	25
# Functions	100
Function Size	20-100 lines
Max Injections	5
Uniform Injection Rate	50%
Mutation Rate	files: 50%, directories(files): 50%, functions: 50%
Rename Rate	files: 50%, directories: 50%
Max Mutations	15% of size in lines

TABLE V
NICAD CASE STUDY RECALL RESULTS

Type	File Injections	Directory Injections	Function Injections	Original Files
File Clone Class	100% (41/41)	100% (117/117)	-	100% (260/260)
Function Clone Class	-	-	98.7% (75/76)	99.4% (2869/2886)
Function Clone Pair	-	-	98.8% (332/336)	99.5% (28708/28860)

validate 20.7% of NiCad's reported file clone pairs, but only 1.46% of its reported function clone pairs. NiCad is reporting a large amount of cloned code beyond that of the known similarities. Part of this is due to unknown similarities arising from clones within the original subject system. However, a large fraction of this is due to the NiCad clone size settings used. A minimum clone size of 3 lines was required to ensure that all cloned functions were detected. However, small standard functions such as getters and setters are very similar after normalization, which was a source of a large number of these clone pairs. Likewise, interfaces and simple classes are likely to be detected as similar after identifier normalization. For practical usage, these small similarities would likely be filtered out in preference of the larger similarities. In summary, NiCad has very good detection performance of similarities between forks, but the quantity of output would make its usage difficult. A post-processing step needs to be added to extract the most useful and important similarity features from its output.

IX. FUTURE EXPERIMENTS

We plan to use ForkSim to evaluate our tools for similarity detection between software variants. We will use a variety ForkSim datasets to evaluate the Fraunhofer Variant Analysis [4] tool's recall and precision. This tool locates and visualizes similarities between software variants with the aim of supporting software product line adoption. We also plan to perform a more in-depth analysis of NiCad's, SimCad's [18], another state of the art near-miss clone detector, and other clone detectors' performance in this domain. These experiments are prime examples of ForkSim's featured use cases, and demonstrate how other researchers and practitioners might utilize ForkSim.

X. FUTURE WORK

The following are additional features we plan to add in future work: **(1)** The option to allow the mixing of mutation operators during file and function mutation. **(2)** Expand directory injection beyond leaf directories, with optional constraints on hierarchy size (breadth and depth) of the injected directory. **(3)** The splitting of files or directories before injection, given a defined probability. **(4)** An alternative mode, in which ForkSim injects into a set of disparate systems rather than duplicating (forking) a single system. This would allow the generation of systems with shared code which were not the result of forking, but of large scale code re-use by copy and paste. **(5)** Using tools we have tested and evaluated with ForkSim datasets, we plan to study the similarity patterns found in real forked systems. We will use our findings to enhance ForkSim and tune its parameters to produce datasets which better mimic real-world forks.

XI. CONCLUSION

In this paper, we have introduced ForkSim, a tool for generating customizable datasets of synthetic forks with known similarities and differences. These datasets can be used in any research on the detection, visualization, and comprehension of code similarity amongst software variants. ForkSim datasets allow similarity detection tools to be evaluated in terms of recall (automatically) and precision (semi-automatically), and can be useful in experiments aiming at evaluating the usability and visualization of similarity tools. We demonstrated ForkSim using a case study evaluating NiCad's cross-project similarity detection for a set of five ForkSim-generated Java forks.

REFERENCES

[1] P. Clements and L. Northrop, *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[2] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," *Proc. CSMR*, pp. 25–34, 2013.

[3] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *Proc. VaMoS*, 2013, pp. 7:1–7:8.

[4] S. Duszynski, J. Knodel, and M. Becker, "Analyzing the source code of multiple software variants for reuse potential," in *Proc. WCRE*, 2011, pp. 303–307.

[5] T. Mende, R. Koschke, and F. Beckwermert, "An evaluation of code similarity identification for the grow-and-prune model," *Journal of Software: Evolution and Process*, vol. 21, no. 2, pp. 143–169, Mar. 2009.

[6] D. Lo and S.-C. Khoo, "Quark: Empirical assessment of automaton-based specification miners," in *Proc. WCRE*, 2006, pp. 51–60.

[7] C. K. Roy, "Detection and analysis of near-miss software clones," in *Proc. ICSM*, 2009, pp. 447–450.

[8] J. Svajlenko, C. K. Roy, and J. R. Cordy, "A mutation analysis based benchmarking framework for clone detectors," in *Proc. IWSC*, May 2013, pp. 8–9.

[9] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *Proc. ICSTW*, 2009, pp. 157–166.

[10] ——, "Towards a mutation-based automatic framework for evaluating code clone detection tools," in *Proc. C3S2E*, 2008, pp. 137–140.

[11] J. R. Cordy, "The txl source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, Aug. 2006.

[12] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection, techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, pp. 470–495, 2009.

[13] R. K. Saha, C. K. Roy, and K. A. Schneider, "An automatic framework for extracting and classifying near-miss clone genealogies," in *Proc. ICSM*, 2011, pp. 293–302.

[14] M. F. Zibran and C. K. Roy, "A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring," in *Proc. SCAM*, 2011, pp. 105–114.

[15] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. ICPC*, 2008, pp. 172–181.

[16] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *Proc. ICPC*, 2011, pp. 219–220.

[17] C. K. Roy and J. R. Cordy, "Near-miss function clones in open source software: an empirical study," *Journal of Software: Evolution and Process*, vol. 22, no. 3, pp. 165–189, 2010.

[18] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, "On the effectiveness of simhash for detecting near-miss clones in large scale software systems," in *Proc. WCRE*, 2011, pp. 13–22.

Diego Mendez, Benoit Baudry and Martin Monperrus. Empirical Evidence of Large-Scale Diversity in API Usage of Object=Oriented Software

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

Aspectual Source Code Analysis with GASR

Johan Fabry
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile
Santiago, Chile
http://pleiad.cl

Coen De Roover
and Viviane Jonckers
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
http://soft.vub.ac.be

***Abstract*—To be able to modularize crosscutting concerns, aspects introduce new programming language features, often in a new language, with a specific syntax. These new features lead to new needs for source code analysis tools, resulting in the requirement for a general-purpose aspectual source code analysis tool. Ignoring this requirement has led to a nontrivial duplication of effort in the aspect-oriented software development community. This is because all code analysis efforts that we are aware of have either built ad-hoc analysis tools or were performed manually. In this paper we present GASR: a source code analysis tool in the tradition of logic program querying that reasons over ASPECTJ source code. By hooking into the IDE plugins for ASPECTJ, GASR provides a library of predicates that can be used to query aspectual source code. We demonstrate the use of GASR by automating the recognition of a number of previously identified aspectual source code assumptions. We then detect assumption instances on two well-known case studies that were manually investigated in the earlier work. In addition to finding the already known aspect assumptions, GASR encounters assumption instances that were overlooked before.**

Keywords*—*Aspect Oriented Programming, Logic Program Querying, Aspectual Assumptions

I. INTRODUCTION

Aspects are a means to modularize *cross-cutting concerns*: concerns whose implementation is spread throughout different modules of the system under construction. Aspects are a new kind of module that encapsulate, in addition to their behavior, when this behavior needs to be invoked, *i.e.,* also define a kind of implicit invocation of their behavior.

To perform Aspect-Oriented Programming, new programming languages have been proposed that are usually extensions of existing OOP languages. These extensions then consist of language features that allow for the specification of these new modules, and most importantly their implicit invocation conditions, known as *pointcuts*. As a result of this, existing source code analysis tools for these OOP languages are incapable to correctly treat these aspects in their reasoning. Firstly existing analyses may be incorrect and, secondly, analyses that specifically consider the aspectual properties of the source code are absent. Considering the first case: as aspects modify the control flow of the program, source code analysis should take these changes into account when reasoning over properties of the code, where appropriate. As for the second case, the extensions made by aspect languages are usually nontrivial. This creates an entirely new class of analyses that take into account the aspectual nature of the code and how these features are used and interact (*e.g.,* [1], [2], [3], [4], [5], [6], [7], [8], [9]). To the best of our knowledge, all of these kinds of analyses have been made on an ad-hoc basis, customized specifically to the analysis task being performed. As a result there has been a considerable duplication in development effort of these analyses. Moreover these are not customizable to the actual software under analysis, *e.g.,* to automatically remove one kind of known false positives from the results.

978-1-4673-5739-5/13/$31.00

We state that there is a need for a general-purpose source code reasoner for aspects. It should ease the definition of multiple kinds of source-code analysis over aspect-oriented source code and also be tailorable to the task at hand by the user. To address this need we built GASR and we present it in this paper.

This paper contains the following contributions:

- It argues for the need for a general-purpose source code analysis tool that is aware of aspects.
- It presents the logic program querying tool GASR, the first such analysis tool, and discusses its implementation along with its library of logical predicates.
- It shows how GASR can be used to automatically verify a subset of previously published inter-aspect assumptions [10], implementing part of the future work of that publication.

The structure of this text is as follows: next we provide the problem statement of the paper, arguing for the need for GASR. In Section III we give an overview of related work, showing that existing analyses have been ad-hoc. We then present GASR in Section IV, discussing its implementation and a selection of its library of logical predicates. This is followed, in Section V, by an illustration of the usefulness of GASR by realizing detection of inter-aspect assumptions, as previously identified in [10]. The paper then briefly discusses threats to validity in Section VI before providing conclusions and avenues for possible future work.

II. PROBLEM STATEMENT

To enable the modular specification of crosscutting concerns, aspects encapsulate both their behavior as well as the invocation conditions for this behavior. This gives rise to new language features and terminology, which in turn requires new features for a source code reasoner.

A. Terminology and Language Features

Broadly put, an aspect contains two parts: its behavior, specified in a number of *advice*, and the invocation specifications for this advice, denoted in *pointcuts*. Advice are linked to pointcuts, and whenever a pointcut matches the linked advice is invoked. Conceptually, to match pointcuts each execution step of the software is reified as a *join point* and pointcuts are predicates over join points. The work of performing join point reificiation, passing them to all pointcuts, and running the associated advice if a pointcut matches is performed by the *aspect weaver*. Implementation strategies for aspect weavers vary from source-code preprocessing to aspect-aware virtual machines. A last item of terminology is the *join point shadow* for a join point: the piece of source code whose execution produced that join point.

We now show different language features that the prototypical aspect-oriented language places at the programmers' disposal, as an indication towards the possible complexity of aspectual source code. The example language is ASPECTJ [11], arguably the best-known and most-used aspect language. ASPECTJ is an extension of Java that introduces aspect features using a specific syntax. We now briefly touch on the different aspectual language features of ASPECTJ, starting with aspects: Aspect declarations are similar to class declarations and are declared using the aspect keyword. Aspects can contain methods and fields, but only one zero-argument constructor. The latter is because aspects cannot be manually instantiated, the weaver performs this when needed (typically aspects are singletons). Aspects may be abstract, extend classes and abstract aspects, and implement interfaces.

Pointcuts in ASPECTJ are a new sort of member declaration that use the pointcut keyword, and have the standard visibility and inheritance semantics. Pointcuts have a body, unless they are declared as abstract. Abstract pointcuts can only be contained in an abstract aspect. The body of a pointcut is a logical combination of pointcut expressions or a primitive pointcut expression. Primitive pointcut expressions firstly specify the kind of pointcut: an execution of a method, a call of a method, getting or setting a field, and so on. Secondly, they provide a pattern that may match on that kind of execution step of the software, *e.g.,* a signature of a method. In this pattern wildcards may be used to generalize over names as well as types.

An example of a quite drastic pointcut that uses a pattern is below: a pointcut named allFoo that matches on the method calls of all methods of the class Foo, irrespective of the return type and number of parameters.

```
1public pointcut allFoo() : call(* Foo.*(..));
```

Advice are similar to methods in that they declare a body of code and have parameters. They differ firstly in that they do not have a name, but instead declare that they need to be invoked before, after, or instead of (around) the join point. They link to a pointcut by providing the pointcut name, or a pointcut body (known as using an anonymous pointcut).

Lastly, aspects may also modify the type hierarchy and add *inter-type declarations*. In the former the aspect declares that given classes or aspects extend or implement other classes or interfaces. In the latter the aspect adds fields or methods to other classes, similar to what is allowed in Open Classes [12].

B. Classic Example: Aspect Reentrancy

As a first, brief, example of a concrete need for aspect-specific source code reasoning we now present a classic example of an ASPECTJ antipattern regarding reentrancy. We include it here as it is important, yet simple enough to be briefly explained. Consider the following as a token of its importance: the ASPECTJ documentation 'pitfalls' section[1] contains just this one example, and no other.

```
1aspect Boom {
2   before(): call(* *(..)) {
3     System.out.println("before"); }}
```

The aspect above declares one advice, with a pointcut body that matches *on all method calls in the program*. The behavior of the advice is to make a method call to the System.out.println method. The pointcut matches on all method calls, hence also this call, hence before the method is called the advice body is again executed, leading to an infinite loop.

The antipattern in the above example can be easily detected by an aspect-aware source code reasoner: there is a possibility for infinite application of an advice when the join point shadows of the associated pointcut are contained in this advice.

C. Problem: A New Reasoning Need

If we consider aspects as simply a means to achieve behavior subject to implicit invocation with implicit announcement [13], it may seem that the need for source code reasoning over aspects is simple. Since in this view aspects essentially are for altering the control flow of the running application, existing source code reasoners just need to be extended to take this control flow into account.

Aspects however go beyond the above as they introduce multiple aspectual language features that interact with the non-aspectual language features as well as among themselves. An example of the former is that aspects may change the class hierarchy of the program. As an example of the latter consider a pointcut named abstractpc, defined as an abstract pointcut in a root aspect Root and also used by an advice of Root. abstractpc may be concretized in a child-aspect Child of Root. It may also be concretized again in an aspect Grandchild that is a child of Child, *i.e.,* a grandchild of Root. The definition of the actual pointcut that is used for the advice in Root is the lowest in the hierarchy [10], *i.e.,* the re-concretization in Grandchild.

As a result, in addition to the classic example of Sect. II-B, many possible issues in aspectual code have been separately identified. We provide three examples. First are aspects assuming specific properties of other aspects to be present [10], which we will discuss in more detail in Section V. Second is the problem of pointcuts that are slightly or subtly incorrect [1], as a result these fail to match the intended join points, or match unintended join points. Third is the fragility of pointcuts when the software evolves [3], [4], in this case pointcuts end up being broken due to changes in the program that were made due to its evolution. We find it remarkable that for these three examples no single source code reasoner can yet be used to detect all of these issues such that they can be revealed using, *e.g.,* a bad smells detection tool.

[1] http://www.eclipse.org/aspectj/doc/next/progguide/pitfalls.html

Also, multiple aspectual design patterns have been presented [14], [15], yet no mining of these patterns with a source code reasoner have been documented. A well-known example is the Wormhole [15]: an aspect intervenes in one part of the control flow to store the value of a specific parameter or variable, and in a second part retrieves this value and injects it back in the control flow. It is as if the value passed through a wormhole that lies between both parts. Given that a pattern is a template, there may be various variations on this template, and various ways in which these are instantiated. Hence an analysis tool to discover such patterns or to verify their correct use, *e.g.,* if the stored value is modified before it is injected, would need to be tailorable to the case at hand.

From the above, we conclude that there is a need for multiple kinds of source-code analysis over aspect-oriented software that are general enough such that they can be used for multiple kinds of analysis and moreover are adaptable such that they can be tailored to the task at hand. In other words, we need a general-purpose source code reasoner for aspects. With this reasoner we would then be able to, *e.g.,* automatically identify aspectual assumptions in code, write a bad smells tool that can reveal errors as in Sect. II-B, or detect incorrect use of the Wormhole pattern.

III. Related Work

To the best of our knowledge, there is no general-purpose aspect source code analysis tool. Directly related work consists of specific ad-hoc analyses made, and indirectly related is work on code comprehension of aspectual source code.

Getting pointcuts correct can be a hard task [1], and as a result of this, pointcuts have been the focus of various tracks of research that include code reasoning. Notable early work is on PointcutDoctor [1], a tool that provides special-purpose reasoning over pointcuts to establish near matches of pointcuts as well as the reasons why a given shadow matches, or does not match a specific pointcut. Related to this is the fragility of pointcuts, as mentioned above. The most recent work is on pointcut rejuvenation [2]. New code that is added as the software evolves may also need to be captured by the existing pointcuts, *i.e.,* they need to be changed. A custom analysis is developed that suggests changes to pointcuts when needed. Earlier work in this area [3], [4] also used custom analyses.

Yet reasoning about aspectual source code is not limited to pointcuts only. For example, ITDVisualizer [5] is a tool that supplies an analysis of the impact of intertype declarations. It shows how they impact method lookup, and identifies how code entities are shadowed by intertype declarations. XFindBugs [6] is a tool that uses static analysis to find potential bugs in aspectual source code. It defines a catalog of multiple bug patterns for aspect-oriented features, and implements a set of bug detectors on top of the FindBugs analysis framework[2]. Last but not least, the work on the Ajana analysis framework [7] for source-code-level interprocedural dataflow analysis yields a control- and data-flow program representation for aspectual source code. Considering this representation it proposes an object effect analysis and a dependency analysis. Again all of the above tools use a custom reasoner to provide the analysis.

[2]http://findbugs.sourceforge.net/

Complementary to the above, code comprehension tools for aspectual source code also include some form of ad-hoc reasoning to be able to display their specific comprehension aids. We highlight two such tools: the AJDT and AspectMaps.

The AspectJ Development Toolkit (AJDT) [8] is arguably the most mature, feature-rich and best known tool suite for AOP. It consists of a set of plug-ins to the Eclipse IDE that add code comprehension features, amongst others. It provides a "Cross References" view that, when editing an aspect or class, shows a summary of the join point shadows or advice that apply, respectively. In the code editor, at each join point shadow, gutter markers are present that reveal information about the advice. AJDT also provides for a visualization of the source code, but this feature has been superseded by other aspectual visualizations, the most recent of which is AspectMaps [9], [16]. AspectMaps is a visualization tool that shows where in the code aspects apply. Of all aspect visualizations, AspectMaps shows the most information about the source code [9]. Moreover, by using a selective structural zoom, it ensures a scalable visualization from package level all the way down to method level. At this finest granularity it shows exactly where advice apply, the order of advice execution at one shadow and whether the advice has any runtime invocation conditions.

A common thread in all the above work is that the required source code analysis is provided ad-hoc, entailing a significant duplication of effort. If a general-purpose aspect-oriented source code reasoner would have existed, this duplication of effort might have been avoided.

IV. Querying AspectJ Programs using Gasr

We introduce Gasr (General-purpose Aspectual Source code Reasoner) as a tool for answering user-specified questions about the structure as well as the behavior of an aspect-oriented program. Examples range from "*which pointcut definitons are overridden in a subtype?*" over "*which pointcuts have a join point shadow in an advice?*" to "*can these advices be executed consecutively?*". Such questions have to be specified as a logic query of which the conditions quantify over the program's source code. The expressiveness of the logic paradigm has been shown to facilitate specifying the characteristics of sought after code. Once specified in a logic program query, retrieving source code elements that exhibit these characteristics is left to the querying tool. This relieves users of having to implement an imperative search themselves. As such, Gasr is a tool in the tradition of logic program querying. Other examples include CodeQuest [17], PQL [18] and Soul [19].

Gasr owes its query language to the core.logic[3] port to Clojure of miniKanren [20], and its IDE integration to the Ekeko[4] Eclipse plugin. The latter enables launching and scheduling program queries, as well as inspecting the solutions to a query and associating warning markers with them — actually building upon our earlier Eclipse plugin suite for program querying [19], [21].

[3]https://github.com/clojure/core.logic

[4]https://github.com/cderoove/damp.ekeko/

A. Launching Program Queries

Queries can be launched from a read-eval-print loop using the `ekeko*` special form. It takes a vector of logic variables, each denoted by a starting question mark, as its first argument and this is then followed by a sequence of logic goals:

```
1 (ekeko* [?x ?y]
2   (contains [1 2] ?x)
3   (contains [3 4] ?y))
```

The binary predicate `contains/2`, used by both goals, holds if its first argument is a collection that contains the second argument. Solutions to a query consist of the different bindings for its variables such that all logic goals succeed. Internally, the logic engine performs an exploration of all possible results, using backtracking to yield the different bindings for logic variables. The four solutions to the above query consist of bindings `[?x ?y]` such that `?x` is an element of vector `[1 2]` and `?y` is an element of vector `[3 4]`: `([1 3] [1 4] [2 3] [2 4])`.

Logic variables have to be introduced explicitly into a lexical scope. Above, the `ekeko*` special form introduced two variables into the scope of its logic conditions. Additional variables can be introduced through the `fresh` special form:

```
1 (ekeko* [?x]
2   (differs ?x 4)
3   (fresh [?y]
4     (equals ?y ?x)
5     (contains [3 4] ?y)))
```

The above query has but one solution: `([3])`. Indeed, `3` is the only binding for `?x` such that all goals suceed. The `differs/2` goal on line 2 imposes a disequality constraint such that any binding for `?x` has to differ from `4`. The `equals/2` goal on line 4 requires `?x` and the newly introduced `?y` to unify.

Finally, new predicates can be defined as regular Clojure functions that return a logic goal. As such, the aforementioned special forms give rise to an *embedding* of logic programming in a functional language.

```
1 (defn contains+ [?c ?e]
2   (conde [(contains ?c ?e)]
3          [(fresh [?x]
4            (contains ?c ?x)
5            (contains+ ?x ?e)])))
```

Here, the special form `conde` returns a goal that is the disjunction of one or more goals. The newly defined predicate `contains+` therefore succeeds for `?e` that reside at an arbitrary depth within a collection `?c`.

Note that an idiomatic Prolog definition of the above would consist of two rules that define the same predicate: one for the base case and one for the recursive case, thus creating an implicit choice point. By relying on function definition, the above implementation has to make such choice points explicit.

B. The Predicate Library of GASR

To enable querying ASPECTJ programs, we have developed a library of predicates that can be used in EKEKO queries. For instance, solutions to the following query correspond to instances of the aspect reentrancy example described in Section II-B:

```
1 (ekeko* [?aspect ?advice]
2   (fresh [?shadow]
3     (aspect-advice ?aspect ?advice)
4     (advice-shadow ?advice ?shadow)
5     (shadow-enclosing ?shadow ?advice)))
```

Upon backtracking, the goal on line 3 successively binds `?advice` with each advice of an aspect `?aspect` —which is also bound successively to each aspect known to the ASPECTJ weaver. The goal on line 4 binds `?shadow` to one of the join point shadows of this advice, while the goal on line 5 requires `?advice` to unify with the immediately enclosing source code entity of `?shadow`. Hence, `?advice` will be bound to an advice that advices itself, *i.e.,* a possible infinite loop. Note that, by convention, the names of predicates that reify an n-ary relation consist of n components separated by a –, each describing an element of the relation. Also, vertical bars | separate words within the description of a single component.

The predicates used in the above query concern the structure of the woven ASPECTJ program. In contrast, the predicates below concern possible behavior of the program at run-time. Its solutions correspond to possible instances of the wormhole pattern described in Section II-C:

```
1 (ekeko* [?aspect ?advice|entry ?advice|exit ?field]
2   (aspect-advice ?aspect ?advice|entry)
3   (type-field ?aspect ?field)
4   (advice|writes-field ?advice|entry ?field)
5   (differs ?advice|exit ?advice|entry)
6   (aspect-advice ?aspect ?advice|exit)
7   (advice|reads-field ?advice|exit ?field)
8   (advice-reachable|advice ?advice|entry ?advice|exit))
```

The first goal binds `?advice|entry` to an advice that will serve as the entry point of the wormhole `?aspect`. Lines 3–4 therefore ensure that this advice writes to a `?field` defined in the same aspect. Lines 5–6 require this aspect to feature a different `?advice|exit` that will serve as the exit point of the wormhole. As such, the exit advice has to read from the field written to by the entry advice (line 7). Note how multiple occurrences of a logic variable link these goals together. The final goal conservatively ensures that there might be an execution of the woven program in which `?advice|exit` is executed after `advice|entry`.

We have developed a comprehensive library of logic predicates, which we do not discuss in full here. Instead, Table I and Table II list representative predicates that reify structural resp. behavioral relations between ASPECTJ source code entities. We refer to the online documentation[5] for an overview of the complete predicate library. The remainder of this section discusses the highlights of its implementation.

1) Predicates Reifying Structural Relations: The predicates listed in Table I reify the structural relations between the source code entities of an ASPECTJ program (*e.g.,* types and their members, aspects and their pointcut definitions, advices and their shadows). To this end, their implementation consults the domain model maintained by the ASPECTJ weaver.

EKEKO supports calling out to Java from within a logic goal. This obviates the need to convert the weaver's domain objects to logic facts. Instead, they are kept as instances

[5]https://github.com/cderoove/damp.ekeko.aspectj

Predicate	Reified Relation
`(type ?type)`	Of all types known to the weaver (*i.e.,* aspects, classes, interfaces, enums, *etc*).
`(type-declaredsuper ?type ?super)`	Between a type and its direct declared superclass or superaspect.
`(type-declaredinterface ?type ?interface)`	Between a type and one of the interfaces it declares to be implementing or extending directly.
`(type-super+ ?type ?super)`	Between a type and one of its direct or indirect super types (classes, aspects as well as interfaces), including those that stem from an intertype declaration.
`(type-method ?type ?method)`	Between a type and one of its declared methods.
`(type-method+ ?type ?method)`	Between a type and one of its declared or inherited methods. Does not include methods stemming from intertype declarations.
`(aspect ?aspect)`	Of all aspects known to the weaver. Subrelation of `type/2`.
`(aspect-pointcutdefinition ?aspect ?pointcutdefinition)`	Between an aspect and one of its declared pointcut definitions.
`(aspect-advice ?aspect ?advice)`	Between an aspect and one of its declared advice.
`(aspect-intertype ?aspect ?intertype)`	Between an aspect and one of its intertype member declarations.
`(aspect-declare ?aspect ?declare)`	Between an aspect and one of its `declare` declarations (*e.g.,* parents, precedence).
`(pointcutdefinition-pointcut ?pointcutdefinition ?pointcut)`	Between a non-abstract pointcut definition and its pointcut.
`(pointcutdefinition-name ?pointcutdefinition ?name)`	Between a pointcut definition and its name.
`(pointcutdefinition|abstract ?pointcutdefinition)`	Of abstract pointcut definitions. Sub-relation of `pointcutdefinition/1`.
`(advice|before ?advice)`	Of `before` advices. Sub-relation of `advice/1`.
`(advice-pointcut ?advice ?pointcut)`	Between an advice and its pointcut. The latter either an anonymous pointcut, or a pointcut definition.
`(advice-pointcutdefinition ?advice ?pointcutdefinition)`	Between an advice and the concrete pointcutdefinition its name resolves to (*i.e.,* overrides of possibly abstract pointcutdefinitions are taken into account).
`(advice-shadow ?advice ?shadow)`	Between an advice and one of its join point shadows.
`(shadow-enclosing ?shadow ?enclosing)`	Between a shadow and its immediately enclosing entity or the entity itself for entity shadows. This entity can be a class, aspect, enum, method, intertype method, advice, *etc* ...
`(shadow-ancesor|type ?shadow ?type)`	Between a shadow and its first enclosing type entity (*e.g.,* aspect, class, enum).
`(intertype-member-target ?intertype ?member ?type)`	Between an intertype declaration, the member (*i.e.,* field, method or constructor) it declares, and the type to which this member is added.
`(declare|parents ?declare)`	Of `declare parents` declarations. Subrelation of `declare/1`.
`(declare|parents-target-parent ?declare ?target ?parent)`	Between a `declare parents` declaration, one of the target types matching its pattern and the corresponding super type.
`(declare|precedence ?declare)`	Of `declare precedence` declarations. Subrelation of `declare/1`.
`(aspect|dominates-aspect ?daspect ?saspect)`	Of actual domination relations between aspects that result from `declare precedence` declarations.

TABLE I. REPRESENTATIVE PREDICATES CONCERNING STRUCTURE.

Predicate	Reified Relation
`(advice|reads-field ?advice ?field)`	Between an advice and one of the fields it reads from.
`(advice|writes-field ?advice ?field)`	Between an advice and one of the fields it writes to.
`(advice-reachable|advice ?advice1 ?advice2)`	Between an advice and another advice such that the latter may be executed after the former. Concretely, this is the relation of two successive advices on a path through the inter-procedural control flow graph of the woven program.
`(field-soot|field ?field ?soot)`	Between a field and the SOOT field that represents its implementation.
`(advice-soot|method ?advice ?soot)`	Between an advice and the SOOT method that represents its implementation.
`(intertype|method-soot|method ?itmethod ?soot)`	Between a method declared by an intertype declaration and the SOOT method that represents its implementation.
`(soot|method-soot|unit ?method ?unit)`	Between a SOOT method and one of the units in its body. These correspond to instructions in SOOT's JIMPLE intermediate representation [22] of the woven program.
`(soot|unit|reads-soot|valuebox ?unit ?value)`	Between a SOOT unit and one of the values (*e.g.,* parameters, field references, expressions...) it reads from.
`(soot|unit|writes-soot|valuebox ?unit ?value)`	Between a SOOT unit and one of the values it writes to.
`(icfg|main-start ?icfg ?icfg|start)`	Between the inter-procedural control flow graph of the woven program and its starting node.
`(icfgnode-unit ?node ?unit)`	Between a node of the inter-procedural control flow graph and a SOOT unit.
`(icfgnode-method ?node ?method)`	Between a node of the inter-procedural control flow graph and the SOOT method in which it resides.
`(icfgnode-stack?node ?stack)`	Between a node of the inter-procedural control flow graph and the (finite) configuration of the call stack at the time it was encountered during a traversal.
`(path ?icfg ?start ?end [`$v_1 \ldots v_n$`] `$q_1 \ldots q_n$`)`	Of inter-procedural control flow graphs `?icfg` in which there exists a path from `?start` till `?end` that is of the form described by the regular path expression $q_1 \ldots q_n$. Here q is one of the regular path primitives provided by the QWAL library [23]: `q=>` skips a single node, `q=>*` skips zero or more nodes, and `q=>+` skips one or more nodes on the path. Primitive `qcurrent` evaluates logic goals against the current node on the path, possibly involving one of the $v_1 \ldots v_n$ logic variables.

TABLE II. REPRESENTATIVE PREDICATES CONCERNING BEHAVIOR.

of various org.aspectj.weaver classes. The binary predicate `aspect-pointcutdefinition/2`, *e.g.,* is as follows:

```
1 (defn aspect-pointcutdefinition [?aspect ?pcdef]
2   (fresh [?pcdefs]
3     (aspect ?aspect)
4     (equals ?pcdefs (.getDeclaredPointcuts ?aspect))
5     (contains ?pcdefs ?pcdef)))
```

The predicate reifies the relation between an aspect and one of its own, non-inherited pointcut definitions. The goal on line 3 ensures that `?aspect` is bound to the weaver's representation of an aspect (*i.e.,* an instance of ResolvedType). This enables the goal on line 4 to unify `?pcdefs` with the result returned by method getDeclaredPointcuts() on the binding of `?aspect`. Upon backtracking, the goal on line 5 will therefore successively unify `?pcdef` with each of the elements of the returned collection of ResolvedPointcutDefinition instances.

2) Predicates Reifying Behavioral Relations: The predicates listed in Table II reify control flow and data flow relations between the source code entities of the *woven* ASPECTJ program. While the former concerns the order in which instructions may be executed at run-time, the latter concerns the values these instructions may operate upon.

The predicates at the top of Table II reify behavioral relations between elements that stem from the weaver's domain model. These can be combined with the structural predicates of Table I. For example, solutions to the following consist of an advice and a type of which the advice modifies a field:

```
1 (ekeko* [?advice ?type]
2   (fresh [?field]
3     (advice|writes-field ?advice ?field)
4     (type-field ?type ?field)))
```

These predicates are implemented themselves in terms of predicates that quantify over static analysis results provided by the SOOT [22] analysis framework (third row in Table II) and predicates that link both sources of information together (second row in Table II). For instance, the binary predicate `advice|writes-field/2` is implemented as follows:

```
1 (defn advice|writes-field [?advice ?field]
2   (fresh [?soot|method ?soot|field ?soot|unit
3           ?vbox ?value]
4     (advice-soot|method ?advice ?soot|method)
5     (field-soot|field ?field ?soot|field)
6     (soot|method-soot|unit ?soot|method ?soot|unit)
7     (soot|unit|writes-soot|valuebox ?soot|unit ?vbox)
8     (soot|valuebox-soot|value ?vbox ?value)
9     (succeeds (instance? soot.jimple.FieldRef ?value))
10    (equals ?soot|field (.getField ?value))))
```

The goal on line 4 retrieves SOOT's representation of the method that represents the weaver's advice `?advice` in the woven program. The goal on line 5 does the same for the weaver's field `?field`. The remaining goals use predicates that reify relations between SOOT elements only. Upon backtracking, the goal on line 6 will successively unify `?soot|unit` with one of the units in the body of `?soot|method`. These correspond to instructions in SOOT's JIMPLE intermediate representation [22] of the woven program. Lines 7–10 ensure that this unit writes to the SOOT field `?soot|field` that represents the weaver's field `?field` in the woven program. Note that the final goal calls out to SOOT to resolve a field reference to the referenced field —possible because we forego a conversion to logic facts.

Predicates such as `advice-reachable|advice/2`, which reifies the relation between an advice and another advice such that the latter may be executed after the former, require more detailed information about the woven program. They are hence implemented in terms of predicates that quantify over the paths through an inter-procedural control flow graph of the woven program (fourth row in Table II). We compute this graph by linking the intra-procedural control flow graphs of callers and callees using the results of SOOT's points-to analysis, *i.e.,* the demand-driven, context-sensitive version by Sridharan et al. [24]. We refer to the online documentation of EKEKO for behavioral predicates that reify may-alias and must-alias dataflow relations between SOOT values based on this analysis.

To summarize: the possible methods an invocation may resolve to are determined using a compile-time approximation of the dynamic type of its receiver, *i.e.,* the types of the objects in its points-to set, rather than its static type — which is more precise. Note that multiple call sites result in control flow splits at the exit points of callees for link-based whole-program graphs. Our graph traversal predicates therefore take care not to follow unrealizable paths, without endangering termination (*i.e.,* a *finite* call stack ensures that successors of a method's exit node agree with an earlier method invocation).

Of the graph traversal predicates at the bottom of Table II, `path/n` is of special interest as it embodies the implementation of parametric regular path expressions [25], [26] in EKEKO (which we have applied in earlier work to query the history of versioned software [23]). Regular path expressions are an intuitive formalism for quantifying over the paths through a graph. They are akin to regular expressions, except that they consist of logic goals to which regular expression operators have been applied. Rather than matching a sequence of characters in a string, they match paths through a graph along which their logic goals succeed. This is illustrated by the implementation of predicate `advice-reachable|advice/2` below:

```
1 (defn advice-reachable|advice [?advice1 ?advice2]
2   (fresh [?s|method1 ?s|method2
3           ?icfg ?icfg|start ?icfg|end]
4     (advice-soot|method ?advice1 ?s|method1)
5     (differs ?advice1 ?advice2)
6     (advice-soot|method ?advice2 ?s|method2)
7     (icfg|main-start ?icfg ?icfg|start)
8     (path ?icfg ?icfg|start ?icfg|end []
9       (q=>*)
10      (qcurrent [?n]
11        (icfgnode-method ?n ?s|method1))
12      (q=>+)
13      (qcurrent [?n]
14        (icfgnode-method n ?s|method2)))))
```

The goals on lines 4–6 of quantify over two distinct advices and their corresponding SOOT methods in the woven program. Line 7 unifies `?icfg` with an inter-procedural control flow graph that starts at the `main()` method of the woven program. The goal on line 8 succeeds if there is a path through this graph from node `?icfg|start` to `?icfg|end` that is of the form described by the regular path expression in its body: zero or more non-distinct nodes (*i.e.,* nodes against which no logic goals have to succeed) (line 9), followed by one node that resides in the SOOT method corresponding to `?advice1` (lines 10-11), followed in turn by one or more non-distinct nodes (line 12), concluded by a node that resides in the SOOT method corresponding to `?advice2` (line 13).

Note that a similar regular path expression can be used to warn about possibly incorrect implementations of the wormhole pattern described in Section II-C. These are characterized by an execution path on which the wormholed field is written to inbetween the entry and exit advice.

V. DETECTING ASPECT ASSUMPTIONS WITH GASR

As an illustration of the usefulness of GASR we now show how it can be used to implement detection of developers' assumptions about aspect usage, effectively extending the "Aspect Assumptions" work of Zschaler and Rashid [10]. For brevity, in the rest of this section we will refer to this work as AA. For AA, Zschaler and Rashid have studied three nontrivial aspectual systems to discover the assumptions made by the different modules about the functionality, presence and implementation of other modules. The authors assert that assumptions that aspects make about the system are "particularly critical" [10] because of the cross-cutting nature of aspects as well as their implicit invocation. They start a catalogue of such assumption types, based on the assumptions discovered in their case studies. To discover these, their investigation consisted of manual inspection of the source code and developer interviews.

AA also proposes a followup that, to the best of our knowledge, has not yet been performed. It consists in codifying the assumptions such that these can be "used to semi-automatically identify assumptions in other aspect code" [10]. This would allow, on the one hand for implicit assumptions to be elicited from the source code, and on the other hand for explicit assumptions to be verified. For the latter, the ideal case would be "making fully automatic verification a feasible goal for at least some of the assumption categories" [10]. In this

section we show how GASR can be used to perform exactly this. We implement elicitation rules for a subset of the aspect assumptions and run them on two of the three case studies used in AA[6]. We consider that providing a complete set of rules would be a separate contribution and hence out of the scope of this work.

Concretely, we restrict ourselves to inter-aspect assumptions (Sect. 3.1.1 in [10]) and run the experiments on the HealthWatcher [27] and MobileMedia [28] systems. We implemented analysis rules for all assumptions that can be sufficiently formalized, or approximated by a heuristic. All rules were developed on a test-first basis and both the rules and the test cases are available online[7]. After running the analyses, the results were verified for correctness and completeness. This was achieved by manually inspecting both the source code as well as the full list of assumption instances published as additional material of the AA paper[8]. Our results confirm the assumption instances listed and, more importantly, provide new assumption instances that were overseen in AA. The latter clearly demonstrates the advantages of automatic aspectual source code reasoning, as provided by GASR.

Due to lack of space, we cannot fully document all the analyses we created for assumption identification. Instead we choose to focus here on interesting analyses: those that achieve fully automatic verification, reveal new assumption instances and show customizability.

A. Assumptions on concretisation of pointcuts

The first assumption we talk about here was already mentioned in Sect. II-C: an abstract pointcut that is concretized in a subclass and re-concretized in one of its subclasses. The assumption is that in such a case the aspect actually wishes to preserve existing behavior and hence should not override already concretised pointcuts. We can use GASR to perform fully automatic verification of this assumption, yielding a first step of the followup work proposed in the AA paper. The following logic rule will reveal violations of the assumption:

```
1 (defn pointcut-concretized-reconcretized
2   [?pointcut ?cpointcut ?rcpointcut]
3   (all
4     (pointcut-concretizedby ?pointcut ?cpointcut)
5     (pointcut-concretizedby ?cpointcut ?rcpointcut)))
```

In line 4 of the code above, we find a `?pointcut` that is concretised by a second `?cpointcut`, and in line 5 we find a `?rcpointcut` that concretises `?cpointcut`. Any solutions for this goal hence consist of a `?pointcut` that is concretised by `?cpointcut` and reconcretised by `?rcpointcut`.

We have queried both example case studies for matches of this rule and have found none. In other words there are no cases where this assumption has been violated. This is in accordance to the results published in AA.

B. Precedence assumptions

ASPECTJ provides for a mechanism to order the execution of advice when multiple advice apply at a given join point. It consists of precedence relations between different aspects and advice. This results in a domination order that determines the execution order of advice. The language contains implicit precedence rules that determine dominance between the aspects in an inheritance tree. Additionally, dominance between advice of the same aspect is determined by their order in the source code. The developer may also explicitly declare precedence between different aspects. One precedence assumption stated in AA is that implicit precedence rules between aspects are not modified by explicit precedence declarations. GASR can also be used to provide fully automatic verification of this assumption, as follows:

```
1 (defn overridden|imp|precedence [?asp1 ?asp2]
2   (all
3     (aspect|dominates-aspect ?asp2 ?asp1)
4     (aspect|implicitdominates-aspect+ ?asp1 ?asp2)))
```

Line 3 of the above rule provides bindings for domination relationships between aspects that have been explicitly declared, while line 4 succeeds for implicit domination relationships that are the opposite. The resulting bindings hence violate the aspect assumption. We have found none in the case studies, again in accordance to the results found in AA.

C. Inclusion assumptions of aspects

AA describes inclusion assumptions of aspects in general as "Some aspects require other aspects to be deployed to function correctly." This assumption cannot be unambiguously defined in a code rule. The paper however also identifies a specific variant: an aspect defines a marker interface, *i.e.,* an empty interface, and another aspect contains a declare parents statement that adds it as an implemented interface to a given class. The cases identified in the code studied for AA are actually a generalization of this: the interface sometimes is stand-alone, *i.e.,* defined in its own compilation unit. Moreover, aspects may refer to a sub-interface of this interface. The rules below successfully identify these assumption instances:

```
1 (defn markerinterface [?interface]
2   (fresh [?member]
3     (interface ?interface)
4     (fails (type-member ?interface ?member))))

5 (defn aspect-declareparents|markerinterface
6   [?aspect ?interface]
7   (fresh [?superinterface ?declare]
8     (markerinterface ?superinterface)
9     (iface-self|or|sub ?superinterface ?interface)
10    (declare|parents-parent|type ?declare ?interface)
11    (aspect-declare ?aspect ?declare)))
```

This code first defines a rule for a marker interface: an interface (line 3) that fails to have any members (line 4), *i.e.,* is a marker interface. This is then used in the assumption rule as a goal in line 8. Line 9 provides bindings for the interface and all its direct and indirect subinterfaces in `?interface`. As a result, line 10 succeeds on all declare parents statements of marker interfaces or their (in)direct subinterfaces. Line 11 reveals the `?aspect` that contains this declaration.

The above rules do not only identify the known instances of this assumption. More importantly, they also reveal three previously unidentified instances in the HealthWatcher case. Firstly, the ServletCommanding aspect refers to the CommandReceiver empty interface, which is stand-alone and

[6]The third system investigated in AA fails to compile due to an ASPECTJ internal compiler error and hence could not be analysed by us.

[7]Available at https://github.com/cderoove/damp.ekeko.aspectj

[8]Available at http://www.steffen-zschaler.de/publications/rivar_data/

specifically designed for aspects to use as a marker interface, as revealed by its comments. Secondly, the UpdateStateObserver aspects refers to the Observer interface contained in the ObserverProtocol aspect. This nested interface was also created specifically for other aspects to mark, as indicated by its comments. Thirdly, analogous to the previous instance, UpdateStateObserver also refers to the Subject interface contained in the ObserverProtocol aspect, also created for this. It is unclear why these are not present in the AA raw data as they are indubitably assumption instances.

D. Mutual Exclusion Assumptions

AA states that "aspects may also be mutually exclusive", *i.e.,* of the mutually exclusive set only one aspect may be deployed. Again, this assumption cannot be unambiguously defined in a code rule. We can infer some heuristics that can however give possible cases for such a mutual exclusion. Based on the conjecture that mutually exclusive aspects may provide different implementations for the same feature and hence act on the same parts of the software, we present two such heuristics here: the same pointcut name and the same join point shadows. Note that we do not claim that this conjecture and the heuristic is particularly efficient, nor even valid. These only serve as an illustration of the use of GASR.

1) Same Pointcut Name: For this heuristic we assume that the name of the pointcuts convey their semantics and hence if two aspects use pointcuts with the same name they may implement the same feature. The following rule reveals such aspects:

```
1 (defn same|pointcutname-aspect1-aspect2
2   [?name ?aspect1 ?aspect2]
3     (fresh [?pc1 ?pc2]
4       (differs ?aspect1 ?aspect2)
5       (aspect-pointcutdefinition ?aspect1 ?pc1)
6       (aspect-pointcutdefinition ?aspect2 ?pc2)
7       (pointcutdefinition-name ?pc1 ?name)
8       (pointcutdefinition-name ?pc2 ?name)))
```

The code of the rule is straightforward, obtaining pointcut definitions of two different aspects where the name of the pointcut is the same. For the HealthWatcher case this rule only reveals two cases where an abstract pointcut is concretized. In MobileMedia however 146 cases are detected, defying manual analysis of each case. It is immediately apparent that a small subset of pointcut names are present a sizeable amount of times: "handleCommandAction", "initMenu" and "constructor". This is as many aspects are used to implement a command pattern and match on these pointcuts to realize the pattern. Using GASR we can eliminate these matches from the rule by amending extra conditions to the query, as below:

```
1 (ekeko [?name ?as1 ?as2]
2   (all
3     (same|pointcutname-aspect1-aspect2 ?name ?as1 ?as2)
4     (differs ?name "handleCommandAction")
5     (differs ?name "constructor")
6     (differs ?name "initMenu")))
```

Running this query returns in eleven different matches, which is a number that allows for manual analysis. This actually reveals six cases of copy-paste reuse of a pointcut: "createMediaData", "getMediaController", "goToPreviousScreen", "initForm", "appendMedias" and "startApp". The two remaining pointcut names: "resetMediaData" and "showImage" do not reveal mutual exclusion of aspects.

The use of this heuristic did not reveal assumption instances but is nonetheless valuable. This is as its use in the MobileMedia case study illustrates the advantage of a general-purpose code reasoner to adapt code queries to the actual case being studied, in this case filtering out a high number of false negatives. This resulted in the discovery of six cases of copy-paste reuse, arguably not a good characteristic of the code.

2) Same join point shadows: Using the same pointcut names is not the only possible indication of implementing the same features. This second heuristic considers the join point shadows of two aspects. If two different aspects have the same collection of shadows, they may implement the same feature. This can be detected using the code below.

```
1 (defn  sameshadows|aspect1-aspect2
2  [?aspect1 ?aspect2]
3  (fresh [?shadows1 ?shadows2]
4    (aspect ?aspect1) (aspect ?aspect2)
5    (differs?aspect1 ?aspect2)
6    (findall ?shadow1
7      (aspect-shadow ?aspect1 ?shadow1) ?shadows1)
8    (findall ?shadow2
9      (aspect-shadow ?aspect2 ?shadow2) ?shadows2)
10   (differs ?shadows1 []) (differs ?shadows2 [])
11   (same-elements ?shadows1 ?shadows2)))
```

Notable here are lines 6 and 8: all shadows of both aspects are gathered in the collections `?shadows1` and `?shadows2`, respectively. Line 10 ensures that the collections are not empty, to exclude aspects without advice. Line 11 verifies that the collections have the same elements, *i.e.,* are the same.

For the HealthWatcher case two matches are found: HWTransactionExceptionHandler and HWDistributionExceptionHandler. Both aspects perform complementary exception handling: one for transaction exceptions and one for RMI exceptions. In MobileMedia nine different matches are found, of which one is a mutual exclusion case: OneAlternativeFeature and TwoAlternativeFeatures. Both add an exit command to the same menu and hence are mutually exclusive. This case is not mentioned in the AA paper but is present in the raw data.

E. Assumptions on the use of Inter-Type Declarations

One kind of purpose for inter-type declarations is to provide additional public features that are packaged in the aspect. In these cases these methods "are often not used from the declaring aspect" [10], so the assumption is that other code will call these aspects at some points. GASR accomplishes automatic verification of this assumption by reasoning over the results of our extended soot analysis, discussed in Sect. IV-B. The code is as follows:

```
1 (defn intertypemethod|unused [?itmethod]
2   (fresh [?sootmethod ?caller]
3     (intertype|method ?itmethod)
4     (fails (all
5       (intertype|method-soot|method
6           ?itmethod ?sootmethod)
7       (soot|method|callee-soot|method|caller
8           ?sootmethod ?caller)))))
```

In the code above, line 3 provides bindings for methods that are inter-type declarations in `?itmethod`. Lines 5 and 6

provide the bridge between the method and the corresponding soot method, and lines 7 and 8 find methods that call that soot method, binding these to `?caller`. The goal in line 4 fails if the goals in lines 5–8 succeed, *i.e.,* if no bindings can be found for `?caller`. As a result the rule succeeds for inter-type declarations that are not called.

In the case studies we have found one violation of this assumption in HealthWatcher: Command.isExecutable() in the CommandProtocol aspect is a default implementation for the abstract method declared in the Command class. Yet this method is never referenced at all. A record of this violation of the assumption is as it is not present in the raw data of AA. Hence this is again a new discovery made thanks to GASR.

F. Conclusion

In this section we have illustrated the usefulness of GASR by implementing detection of aspect assumptions, as originally envisioned by Zschaler and Rashid [10], which we name AA for brevity. We have implemented logic rules for the detection of inter-aspect assumptions, effectively achieving some of the future work laid out in AA. We have run these rules on two of the three systems used in AA for discovering the aspect assumptions. The results of these GASR queries were verified for correctness and completeness by manually inspecting both the source code as well as by cross-checking with the full published list of assumption instances.

All the assumption instances that are present in the AA paper or in the raw data were detected using GASR. More important though are the three following results: First, we achieved fully automatic verification of assumption instances in Section V-A and V-B. Second, we also detected three previously unknown assumption instances in Section V-C and one in Section V-E. Third, we have shown in Section V-D1 how the general-purpose nature of GASR enables the tailoring of an existing rule to the software under study. Thanks to this, we incidentally found six cases of copy-paste reuse of pointcuts.

VI. THREATS TO VALIDITY

We consider GASR a reasonable baseline for a general-purpose source code analysis tool for aspect-oriented programming. However we can not and do not claim that it is suitable for all possible kinds of reasoning over aspectual source code.

Considering the above restrictions, the first threat to validity is the fact that we only performed the experiments on two concrete cases. Nonetheless, the rules were developed independently of the case studies, on a test-first basis, and should hence perform equally well on other case studies. Secondly, we have only shown here that GASR works for rules from the work on Aspect Assumptions [10]. As highlighted in Section III, there is a large amount of work that performs analysis of aspectual code and we do not validate that GASR is as effective for those cases. By providing a comprehensive library of predicates, discussed in Section IV-B, we do however provide a large number of basic building blocks that can be used to build these analyses using GASR. It remains to be shown whether the library is extensive enough. If not, it may need to be extended.

GASR is a source code analysis tool that works, as-is, on ASPECTJ source code only. It has however been argued before, *e.g.,* by Zschaler and Rashid [10], that ASPECTJ is probably the most used aspect-oriented language. As a result, GASR can be used to analyse a large amount of aspectual source code. Moreover, the predicate library is relatively language-agnostic as it works in terms of the aspect-oriented concepts. We are confident that if the library is adapted to work on other languages, the majority of analyses built using GASR will be straightforwardly reusable. We have demonstrated similar results previously in earlier work on language-independent source code analysis [29]. Hence, in our opinion, GASR truly is general-purpose.

VII. CONCLUSION AND FUTURE WORK

There is a need for source code analysis of aspect-oriented source code that is demonstrated by the multiple tracks of research performing such analysis. On the one hand, existing analyses need to be extended to take into account the aspect-oriented nature of the software, and on the other hand this nature gives rise to new kinds of analyses being required. Yet, to the best of our knowledge, all of this work has been ad-hoc and limited in scope to the specific analysis at hand. As a result there has been a significant amount of duplicate work and it is unclear whether any analyses may be customized to the software being analysed, *e.g.,* as we perform in Section V-D1.

We state that what is required is a general-purpose aspectual source code analysis tool, such that duplicate work building analyses may be avoided and that existing analyses may be customized to the task at hand. To the best of our knowledge no such work has yet been published.

To address this need, we have implemented GASR: a source code analysis tool in the tradition of logic querying. GASR is a General-purpose Aspectual Source code Reasoner whose analyses may be customized relatively straightforwardly, as illustrated in Section V-D1. In this paper we presented GASR, have shown illustrative predicates for the reification of structural and behavioral relations, and discussed their implementation.

Following this, we performed source code analysis on two representative pieces of aspect-oriented software. We detected a subset of inter-aspect assumptions that were previously identified by Zschaler and Rashid [10] by manual inspection of the same software. Our automated analysis effectively consists in realizing part of the future work outlined in that text: allowing detection of assumptions and fully automatic verification that some assumptions are not being violated. We detected the same assumption instances as Zschaler and Rashid. More importantly, we also found assumption instances that were overlooked by these authors.

There are multiple avenues for possible future work. Firstly, we contend that the current state of GASR is a reasonable baseline for performing aspect-oriented source code analysis but do not assert that it is sufficient for all kinds of analyses. More experiments, implementing different analyses and executing them on multiple case studies may reveal areas where GASR is lacking. Secondly, the assumptions of Zschaler and Rashid [10] which we did not implement yet are an avenue for further work. Some of these however require the source code to be annotated somehow with the intent of the developer. This would require some formal notation for these intentions and GASR to be extended to reason over these annotations.

Thirdly, GASR can be seen as base infrastructure on which new and more advanced analyses can be built. The possibilities are vast, obviously. Our personal preferences are for analyses that can extract design-level documents [30] and that provide information on whether there exist any dependencies and interactions between aspects [31].

ACKNOWLEDGMENTS

Johan Fabry is partially funded by FONDECYT project number 1130253, Coen De Roover is funded by the *Cha-Q* SBO project sponsored by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen). Thanks to Romain Robbes for feedback on draft versions of this text.

REFERENCES

[1] L. Ye and K. De Volder, "Tool support for understanding and diagnosing pointcut expressions," in *Proceedings of the 7th international conference on Aspect-oriented software development*, ser. AOSD '08. New York, NY, USA: ACM, 2008, pp. 144–155.

[2] R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu, "Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software," in *24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, 2009, pp. 575–579.

[3] C. Koppen and M. Stoerzer, "Pcdiff: Attacking the fragile pointcut problem," in *European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[4] A. Kellens, K. Mens, J. Brichau, and K. Gybels, "Managing the evolution of aspect-oriented software with model-based pointcuts," in *European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, no. 4067, 2006, pp. 501–525.

[5] D. Zhang, E. Duala-Ekoko, and L. Hendren, "Impact analysis and visualization toolkit for static crosscutting in aspectj," in *International Conference on Program Comprehension (ICPC)*, 2009.

[6] H. Shen, S. Zhang, J. Zhao, J. Fang, and S. Yao, "Xfindbugs: extended findbugs for aspectj," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE08)*, 2008, pp. 70–76.

[7] G. Xu and A. Rountev, "Ajana: a general framework for source-code-level interprocedural dataflow analysis of aspectj software," in *Proceedings of the 7th international conference on Aspect-oriented Software Development (AOSD08)*, ser. AOSD '08, 2008, pp. 36–47.

[8] A. Colyer, A. Clement, G. Harley, and M. Webster, *Eclipse AspectJ: aspect-oriented programming with AspectJ and the Eclipse AspectJ development tools*. Addison-Wesley Professional, 2004.

[9] J. Fabry, A. Kellens, and S. Ducasse, "AspectMaps: A scalable visualization of join point shadows," in *Proceedings of 19th IEEE International Conference on Program Comprehension (ICPC2011)*. IEEE, Jul 2011, pp. 121–130.

[10] S. Zschaler and A. Rashid, "Aspect assumptions: a retrospective study of aspectj developers' assumptions about aspect usage," in *Proceedings of the tenth international conference on Aspect-oriented software development*, ser. AOSD '11. New York, NY, USA: ACM, 2011, pp. 93–104.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of AspectJ," in *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, ser. Lecture Notes in Computer Science, J. L. Knudsen, Ed., no. 2072. Budapest, Hungary: Springer-Verlag, Jun. 2001, pp. 327–353.

[12] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein, "Multijava: modular open classes and symmetric multiple dispatch for java," in *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '00. New York, NY, USA: ACM, 2000, pp. 130–145.

[13] J. Xu, H. Rajan, and K. Sullivan, "Understanding aspects via implicit invocation," in *Proceedings. 19th International Conference on Automated Software Engineering (ASE)*, 2004, pp. 332–335.

[14] S. Hanenberg and A. Schmidmeier, "Idioms for building software frameworks in aspectj," in *Proceedings of the workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) at AOSD 2003*, 2003, p. 55.

[15] R. Laddad, *AspectJ in action*, 2nd ed. Manning Publications, 2009.

[16] J. Fabry, A. Kellens, S. Denier, and S. Ducasse, "AspectMaps: Extending Moose to visualize AOP software," *Science of Computer Programming*, 2013, To appear. http://dx.doi.org/10.1016/j.scico.2012.02.007.

[17] E. Hajiyev, M. Verbaere, and O. de Moor, "CodeQuest: Scalable source code queries with Datalog," in *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP06)*, ser. Lecture Notes in Computer Science, vol. 4067, 2006, pp. 2–27.

[18] M. Martin, B. Livshits, and M. Lam, "Finding application errors and security flaws using PQL: a program query language," in *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA05)*, 2005, pp. 365–383.

[19] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "The SOUL tool suite for querying programs in symbiosis with eclipse," in *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ11)*, 2011.

[20] W. E. Byrd, "Relational programming in minikanren: Techniques, applications, and implementations," Ph.D. dissertation, Indiana University, August 2009.

[21] A. K. Carlos Noguera, Coen De Roover and V. Jonckers, "Program querying with a SOUL: the barista tool suite," in *Proceedings of the 27th IEEE International Conference on Software Maintenance, Tool Demo Track (ICSM11)*, 2011.

[22] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, Oct. 2011.

[23] R. Stevens, C. De Roover, C. Noguera, and V. Jonckers, "A history querying tool and its application to detect multi-version refactorings," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*, 2013.

[24] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI06)*, 2006.

[25] O. de Moor, D. Lacey, and E. V. Wyk, "Universal regular path queries," *Higher-order and Symbolic Computation*, vol. 16, no. 1-2, pp. 15–35, 2003.

[26] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu, "Parametric regular path queries." in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI04)*, 2004, pp. 219–230.

[27] S. Soares, P. Borba, and E. Laureano, "Distribution and persistence as aspects," *Software: Practice and Experience*, vol. 36, no. 7, pp. 711–759, 2006.

[28] E. Figueiredo, I. Galvao, S. Khan, A. Garcia, C. Sant'Anna, A. Pimentel, A. Medeiros, L. Fernandes, T. Batista, R. Ribeiro, P. van den Broek, M. Aksit, S. Zschaler, and A. Moreira, "Detecting architecture instabilities with concern traces: An exploratory study," in *Proceedings of 8th Joint Working IEEE/IFIP Conference on Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009.*, 2009, pp. 261–264.

[29] J. Fabry and T. Mens, "Language-independent detection of object-oriented design patterns," *Science of Computer Programming*, vol. 30, no. 1-2, pp. 21–33, April-July 2004.

[30] J. Fabry, A. Zambrano, and S. Gordillo, "Expressing aspectual interactions in design: Experiences in the slot machine domain," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, J. Whittle, T. Clark, and T. Kühne, Eds. Springer Berlin / Heidelberg, 2011, vol. 6981, pp. 93–107.

[31] R. Chitchyan, J. Fabry, S. Katz, and A. Rensink, "Editorial for special section on dependencies and interactions with aspects," *Transactions on Aspect-Oriented Software Development*, vol. LNCS 5490, pp. 133–134, 2009.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

Driving a Sound Static Software Analyzer with Branch-and-Bound

Sven Mattsen
Hamburg University of Technology (TUHH)
Institute for Software Systems (STS)
sven.mattsen@tuhh.de

Pascal Cuoq
CEA, LIST
Gif-sur-Yvette
91191, France
pascal.cuoq@cea.fr

Sibylle Schupp
Hamburg University of Technology (TUHH)
Institute for Software Systems (STS)
schupp@tuhh.de

***Abstract*—During the last decade, static analyzers of source code have improved greatly. Today, precise analyzers that propagate values for the program's variables, for instance with interval arithmetic, are used in the industry. The simultaneous propagation of sets of values, while computationally efficient, is a source of approximations, and ultimately of false positives. When the loss of precision is detrimental to the user's goals, a user needs to provide some kind of manual guidance. Frama-C, a framework for the static analysis of C programs, provides a sound value analyzer. This analyzer can optionally be guided by skillfully placed user annotations.**

This article describes SPALTER, a Frama-C plug-in that uses a variation of the Skelboe-Moore algorithm from the field of interval arithmetic to guide Frama-C's value analyzer towards a high-level objective set by the user. SPALTER reproduces the results of a case study that used Frama-C's value analysis and required extensive manual guidance. In difference, our approach with SPALTER required no guidance, except preparation of the analyzed program by slicing.

I. Introduction

The common use case for sound static analyzers is to prove the absence of entire classes of bugs. Frama-C's value analysis [1] does this by computing variation domains for all variables of a C program, and checking if included values can cause run-time errors. If so, an alarm is emitted. It becomes the user's responsibility to check whether this alarm represents an actual bug or is caused by over-approximation. If the user is convinced that the alarm is caused by approximation, (s)he might want to guide the value analysis towards a less approximate result. Otherwise, a concrete input vector that causes the unwanted behavior is desirable: it confirms the alarm and helps with comprehension and debugging.

One way to work towards either goal is to sub-divide input variation domains. With Frama-C's value analysis, the user can provide sub-variation domains to analyze separately for improved precision [2, 7.1.2 §Case analysis]. This requires understanding of the target program and of the analyzer's principles, and is therefore at odds with the idea of a mostly automatic analyzer.

SPALTER provides a fully automatic alternative for the precise analysis of functions at the unit level. It does not require the manual insertion of sub-division annotations. SPALTER uses a variation of the Skelboe-Moore algorithm [3, 6.4 §Skelboes's Algorithm] from interval arithmetic, to improve precision towards a goal set by the user.

This article is structured as follows. In Section II we provide a brief introduction to Frama-C's value analysis. Subsequently, Section III describes the Skelboe-Moore algorithm and the challenges in adapting it to Frama-C. In sections IV and V, we go on to show SPALTER's effectiveness in removing false alarms and computing more precise bounds. Afterwards, we share some lessons learned in Section VI, discuss related work in Section VII, outline future work in Section VIII, and conclude in Section IX.

II. Frama-C's Value Analysis

Frama-C's value analysis is non-relational, and it can produce sub-optimal results on even simple expressions. But any abstract interpreter has its limits. If another abstract interpreter was used instead of Frama-C's value analysis, this section's message would remain the same. Only the examples would change.

Consider the source code snippet below, inspired by a question on the Frama-C mailing list[1]:

```
/*@ requires 0 <= x <= 20;
ensures \result >= 0;*/
int foo(int x) {
  return x - (x % 10);
}
```

The user was surprised that Frama-C's value analysis, without any manual help, was unable to conclude that the postcondition `\result >= 0` always holds. Without user assistance, Frama-C determines the result of the function `foo()` to be within the variation domain $[-9\,..\,20]$, a sound but approximate prediction. The reason is that whenever two operands in the analyzed program are in relation to each

[1] http://lists.gforge.inria.fr/pipermail/frama-c-discuss/2012-July/003292.html

978-1-4673-5739-5/13/$31.00 © 2013 IEEE

other, Frama-C's value analysis may fail to take the relation into account, and lose precision.

In the current example, the operands of the subtraction, `x` and `x % 10`, are related. This relation is lost in interval arithmetic. Frama-C's value analysis manual describes how to force the analyzer to take relations into account, via explicit sub-division. In the above example, a helpful sub-division of x is $x^1 \in [0\,..\,4]$, $x^2 \in [5\,..\,20]$. When making Frama-C analyze `foo()` for x^1 and x^2 separately, a relation between x^n and the computed variation domain for $\texttt{x}^n$ `% 10` is introduced, which improves the precision of the variation domain of `foo()`'s return value to $[-4\,..\,20]$. However, the annotation necessary to achieve maximum precision is neither obvious nor pleasant to write, as this annotation must instruct a separate analysis for each individual value of x below 10. The annotation would have to list these 10 cases explicitly, together with the additional catch-all case $x >= 10$ for which the post-condition is straightforward to verify. Different examples, or a larger constant in this example, make this approach intractable. With SPALTER, we aim for a general, automatic solution for improving precision when analyzing individual functions, such as `foo()` above, for which Frama-C's value analysis alone provides unsatisfactory results. SPALTER is implemented as a Frama-C plug-in [4].

III. Skelboe-Moore algorithm for value analyzers

A. The original Skelboe-Moore algorithm

The Skelboe-Moore algorithm [3, 6.4 §Skelboes's Algorithm] was originally conceived to improve the precision of interval arithmetic computations through sub-division. However, it can also be seen as a sound algorithm in the field of global optimization, where the goal is to find the extremum of a mathematical function or the input that generates it, or at least an approximation thereof. As a global optimization algorithm, Skelboe-Moore belongs to the class of branch-and-bound algorithms, where "branching" means to sub-divide an input range to analyze it more precisely, and "bounding" means to discard input ranges that can already be deduced not to generate the extremum.

More specifically, the algorithm uses a worklist W, an interval evaluation function F that is an interval extension of f, a goal function g, a goal G, and a range X in which to search for an extremum of $f_g = g \circ F$. The goal function g must fulfill

$$Z' \subseteq Z \Rightarrow g(Z') \sqsubseteq g(Z) \quad \text{(monotonicity)}$$

and we will use $\sqsubseteq$ to order the worklist.

Let us assume we are searching for the greatest lower bound of f. Then, we would define g to be a function that takes an interval and returns its lower bound and the algorithm would work as follows:

1) Set $G = \infty$.
2) Initialize W with $(X, f_g(X))$ and keep it sorted in descending order according to $\sqsubseteq$ on the second tuple element.
3) Take the first tuple element (X, F_G) from the worklist.
4) Generate the intervals $X^1, X^2 \in X$ such that $X^1 \cup X^2 = X$ and $X^1 \cap X^2 = \emptyset$ (branching).
5) Evaluate $[l_1\,..\,h_1] = F(X^1)$ and $[l_2\,..\,h_2] = F(X^2)$.
6) Insert $(X^1, f_g(X^1))$ and $(X^2, f_g(X^2))$ into W.
7) Set $G = \min(G, h_1, h_2)$ and delete all tuples (X, F_G) from W that fulfill $F_G \geq G$ (bounding).
8) Go to 3.

The algorithm terminates when W is empty or when the first element in W is a singleton.

B. Pitfalls when applying Skelboe-Moore to programs

Frama-C's value analysis uses non-relational abstract domains for the values of C expressions: intervals with congruence information for integers, intervals for floating-point, and symbolic representations for pointers. Like in straightforward interval arithmetic, these variation domains can be sub-divided in the hope of reducing approximations. The Skelboe-Moore algorithm uses the evaluation function F as a black box. It can be employed with Frama-C's value analysis as evaluation function, with a few caveats. Indeed, certain assumptions commonly made for interval arithmetic evaluation of mathematical functions, and for the evaluation function in the Skelboe-Moore algorithm, do not hold for Frama-C's value analysis applied to C programs:

1) Inclusion isotonicity: A function is said to be inclusion isotonic if for all possible inputs X_i the following holds:

$$Y_i \subseteq X_i \text{ for } i = 1, ..., n \Rightarrow f(Y_i, ..., Y_n) \subseteq f(X_i, ..., X_n)$$

In other words, if we analyze a function once for all inputs in X and once for only a part of these inputs, inclusion isotonicity ensures that the second analysis is at least as precise as the first one. Within Skelboe-Moore, inclusion isotonicity ensures convergence: The first worklist item will never be less precise than a previous one. For an abstract interpreter, like Frama-C's value analysis, inclusion isotonicity cannot be guaranteed in the presence of widening[2].

However, we can use the fact that Frama-C's value analysis is sound to restore inclusion isotonicity. Assume that F is a function that takes an input range to be analyzed and applies Frama-C's value analysis. Then, whenever widening causes $F(Y \subseteq X)$ to be less precise than $F(X)$, we may safely use $F(X)$ instead of $F(Y)$, since soundness ensures that $F(X)$ includes all feasible values of $F(Y)$. Therefore, we may use the evaluation function $g \circ F'$, where g is a valid goal function and F' is defined as follows:

$$F'(Y \subseteq X) = \begin{cases} F(Y) & \text{if } F(Y) \subseteq F(X) \\ F(X) & \text{otherwise} \end{cases}$$

[2] http://blog.frama-c.com/index.php?post/2011/09/12/better

2) Totality: C functions can be partial through run-time errors and non-termination. In both cases, a C function can not be relied upon to return a value for each input.

Run-time errors are provided by the value analysis "on the side", and the values it predicts are for the legal executions only. In the following function, run-time errors render it hard to optimize the lower bound of the return value:

```
/*@ requires 0 <= x <= 100 ; */
int f(int x){
  y = 1 / (x * (x-1));
  if (x <= 1) return 0;
  return x;
}
```

The actual minimum of function `f()` for legal executions is 2, obtained for the input 2. If the Skelboe-Moore algorithm leads to the study of the input subrange $[0\,..\,1]$, interval arithmetic makes it look like the output can be zero. Specifically, interval arithmetic makes it suggests there might be some defined executions of the function `f()` for $[0\,..\,1]$, and the result interval $[0\,..\,0]$ looks like it cannot be improved upon. The classic Skelboe-Moore would stop there, and return 0 as a bound. In our use case, it is necessary to refrain from the optimization that consists in re-using the already computed h_1 and h_2 at step 7. In the example of function `f()`, 0, as a proposed minimum, is sound but approximate.

It may seem that partiality is only an issue with run-time errors, and that first verifying the absence of run-time errors would eliminate the problem. But non-termination introduces the same problem, and unlike run-time errors, Frama-C's value analysis does not even inform the user when some of the inputs may prevent the function from terminating. For an example that illustrates the issue with non-termination, consider the following C source:

```
/*@ requires 0 <= x <= 100 ; */
int f(int x){
  while (!(x * (x-1))) /* do nothing */ ;
  if (x <= 1) return 0;
  return x;
}
```

Because interval arithmetic causes $x * (x-1)$ to evaluate to $[-1\,..\,0]$ for $x \in [0\,..\,1]$, Frama-C does not detect that the **while** loop will not terminate for this range. As a consequence, it looks like `f()` may terminate for the input $[0\,..\,1]$ and return $[0\,..\,0]$, which would again appear to be the function's minimum. However, the actual minimum for the function `f()` is 2.

In order to circumvent the issue with partial functions, regardless of whether it originates from run-time errors or from infinite loops, SPALTER avoids deducing bounds from the evaluation of sub-intervals. Frama-C's value analysis propagates singleton inputs into singleton outputs—and can be configured never to join any abstract states, making it a plain (non-abstract) interpreter [5]. This property can be relied upon in order to obtain bounds that are guaranteed to be feasible. At the time of obtaining a bound for Skelboe-Moore's algorithm, the value analysis is called on a singleton input vector picked arbitrarily in the subdomain that contains the best values so far. If execution does not terminate for this particular input, another can be picked. If the subdomain does not contain any input vector that makes execution terminate, this subdomain was a red herring: it is discarded, and the next subdomain in the queue is considered instead.

IV. Eliminating False Alarms

Besides optimizing the lower or upper bound of a function's return value, SPALTER can also be used to diagnose value analysis alarms as false. For this, we define a goal function g to map function inputs to a boolean that represents the absence of alarms. Then, we define an order on booleans as `True` > `False` and use SPALTER to find the minimum of g. SPALTER will terminate if it finds a concrete value for which an alarm is present, or when it finds that all alarms vanish if the inputs are split into narrow enough variations domains. This way, instead of trying to optimize the precision of a variation domain's border, SPALTER adapts the sub-divisions to the goal of classifying all alarms.

We illustrate this mode on the following example[3]:

```
struct point { double x; double y; };
const struct point t[] =
  {  {0.,    1.}
    ,{0.125, 1.00782267782571089}
    ,{0.25,  1.03141309987957319}
    ,{0.5,   1.1276259652063807}
    ,{1.,    1.54308063481524371}
    ,{2.,    3.76219569108363139}
    ,{5.,    74.2099485247878476}
    ,{10.,   11013.2329201033244} };

/*@ requires 0. <= x <= 10. ;
    ensures 0 <= \result <= 6 ;
    ensures t[\result].x <= x <= t[\result + 1].x ;
*/
int choose_segment(double x);

/*@ requires 0. <= x <= 10. ; */
double interp(double x) {
  double x0, x1, y0, y1;
  int i;
  i = choose_segment(x);
  x0 = t[i].x;
  y0 = t[i].y;
  x1 = t[i+1].x;
  y1 = t[i+1].y;
  return (y1 * (x - x0) + y0 * (x1 - x)) / (x1 - x0);
}
```

Frama-C's value analysis without assistance, run on the `interp()` function, emits an alarm regarding a possible floating-point overflow for the division. The cause is that

[3] http://blog.frama-c.com/index.php?post/2011/03/23/Helping-the-value-analysis-1

the value analysis does not detect that the variation domain of expression `x1 - x0` does not actually include 0. This emitted alarm is false and should therefore disappear with a more precise analysis. In fact, it disappears as soon as sub-division makes the variation domain of `i` a single concrete value. Then, the relation between `i` and `i + 1` is captured and `x1` and `x0` have disjoint values, allowing the conclusion that there is no floating-point overflow in function `interp()`.

It takes SPALTER about one second of CPU time to conclude that the `interp()` function is safe, with no user intervention other than launching it.

V. Andy Sloane's Donut

The previous examples are simple and can also be handled manually by a knowledgeable user. But SPALTER scales to difficult functions that require fine-grained sub-division before a conclusion can be reached.

Andy Sloane's winning 2006 IOCCC entry is a donut-shaped C program that animates an ASCII revolving donut using a Z-buffer technique. The safety of that program was previously analyzed manually with Frama-C's value analysis[4]. The conclusion of the case study is that the program is safe, but this conclusion requires both human time for instrumentation, and a considerable amount of computing time.

The last step in determining the donut's safety consisted of proving that a `compute()` function, sliced from the original program, does not produce return values larger than 11 (this value is used as index in the string `".,-~:;=!*#$@"` for dithering). We use the plug-in SPALTER to compute the upper bound of function `compute()`, illustrating its operation principles on this example.

The computation at the core of the obfuscated donut program is captured by the `compute()` function below.

```
double Frama_C_sin_precise(double), Frama_C_cos_precise(double);
double sin(double x) { return Frama_C_sin_precise(x); }
double cos(double x) { return Frama_C_cos_precise(x); }
/*@ requires -0x1.921ff2e48e8a7p+1 <= A <= 0x1.921ff2e48e8a7p+1
&& -0x1.921ff2e48e8a7p+1 <= B <= 0x1.921ff2e48e8a7p+1
&& -0x1.921ff2e48e8a7p+1 <= i <= 0x1.921ff2e48e8a7p+1
&& -0x1.921ff2e48e8a7p+1 <= j <= 0x1.921ff2e48e8a7p+1;
ensures \result <= 11;*/
int compute(float A, float B, float i, float j) {
  float c = sin(i), l = cos(i);
  float d = cos(j), f = sin(j);
  float g = cos(A), e = sin(A);
  float m = cos(B), n = sin(B);
  int N = 8 *
  ((f * e - c * d * g) * m - c * d * e - f * g - l * d * n);
  return N > 0 ? N : 0;
}
```

Without help from the user, Frama-C's value analysis claims that function `compute()` returns values within $[0\,..\,40]$, which is correct but imprecise. SPALTER needs about 1.5 million iterations to reach the optimal bound of 11, taking 35 hours on a 2006 workstation. Each of these iterations takes one element from the worklist and, if no bounding occurs, adds back to the worklist up to 16 sub-domains (the `compute()` function has four input variables, and SPALTER indiscriminately sub-divides each variable at each step). The worklist grew up to 1.2 million items at one point of the process, taking about 750 MB of memory.

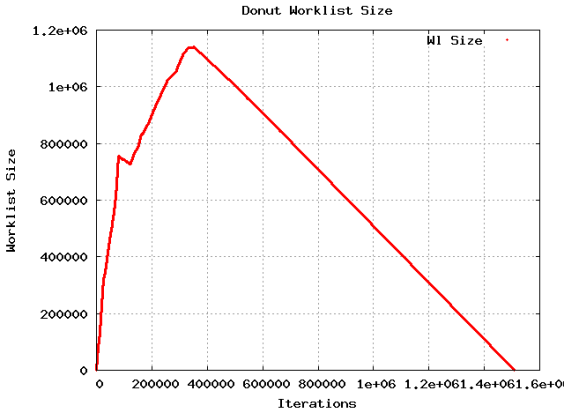


Figure 1. Worklist Size During Analysis of Donut

Figure 1 shows the size of the worklist evolving over time. SPALTER discovers early that 11 is included in the result domain of `interp()`, and sets the goal accordingly. During the first 300000 iterations, the worklist is growing. This does not mean that sub-divisions are not improving precision as intended. Remember that up to 16 new sub-input-vectors are created at each iteration. It is normal that for a long time, fewer on average than 15 of these newly created input vectors can immediately be discarded. This makes the worklist grow. Later in the process, more than 15 on average of the sub-input-vectors fall below the known bound and are discarded immediately, causing the worklist to shrink back until it is empty.

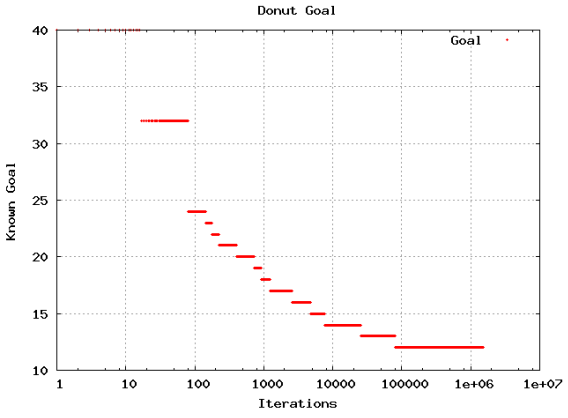


Figure 2. Worklist Entries by their Known Best Value

Figure 2 shows the known best value over logarithmic time. The first known best value is 40: this is what a normal

[4]http://blog.frama-c.com/index.php?tag/donut

run of Frama-C's value analysis returns. With smaller input ranges however, the best known bound decreases as the algorithm progresses. The figure also shows that it becomes increasingly expensive to improve precision; improving the best known bound from 12 to 11 (the optimum) took about 95% of the analysis time. For some usages, the user of SPALTER may want to specify the goal to reach, in order to avoid unnecessary computations. It is also possible to interrupt the algorithm at any time and to take the best computed bound at that point.

VI. Lessons Learned

A. Pathological examples

As the Donut case study shows, it is not necessarily easy to see how quickly SPALTER will provide its answer. It depends on the goal set by the user, on the precision required to reach it, and on how well the analyzed code responds to sub-division.

Consider the expressions `x * x` and `x - x`. For the analysis of `x * x` to achieve maximum precision, it is enough to analyze the cases $x <= 0$ and $x > 0$ independently. This is because interval propagation is optimal for $x * x$ on these two subdomains. Although SPALTER does not recognize directly that a single split at 0 is enough, it approaches the problem by dichotomy. Thus it requires at most a logarithmic number of steps to discover the optimal interval representing the result of `x * x`.

The expression `x - x` on the other hand does not have such an optimal splitting point. In fact, maximum precision can only be achieved by going all the way down to singletons for the abstract values of x, which SPALTER eventually does as it becomes clear that there is no escaping it. In this case SPALTER incurs only a constant-factor overhead in the number of evaluations with respect to the optimal sub-division that would start enumerating singletons immediately. At the same time, SPALTER can provide an approximate but sound bound if it is interrupted early.

B. Finger trees to represent branch-and-bound worklists

The worklist data structure in a branch-and-bound implementation must provide the following three primitives:

1) return one of the subdomains that is currently limiting the answer, and remove this subdomain from the worklist,
2) insert new subdomains into the worklist, and
3) discard all subdomains stored in the worklist that a new bound shows not to be relevant to the computation of the answer.

Operation 1 is made fast by storing the computed maximum together with each subdomain and by keeping the worklist sorted by this maximum. The simplest solution is a sorted list, but this makes operations 2 and 3 linear. A well-suited data structure to represent the worklist is a "finger tree". Then operation 1 takes amortized constant time, while sorted insertion and pruning each take $O(log(n))$. In SPALTER, we use two-three finger trees [6].

VII. Related Work

With (automated or intellectual) analysis at the unit level being standard industrial practice [7]–[9], SPALTER addresses a real demand for more automatic static software analyzers. This is the first time, to the authors' knowledge, that a branch-and-bound type of algorithm has been used to optimize the output of a sound static analyzer for algorithmic programs, as opposed to interval evaluation of mathematical functions.

A parallel can be drawn to a technique [10] used in the static analyzer Astrée where, for a specific alarm, the error values (say, out of bounds indexes for an array access) are propagated backwards in a sound manner along the dangerous traces in order to obtain a superset of the inputs that can cause this alarm for these traces. This superset, presumably smaller than the entire inputs, can then be propagated forward along the dangerous traces. If the smaller size of the inputs causes the alarm not to be emitted, it means that the alarm was false in the first place. If the set of error values is nonempty but smaller than initially, then the method can be iterated. However, this method can reach a fixpoint without having answered the question of whether the alarm was true or false. By contrast, SPALTER never gives up, and it does not rely on backwards propagation (this simplifies the implementation but means that opportunities to reach a straightforward conclusion are wasted). On the other hand, Rival's technique seems geared towards whole-program analysis, whereas our prototype implementation of SPALTER has only shown its usefulness in a unitary analysis context for now. Future work, as outlined in the next section, will aim at improving this aspect.

VIII. Future Work

SPALTER works well for improving the precision of the analysis of individual functions or small call graphs. It is suboptimal for improving the precision of analyses of large programs.

SPALTER sub-divides the values taken by inputs of the program, and it sub-divides before the value analysis plug-in is launched. This works for a single function, and can be adapted to the case of a moderate-sized program that reads inputs along execution. Sub-dividing the function's input is a general, robust approach that can improve precision for numerous target functions. But it can be inefficient. To improve precision within a large program, it can be sufficient, and more economical, to sub-divide the domain of strategically chosen variables at a strategically chosen point of the program. A future version of Frama-C's value analysis plug-in could provide a mechanism to make case analyses more local and greatly lower the required analysis time. A plug-in dedicated to automatically improving the

results of the value analysis could take advantage of the modular nature of Frama-C, and of the existing dependency analyses, to generate the required sub-division and localization instructions in order to further automate the process of analyzing whole programs.

An important virtue of SPALTER is its flexibility in terms of user-specified optimization goals. In principle, SPALTER can find any property of an analyzed function, as long as it can be expressed as a monotonous goal function g mapping the result of Frama-C's value analysis to a totally comparable type. The function is easy to implement in practice thanks to Frama-C's modular architecture. Section IV shows an example of this flexibility by illustrating how SPALTER can be configured to automatically eliminate false alarms. In fact, Frama-C contains analyzers other than the value analysis plug-in that build upon the values it computes. The precision of these derived analyses is often effectively limited by the precision of the value analysis [9]. It would be straightforward to implement a goal function that measures not how well the value analyzer is doing, but how well the derived analyzer performs with the value analysis results, and drives the former to be just precise enough for the latter to give statisfaction.

IX. Conclusion

Frama-C's value analysis does not manipulate intervals only: a C program can have pointers, structs, arrays, and function pointers. In a C program, continuous data is typically represented with IEEE 754 floating-point, and a sound analyzer takes this into account for better and for worse [11], whereas a mathematical function is composed of simpler mathematical operations. A sound abstract interpreter can only map singleton abstract states to singleton abstract states by making strong assumptions regarding floating-point[5] values but in practice SPALTER works very well even with Frama-C options that make the value analysis conservative with respect to floating-point computations, losing the "singleton abstract state" property.

We have validated our approach by analyzing (unavailable) industrial examples where, on its own, Frama-C's value analysis was not precise enough. Although abstract interpreters with more sophisticated and specialized abstract domains can do better than the value analysis' non-relational domains, these other analyzers have their limits as well. Before SPALTER, Andy Sloane's Donut was, to our knowledge, beyond the limit for automatic analyzers. SPALTER verifies the Donut with about the same computational effort that was required in the previous semi-automated case study, but requires no guidance, apart from slicing the `compute()` function from the original source to make unit analysis possible.

[5]http://blog.frama-c.com/index.php?post/2011/11/18/Analyzing-single-precision-floating-point-constants

Additional material, including the source code for the SPALTER prototype, can be found at http://www.sts.tu-harburg.de/research/spalter.html.

References

[1] G. Canet, P. Cuoq, and B. Monate, "A Value Analysis for C Programs," in *Ninth International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*, 2009, pp. 123–124.

[2] P. Cuoq, B. Yakobowski, and V. Prevosto, "Frama-C's Value Analysis Plug-in," Available at http://frama-c.com/download.html, 2013.

[3] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2009.

[4] P. Cuoq and J. Signoles, "Experience report: OCaml for an industrial-strength static analysis framework," in *International Conference on Functional Programming*, 2009, pp. 281–286.

[5] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang, "Testing static analyzers with randomly generated programs," in *Proceedings of the 4th international conference on NASA Formal Methods*, ser. NFM'12, 2012, pp. 120–125.

[6] R. Hinze and R. Paterson, "Finger trees: a simple general-purpose data structure," *Journal of Functional Programming*, vol. 16, no. 2, pp. 197–217, Mar. 2006.

[7] H. Delseny, "Formal Methods for Avionics Software Verification," Open-DO Conference, presentation available at http://www.open-do.org/2010/04/28/formal-versus-agile-survival-of-the-fittest-herve-delseny/, 2010.

[8] D. Pariente and E. Ledinot, "Formal Verification of Industrial C Code using Frama-C: a Case Study," in *International Conference on Formal Verification of Object-Oriented Software (FoVeOOS)*, 2010, pp. 205–219.

[9] D. Delmas, P. Cuoq, V. Moya Lamiel, and S. Duprat, "Fan-C, a Frama-C plug-in for data flow verification," in *Embedded Real time Software and Systems*[2], 2012.

[10] X. Rival, "Understanding the origin of alarms in ASTRÉE," in *12th Static Analysis Symposium (SAS'05)*, ser. LNCS, vol. 3672. London (UK): Springer, Sep. 2005, pp. 303–319.

[11] D. Monniaux, "The pitfalls of verifying floating-point computations," *ACM Transactions on Programming Languages and Systems*, vol. 30, no. 3, pp. 1–41, 2008.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

PtrTracker: Pragmatic Pointer Analysis

Sebastian Biallas
RWTH Aachen
biallas@embedded.rwth-aachen.de

Mads Chr. Olesen
Aalborg University
mchro@cs.aau.dk

Franck Cassez
NICTA and UNSW
franck.cassez@nicta.com.au

Ralf Huuck
NICTA and UNSW
ralf.huuck@nicta.com.au

Abstract*—Static program analysis for bug detection in industrial C/C++ code has many challenges. One of them is to analyze pointer and pointer structures efficiently. While there has been much research into various aspects of pointer analysis either for compiler optimization or for verification tasks, both classical categories are not optimized for bug detection, where speed and precision are important, but soundness (no missed bugs) and completeness (no false positives) do not necessarily need to be *guaranteed.

In this work we present a new pointer analysis tool for C/C++ code. The tool introduces the notion of heap graphs that are inspired by shape analysis without the computational overhead, but also without the verification soundness guarantees. We explain the underlying ideas and that it lends itself to a fast, modular and incremental analysis, features that are essential for large code bases.

To demonstrate the practicality of the solution we integrate the pointer analyzer into the C/C++ bug checking tool Goanna. We show that run-times of the new analyzer are close to compile times on large code bases and, most importantly, that the new solution is able to reduce false positives as well as to detect previously unknown pointer bugs in the Git source code.

I. Introduction

Pointer Analysis. A pointer analysis (or alias analysis) determines a set of all possible (symbolic) memory locations a pointer variable might point to during execution. The result of a pointer analysis is for instance instrumental in the *optimization pass* in compilers (e.g. to optimize register reloads) and also in the *static analysis* (verification) of programs (e.g. to check for possible NULL pointer dereferences.) However, the type pointer analysis required depends on the subsequent phase: the optimizing pass of a compiler uses the pointer analysis to optimize register reloads; the analysis must be fast, conservative but not necessarily precise. For verification purposes, the pointer analysis is used to improve the accuracy of the static analysis phase and to remove *false positives* (spurious warnings). A conservative (sound) static analysis usually assumes that every pair of pointers can alias each other and this causes some false positives. To remove the false positives, the pointer analysis must be more precise than for optimization purposes and thus it is usually slower and more memory intensive. Since pointer analyses play such a central role, a multitude of different analyses with different scopes and granularity have been extensively studied in the past [1] and this is still an ongoing research topic [2].

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

Pointer Analysis for Static Analysis. It is now well established that *sound* static analysis is time-consuming and generates a lot of false positives. If we drop the soundness requirement for a *best effort* to find bugs, we can significantly gain in efficiency and accuracy [3].

In this paper, we consider pointer analysis in the context of bug finding without the requirement that our bug finding tool be sound (no bug is missed, i.e. no false negatives) or complete (every warning issued by the tool is true, i.e. no false positives). In this respect our pointer analysis is the middle-ground between a fast and coarse optimization-oriented pointer analysis and a sound and computationally expensive verification-oriented pointer analysis.

We distilled three key requirements for PtrTracker to make it applicable in the static analysis of large code bases composed of thousands of files and with millions of lines of codes: (1) the pointer analysis must be fast (less than the compile time) while still being precise; (2) the pointer analysis must be modular and incremental; modular means that we can analyze a function f without knowing the calling context or the callees and build a summary for the function f; incremental means that we can later enrich the summary of f if in the course of the static analysis we later analyze a file containing a function g called by f and collect some useful information (summary) for g; (3) in addition, it should be able to integrate with an existing tool with minimal adjustments to incorporate our pointer analysis results. In our case, we want to integrate in the tool Goanna [4], a state-of-the-art bug finding tool for C/C++ which yet lacks a fast and precise pointer analysis module.

An example of a bug we would like to find is represented by this C fragment:

```
void foo(struct bar *a, struct bar *b) {
  b = a;
  a->f = 0;
  42 / b->f;
}
```

A classic division by zero bug is masked by pointer indirection. If we do not detect/assume that a and b can alias, this bug will remain undetected. In real-world cases the statements are spread out with hundreds of lines in between, including branching or function calls.

Related Work. The area of pointer (alias) analysis has been researched extensively [1], [5], [2], mostly targeting compiler

optimization techniques, and focusing on computing *may-point-to* information. For static program analysis (i.e. bug finding), *must-point-to* information is much more valuable than *may-point-to* information, as it enables to decrease the false-positives rate. There is a substantial body of work addressing pointer analysis in the context of static analysis [6], [7] but the analysis times reported in the experiments are unlikely to scale to large code bases. Our own previous attempt at integrating pointer analysis in Goanna [8], resulted in a static analysis time increased by a factor of three to six, which is too slow in practice. PtrTracker is inspired by shape analysis [9]. Shape analysis is claimed to scale up [10], still the analysis times are far from those expected from an industrial-grade bug finding tool.

Our Contribution. We present PtrTracker, a tool that annotates all pointer dereferences in a program with a set of possible pointees, which themselves are normal variables. This information is gathered in a flow-sensitive way while we lazily introduce new symbolic names for traversed data structures. As we will show, this approach is very fast while offering the required precision to find bugs and suppress spurious warnings depending on alias information.

II. Preliminaries and the Existing Goanna Architecture

Tool Overview. PtrTracker builds on and integrates in Goanna [4], [11], an automata-based static analyzer for detecting software bugs, memory leaks and security vulnerabilities in C/C++ programs. Goanna combines static analysis techniques (e.g. abstract interpretation) with CTL model-checking. The tools' high-level architecture is depicted in Figure 1. Goanna is used as a drop-in replacement for the compiler (e.g. `gcc`), for easy integration into existing tool chains. It takes as input a project composed of C/C++ files, a set of *checks* which are properties that characterize the presence of bugs (if the property is not satisfied there is a bug) and an optional database that records (previously analyzed) functions' summaries. Goanna analyses programs file by file and cross-relates information. For each file, it first parses the file and builds an Abstract Syntax Tree (AST) in the form of an XML document; this XML document is central in the architecture. The XML is subsequently annotated by a *value interval analysis*, where each AST node is annotated with interval information (e.g. range of some integer variables.) This annotated AST, AST_2 is converted to a Control-Flow Graph (CFG). The last stage of the analysis is performed by a model-checker that takes as input a set of CTL formulae (the formal definitions of the checks), the CFG and the optional database of summaries. In case a formula is not satisfied by the CFG, the model-checker generates a warning and returns a witness trace (sequence of instructions) to explain the bug. The model-checker features state-of-the-art abstraction refinement techniques to remove spurious bugs (that are artifacts of the abstract CFG.) Each trace (counter-example) returned by the model-checker is analyzed (False-Positive Elimination) using an SMT-solver [12] and if it is spurious (infeasible in the concrete C/C++ semantics), the CFG is refined accordingly and the model-checker run again on the refined CFG. This enables us to remove spurious counter-examples (false-positives) and get more accurate analysis results. Notice that for each file, for each function in the file, a *summary* of the analysis can be stored in a database and can be re-used in subsequent analysis or in the analysis of files that are processed later. If the optional database Database_1 is used, it is enriched after each function analysis and updated to become Database_2.

Variables Binding in Goanna. Goanna has limited capability w.r.t the analysis of variables' ranges (and pointers): some variables are ignored if not *bound* to explicitly *declared* variables. To illustrate these concepts, consider the expression `a + b` where `a` and `b` are integers. The AST's XML representation of the expression is

```
<Op2 op="Add">
  <Ref name="a" binder="7" />
  <Ref name="b" binder="8" />
</Op2>
```

An `Op2` node is any binary operation, and a `Ref` node is a variable reference, where the *binder* uniquely represents the variables in the program (in cases a local variable overlaps a global variable.) A binder refers back to an XML node (7) containing the declaration of this variable. The interval analysis determines some possible ranges for the variables and this information is stored in the *annotated* XML AST_2: `min` and `max` are new attributes added to the relevant nodes. For instance if we infer that `a` is in the interval $[1, 2]$ and `b` in $[42, 42]$, the node `Op2` is annotated with $[43, 44]$, because the result of the expression `a + b` is in $[43, 44]$.

```
<Op2 op="Add" min="43" max="44">
  <Ref name="a" binder="7" min="1" max="2" />
  <Ref name="b" binder="8" min="42" max="42" />
</Op2>
```

The current interval analysis only knows about *binders*, and because binders only exist for variables that are *syntactically* declared, some variables are ignored in the interval analysis. As an example consider a struct pointer dereference `p->f`, which is represented in the AST's XML as:

```
<Op2 op="Arrow">
  <Ref name="p" binder="7" is_pointer="true" />
  <Field name="f" />
</Op2>
```

Because `p->f` is not explicitly *syntactically* declared, the interval analysis (and subsequent steps) ignores it. Notice that it does not matter whether field `f` above is a basic type or a pointer. It is ignored in the interval analysis. To be able to reason about pointers, we have to overcome the previous limitation imposed by explicitly declared binders. The solution to this is to introduce new variables (representing memory locations), which are binders, for syntactic items such as `p->f`. Notice that we do this in a *lazy* manner, only when such a binder is needed.

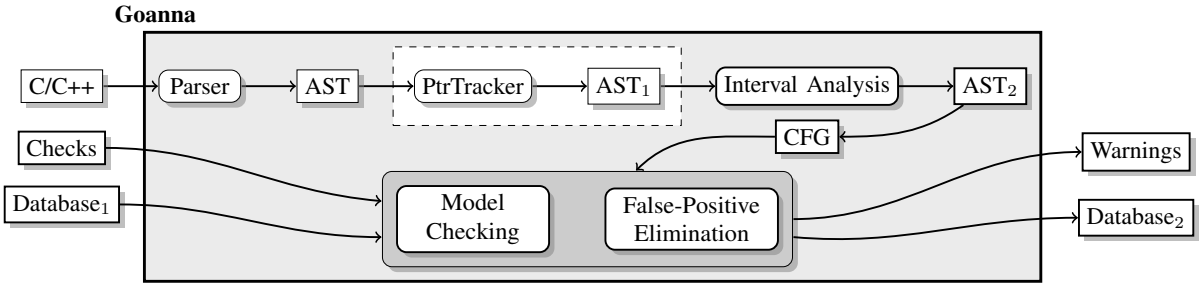


Figure 1. The architecture of Goanna, with the integration of PtrTracker highlighted.

III. PtrTracker: Pointer Analysis with Heap Graphs

A. Pointer Operations in C

The foundation for our pointer analysis builds on the facts that (1) C pointers are either an l-value (something on the left-hand side of an assignment) or an r-value (something on the right-hand side of an assignment) and (2) all pointer operations in C can be encoded with three basic operators:

1) addr: taking the address of an l-value and returning it as an r-value, e.g. `&p`,
2) deref: dereferencing an address (an r-value) and returning the l-value it points to, e.g. `*p`,
3) value-of: doing a deref followed by a addr operation. This is a derived operation, and needed in cases of pointer arithmetic and pointer assignments such as `p = q` where `q` is an l-value, but needs to be turned into an r-value. For our analysis, this is semantically equivalent to `p = &*q`.

The relationship between the pointer operations and the basic syntactic C expressions is rather straightforward. Figure 2 shows the semantics of the basic operators. Array references such as p[i] are handled as *(p+i) according to the C standard.

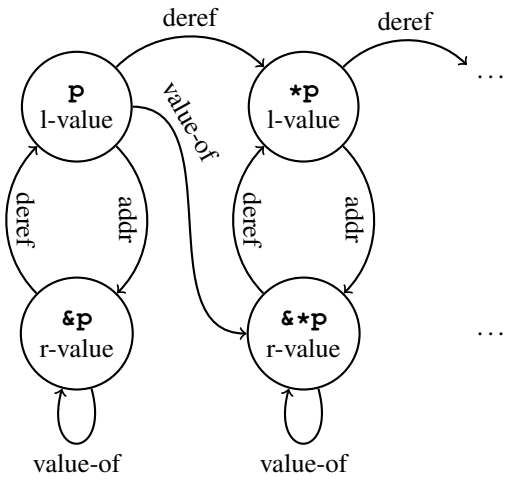


Figure 2. Pointer Operations in C

B. Heap Graphs

PtrTracker performs an abstract interpretation over the structure of the *heap*, where the underlying abstract domain is a *heap graph*. An example is depicted in Figure 3. Each node of the heap graph represents a contiguous chunk of memory (an "object" in C99 terms). Moreover, each node is made up of one or more memory cells, i.e. the memory node represents a `struct`, with a memory cell for each field. If the cell is of a pointer type (indicated by the bullets) it has one or more out-going edges, representing the memory cells it is possibly pointing to.

Note, pointers can point inside `structs`, `i` points to `a->c`. Non-`struct` objects, such as an `int*` pointer, are equivalent to a `struct` with a single field and are covered by the previous representation.

The abstract transformers for the heap graph domain are expressed in terms of the basic operations described in subsection III-A. Abstract transformers are currently implemented for C and a large part of C++.

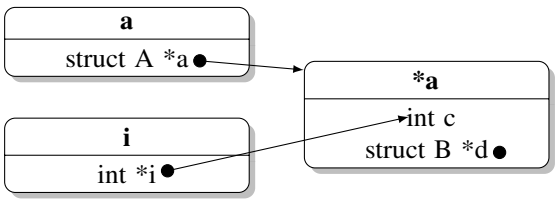


Figure 3. A heap graph

Assignments are performed as a *strong update* if we only have one l-value. In this case, we remove the existing arrow(s) and add arrows to the r-values. If, on the other hand, we have multiple l-values, we perform a *weak update* adding new arrows to all r-values.

C. Inter-procedural Analysis

Goanna itself supports inter-procedural analysis. Because functions can be analyzed in any order (Goanna also features concurrent/multi-core analysis) PtrTracker assumes as a minimal base:

A1 *deep no-aliasing*: function parameters do not alias each other, and following parameter pointers also

do not lead to aliases to other parameters. As an example, consider the function `void foo(struct bar *a, struct bar *b);` we assume that `a` and `b` do not alias when we enter the function; the same holds for any pointer field `f` `a->f != b->f` for any sequence of dereferences.

A2 *No-aliasing on global variables*: Parameters do not alias global variables.

A3 *Side-effect free function calls*: Function calls do not modify parameters and always return new memory cells.

These assumptions are in general unsound, but they are essential for the pointer analysis to derive practical information. For instance without A3, if we analyze a function f that calls g and g has not yet been analyzed, we cannot assume anything of the (pointer) variables that are passed to g, nor of the global variables. This implies, after the call to g, we have to assume that two pointers can alias each other or a global variable. Note, if during the course of the analysis we discover that two parameters of f can alias each other, we can trigger a re-analysis of f under the revised assumption collecting *contextual* knowledge for the behavior of f. This feature will be available soon in PtrTracker.

IV. Integrating PtrTracker and Goanna

The goal of introducing a pointer analysis in Goanna is to eliminate both false positives and false negatives. An essential part is that the analysis can integrate such that the key features of the tool are preserved: performance, ability to run as a compiler drop-in, re-use of existing checks and applicability to real-world programs. To be useful for bug-hunting the pointer analysis needs to be flow-sensitive, because flow-insensitive information would mask any manipulation done to pointers — which is counter-productive if one wants to detect bugs in pointer manipulation.

A. Implementation Details

There is a number of implementation details in PtrTracker which differs from a standard abstract interpretation or shape analysis, because of the special setting. Firstly, the analysis does *not* compute a fixed-point because this is unnecessary. In shape analysis summary nodes would be used to ensure termination (due to recursive data structures such as linked lists), but because we are more interested in must-information summary nodes are of little value.

Additionally, any syntactic element that does not point to any one unique memory location is also of little interest since this will disallow strong updates. Consider the `C` fragment:

```
struct list *a;
struct list *cur;
for (cur=a; cur!=NULL; cur=cur->next){
  foo(cur, a);
}
```

Here the best information we can derive for the call to `foo` is that `cur` points to some element of the list starting at `a`. However, we can derive that `a` always points to the same memory cell. Because of these observations, it is sufficient to recurse into recursive data structures until enough information has been gained such that loop invariants have been learned, and verified to be invariants.

The analysis is implemented in a lazy way, such that memory nodes are only added to the heap graph if needed. As such a `struct` with many members is only represented by the members referenced in the function currently being analyzed. Because the analysis computes information for each program point there is a lot of redundancy in the analysis result: we exploit this redundancy by storing the memory nodes in a flow-insensitive way, and only storing the edges ("points-to" information) of the graph for each program point. Both of these optimizations are essential for the performance of the analysis.

B. Arrays and Pointer Arithmetic

In C99, pointer arithmetic can only be used to move a pointer within a chunk of memory. That is, dereferencing a pointer outside "its" object is an error and should ideally be detected. In our analysis, we mark all nodes which are accessed via pointers which were subject to pointer arithmetic as containing arrays. Here, the index operator [] is handled as a special case of pointer arithmetic as described in Sect. III-A. Afterwards, we assume that the pointer aliases the complete array represented by the node. Hence, we can ignore pointer arithmetic altogether.

In a subsequent analysis, however, all pointer offsets can be recalculated using an integer analysis taking the pointer arithmetic into account. If we have must-alias information, we can infer tight bounds on the stride of memory that is accessed using this pointer.

C. Re-Using Existing Checks

To a large extent the existing Goanna checks were re-used without modification. In a few places some patterns were generalized to not look for explicit variable references, but also looking at other AST nodes that were annotated with a binder.

Some checks were rewritten to gain additional accuracy by taking pointer information into account: freeing allocated memory twice, losing the last reference to allocated memory, returning pointers to the local stack, and dereferencing a pointer that is possibly NULL. These rewritten checks improved both the rate of false positives and false negatives, compared to the old checks they replaced.

D. Path-Sensitive Checks

Our analysis determines alias information in a flow-sensitive way. This information, however, is not directly used to generate warnings. Instead, we employ a model-checker to verify certain properties of the program. This method can, in principle, recover the path-sensitivity for certain checks. Take, e. g., a possible double-free, which manifests itself as two calls to the free function where the argument points to the same object in both cases. In this case, our model-checker will determine a path containing two successive calls to the free function.

Instead of directly reporting this as an error, we first check whether this path is actually feasible. If the path is not feasible (e. g., it depends on two consecutive if-statements with contrary conditions), the false positive elimination module [12] can eliminate it and refine the CFG. When we analyze the refined CFG we effectively regain the path-sensitive information.

V. Case Study

We have tried our prototype on a number of open source projects, to make sure that it supports a sufficient portion of C to be useful. We here report on running Goanna+PtrTracker on the well-known Git project[1].

The runtime of the stock Goanna on a single core without PtrTracker was $26m52s$, which produced 266 warnings. Enabling PtrTracker slightly increased the runtime to $27m35s$ (an increase of 2.6%), while producing 271 warnings. Because Goanna is a drop-in replacement for the compiler in the build system both run times include the overhead of the actual compilation.

The 5 additional warnings are classified as follows:

- 1 true positive about the address of a local being stored in a global variable.
- 1 true positive about an arithmetic shift resulting in undefined behaviour, see Figure 4, found by an existing check being able to reason about pointers as additional variables.
- 1 true positive about a NULL dereference, involving interprocedural reasoning.
- 2 probably false positive about out of bounds array accesses using enum variables.

```
struct histindex index;
int sz;
...
index.table_bits = xdl_hashbits(count1);
sz = index.records_size = 1 <<
  index.table_bits;
...
```

returns [1,32]

Figure 4. Example of bug found in Git: "interval $[1, 32]$ is out of range of the shift operator"

VI. Conclusion & Lessons Learned

We have integrated a new pointer analysis tool, PtrTracker, in the industrial-grade static analyzer Goanna. PtrTracker is a very pragmatic pointer analysis that satisfies the requirements of being very scalable, generally applicable and easily integrateable. The analysis and design choices are interesting from a practical perspective: it gives an alternative to more traditional pointer-analyses in terms of usefulness for bug finding. While designing and implementing PtrTracker, we learned the following lessons:

- Real-world C programs contain complex pointer expressions (sometimes arising from pre-processor inlining) which must be handled compositionaly. We were able to reduce all expressions to the operation depicted in Fig. 2, which is key to the design of our analysis.
- A pragmatic approach to handle parameters and function calls is necessary (our assumptions A1–A3). Otherwise, a pointer analysis will likely generate no helpful results at all (i.e., aliasing everything) resulting in too many false positives.
- A multi-pass analysis and contextual analysis can be used to refine/invalidate assumptions, propagating alias information across function calls.
- Must-alias information, i.e. pointer dereferences with a fixed target, is much more helpful for bug-detection than may-alias information, and effectively rules out false positives.
- Pointer analysis is a complex and sometimes delicate topic. We keep the pointer analysis separate, and provide a set of possible variables for each pointer dereference, which helps tremendously in reducing the complexity.

[1] http://git-scm.com

References

[1] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 54–61.

[2] U. Khedker, A. Mycroft, and P. Rawat, "Liveness-Based Pointer Analysis," in *Static Analysis Symposium*. Springer, 2012, pp. 265–282.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[4] R. Huuck, A. Fehnker, S. Seefried, and J. Brauer, "Goanna: Syntactic Software Model Checking," *Automated Technology for Verification and Analysis*, pp. 216–221, 2008.

[5] B. Hardekopf and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," in *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 290–299.

[6] M. Buss, D. Brand, V. Sreedhar, and S. A. Edwards, "A novel analysis space for pointer analysis and its application for bug finding," *Science of Computer Programming*, vol. 75, no. 11, p. 921, 2010.

[7] V. B. Livshits and M. S. Lam, "Tracking pointers with path and context sensitivity for bug detection in C programs," in *ESEC / SIGSOFT FSE*, 2003, pp. 317–326.

[8] J. Brauer, R. Huuck, and B. Schlich, "Interprocedural Pointer Analysis in Goanna," *Electronic Notes in Theoretical Computer Science*, vol. 254, pp. 65–83, 2009.

[9] R. Wilhelm, M. Sagiv, and T. Reps, "Shape Analysis," in *Compiler Construction*, D. Watt, Ed. Springer Berlin Heidelberg, 2000, vol. 1781, pp. 1–17.

[10] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn, "Scalable Shape Analysis for Systems Code," in *Computer Aided Verification (CAV)*, 2008, pp. 385–398.

[11] M. Bradley, F. Cassez, A. Fehnker, T. Given-Wilson, and R. Huuck, "High Performance Static Analysis for Industry," *Electronic Notes in Theoretical Computer Science*, vol. 289, pp. 3–14, 2012.

[12] M. Junker, R. Huuck, A. Fehnker, and A. Knapp, "SMT-based false positive elimination in static program analysis," in *International Conference on Formal Engineering Methods (ICFEM)*, 2012, pp. 316–331.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

Tracing with a Minimal Number of Probes

David Baca
Czech Technical University, Electrical Engineering
&
Freescale Semiconductor

***Abstract*—When a program execution must be observed, probes are injected to trace its execution. Probes are intrusive, causing execution overhead and modifying a program's real time properties. It is therefore desirable to place them efficiently, so that their negative effects are mitigated. This paper presents a new algorithm for efficiently allocating probes at vertices in the control flow graph of a program in a way that makes it possible to reconstruct program traces. Despite of the intractability of finding minimal vertex probe placement, a fact proven in this paper, the presented algorithm is capable of identifying a locally minimal set of vertices, which reduces the set of needed probes beyond the current state-of-the-art. First, the paper shows the advantages of placing probes at vertices, rather than edges. Next, the known methods for reducing the set of probes needed to *profile* execution are proven to be applicable to reduce the set needed to *trace* execution. Further, the new algorithm is described along with the algorithm for reconstructing the executed path from a trace file. In addition, the new algorithm is combined with methods for placing profiling probes to reduce the execution overhead even further. Finally, the new probe placement algorithm is compared with known algorithms to elaborate on its benefits.**

I. Introduction

A. Paper aim

This paper addresses placement of probes for collecting execution traces. Tracing logs executed path information and reconstructs the path executed during a program run. The information logging is performed by probes injected in the program; the injection process is called program instrumentation.

A trace is collected during an execution of a program. The probes log the tracing data as they are traversed by the execution. The probes execution modifies the program real-time behavior and also causes an unpleasant overhead by extending the execution time. Furthermore, the size of log file can be also important. Therefore, it is desirable to reduce the set of probes, so a trace can be still reconstructed but the probe side-effects are mitigated. Moreover, placing probes to less executed spots can further reduce the logging overhead.

The objectives of the presented work had been defined from the perspective of tracing embedded programs. Thus, the main objective is to reduce as much as possible the number of probes to limit intrusion. The secondary objective asks for reducing the trace collection overhead by allocating the probes to program locations that are executed less. This objective is set not to only cut down the execution time of instrumented programs but also to reduce the log file size, which is usually stored on embedded systems in their free memory.

This paper presents a new approach for probe allocation that substantially reduces the number of probes. The reduction is superior to any other previously known methods for determining efficient probe placement. Moreover, the paper also proves that finding the minimal set of probes is an NP-complete problem.

978-1-4673-5739-5/13/$31.00 © 2013 IEEE

B. Related Work

Numerous work has addressed the allocation of probes to bring down both the number of probes and the execution overhead they produce [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. However, those works mostly focus on profiling.

Unlike tracing, profiling measures execution frequencies of program control flow elements - a profile. While reconstructed path by tracing yields also a profile, profiling data cannot be used in general to reconstruct executed paths. Nevertheless, [5], [6] has shown that profiling probe placement to edges can be also used for tracing probe placement. Furthermore, path profiling [14], [7] measures frequencies of paths where each path is identified by its number computed along traversed probes. Yet, the probes are allocated to edges where profiling counters are placed.

Ramamoorthy, Kim, and Chen, [1] address the problem of finding the minimal set of edge traversal markers (probes) for acyclic control flow graphs. They define a condition when a set of edge probes is minimal for an acyclic control flow graph and sketched an algorithm for finding the minimal set. Yet, S. Maheshwari has proved that this problem is NP-complete [15].

Larus and Ball published several papers on efficient profiling and tracing [5], [16], [14], [17]. However, they always rather focused on probe allocation to edges (branches) when tracing was considered. In particular, they argue that edge probes are more suitable for profiling and they show that a profiling edge probe placement can be reused for tracing.

II. Background

Programs written in an imperative language consist of procedures and each procedure control-flow can be represented in a form of a control flow graph. A procedure execution start is symbolized by vertex $ENTRY$ and vertex $EXIT$ represents the execution exit. It is always possible to model the control flow with a single $EXIT$ vertex only. If a procedure has multiple exit statements, then the basic block ending with the exit statement has on outgoing edge to $EXIT$.

In this paper, only intra-procedure control flow is considered. This means that any call to a procedure including recursive calls is just a statement and the execution is assumed to return the very next statement after the call.

This section defines the terminology and formalisms used in modeling control flow and determining probe placement. In particular, the initial part defines and formally describes control flow graphs. Furthermore, the next subsection reviews currently known methods for probe placement for tracing and defines the problem of finding efficient probe placement formally.

A. Formal Terminology

A control flow graph is a *directed* graph where branches (edges) represent the flow of the execution between basic blocks (vertices). A basic block is the maximal, totally ordered sequence of statements or instructions that can be entered only at the beginning and exited only at the end.

Formally, control flow graph as a directed graph is a tuple $G = \langle V, E\rangle$ where $V, V = V(G)$, is a set of vertices and $E, E = E(G)$, is a set of directed edges. An edge e is an ordered pair of vertices denoted $v \rightarrow w$ meaning that execution can flow from the end of the basic block v to the start of w. For edge $e = v \rightarrow w$, vertex v is the edge start denoted $start(e)$ and vertex w is the edge end denoted $end(e)$. Further, edge e is an *incoming* edge of vertex w and an *outgoing* edge of vertex v. Also, given the edge e, v is a *predecessor* of w and w is a *successor* of v.

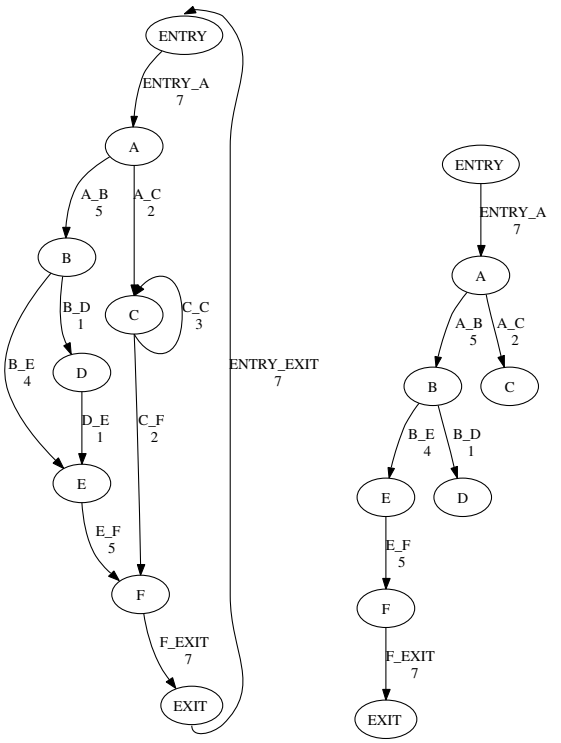
Each vertex $v, v \in V(G)$ has defined

- its *label*, the unique identifier, label, $label(v)$ such that for any $u \in V(G)$ $label(u) = label(v)$ if and only if $u = v$,
- its successors $succ(v, G)$ where $\forall u \in V(G)$ if $v \rightarrow u \in E(G)$ then $u \in succ(v)$, and
- its predecessors $pred(v, G)$ where $\forall u \in V(G)$ if $u \rightarrow v \in E(G)$ then $u \in pred(v, G)$.

A *path* in a directed graph G is a sequence $v_0, e_1, v_1, \ldots, e_n, v_n$ where $v_i \in V(G), e_i \in E(G)$, and for each edge $e_i, e_i = v_{i-1} \rightarrow v_i$. Such a path is directed and a path in a directed graph is always assumed to mean a directed path. A path is *undirected*, and such is stated explicitly for a directed graph, if for each e_i either $e_i = v_{i-1} \rightarrow v_i$ or $e_i = v_i \rightarrow v_{i-1}$. A *cycle* is a path such that $v_0 = v_n$ and for any other two vertices from the path $v_i, v_j,\ i \neq j,\ v_i \neq v_j$.

A control flow graph has two special vertices called $ENTRY$ and $EXIT$. $ENTRY$ is the very first vertex of the graph symbolizing the entry point of the execution. $EXIT$ is the last vertex visited on each finite execution path. Thus, a control flow graph is a directed graph tuple $G = \langle V, E\rangle$ where $ENTRY, EXIT \in V$. Every vertex is reachable from $ENTRY$ and $EXIT$ is reachable from every vertex in the graph.

Each edge can be assigned a value, usually a natural number or a real number. This value is called edge weight or edge cost. Formally, weighting of a graph $G = \langle V, E, \omega\rangle$ is a mapping $\omega : E \mapsto \mathbb{N}$ or $\omega : E \mapsto \mathbb{R}$. Usually, the weighting denotes the number of times an edge is traversed during execution. Since a negative traversal count is impossible, $\omega : E \mapsto \mathbb{R}^+$, $\mathbb{R}^+ = \{x | x \in \mathbb{R}, x \geq 0\}$.



(a) A control flow graph

(b) Maximum spanning tree

Fig. 1. A control flow graph example with its maximum spanning tree

If edge weights are traversal counts, they shall follow the Kirchhoff's flow law: For each vertex $v, v \in V(G)$ the sum of weights of incoming edges must be equal to the sum of weights of outgoing edges.

$$\forall v, v \in V, \sum_{\forall e, end(e)=v} \omega(e) = \sum_{\forall e, start(e)=v} \omega(e) = \omega(v)$$

Those sums are then regarded as vertex weight or vertex cost.

A *tree* is a graph where there is always just one undirected path between any two vertices. In other words, there is no undirected cycle. A *spanning tree* of a directed graph $G = \langle V, E\rangle$ is a subgraph $G_S = \langle V_S, E_S\rangle$ such that $V_S = V$, $E_S \subseteq E$, and there is always one and only one undirected path between any two vertices. A directed graph has usually many spanning trees. A minimum spanning tree is a spanning tree whose sum of edge weights is minimal. Conversely, a maximum spanning tree is a spanning tree whose sum of edge weights is maximal. There can be more than one maximal (minimal) spanning tree. See Figure 1. Algorithms for finding a minimal spanning tree are described in [18], [19].

B. Probe Placement for Tracing

A program trace with respect to a procedure CFG is a sequence of basic blocks and edges totally ordered by their execution order. Also, a sequence of executed basic blocks or

a sequence of executed edges is a program trace as it uniquely describes the execution order.

The simplest way for generating a trace is to place a probe at each basic block or each edge. A probe writes a unique token (label) to a trace file. The program trace is then reconstructed from the trace file.

The tracing probes are intrusive, modify real-time properties of the executed program, and cause significant execution overhead. Many works have attempted to reduce the the number of probes and the associated overhead of tracing.

Ramamoorthy, Kim, and Chen [1] give a condition when a set of edge probes (traversal markers) is minimal in the case of an acyclic control flow graph. They define a *uniconnected subgraph* of acyclic control flow graph, denoted *U-subgraph*. If $G = \langle V, E\rangle$ is the control flow graph, then a U-subgraph is $G_U = \langle V, E_U\rangle$ with at most one directed path between every pair of vertices in V. A *maximal U-subgraph* is a U-subgraph for which $|E_U|$ is maximal. Given a maximal U-subgraph $G_{Umax} = \langle V, E_{Umax}\rangle$, the minimal set of edge probes for tracing is $\mathcal{P}_e = E - E_{Umax}$. However, finding the maximal U-subgraph is an NP-complete problem as is the problem of finding the minimal set of probes placed to edges [15].

Larus and Ball [5], [6] show that any minimal probe placement to edges for profiling can be used also for tracing. Profiling measures the execution frequencies of basic blocks and edges. As with tracing, profiling faces the similar effects of program instrumentation with probes and therefore efficient probe placement is sought as well.

The simplest placement for profiling is to place probes (counters) to every control flow element. Yet, it is more beneficial to place counters to edges as an edge profile yields a basic block profile, but the opposite doesn't hold. Nevertheless, if basic block frequencies are the only interest, vertex counters are also used.

Given a control flow graph G, $G = \langle V, E\rangle$, $ecount(G, \mathcal{P}_e)$ denotes the problem of measuring edge frequencies with edge counters $\mathcal{P}_e \subset E$ and $vcount(G, \mathcal{P}_v)$ denotes the problem of measuring vertex frequencies with vertex counters $\mathcal{P}_v \subset V$. For the sake of completeness, for $ecount(G, \mathcal{P}_v)$ no $\mathcal{P}_v$ need exist and $vcount(G, \mathcal{P}_e)$ can be solved by $\mathcal{P}_e$ satisfying $ecount(G, \mathcal{P}_e)$; however, finding a minimal solution to $vcount(G, \mathcal{P}_e)$ may be NP-complete per [6].

For both $ecount(G, \mathcal{P}_e)$ and $vcount(G, \mathcal{P}_v)$ minimal solutions can be found. The solutions are based on constructing spanning trees as described in [2] and [3], respectively. Yet for $vcount(G, \mathcal{P}_v)$, the spanning tree is constructed for a reduced control flow graph where edges represent execution flow through vertices in the original graph G. In both cases, there are as many minimal probe placements as there are spanning trees.

Optimal solutions to $ecount(G, \mathcal{P}_e)$ and $vcount(G, \mathcal{P}_v)$ can be found by constructing maximal spanning trees. Given a weighting ω for graph G, the maximal spanning tree eliminates the most often executed edges or vertices, respectively. Hence, an optimal probe placement is one of the minimal probe placements. Since an optimal placement $\mathcal{P}_e$, $ecount(G, \mathcal{P}_e)$, has often lower sum of weighting than an optimal placement $\mathcal{P}_v$, $vcount(G, \mathcal{P}_v)$ and an edge profile yields vertex profile [5], allocating probes to edges is considered as the preferred alternative.

Any minimal probe placement $\mathcal{P}_e$ for $ecount(G, \mathcal{P}_e)$ can be also used for allocating tracing probes as proved by Larus and Ball in [5]. Yet, this is indirectly obvious from [1] as well. If E_S are the edges of the spanning tree of G, $G = \langle V, E\rangle$, then $\mathcal{P}_e = E - E_S$ are the probed edges. $\mathcal{P}_e$ is then a probe placement suitable for tracing because the spanning tree is a U-subgraph, $E - E_U$ is suitable placement for tracing for acyclic graph, and all edges creating cycles do not belong to the spanning tree.

Formally, given a control flow graph G, $trace(G, \mathcal{P})$ denotes the tracing problem as a problem of finding probe placement $\mathcal{P}$ such that every executed path, every program trace, can be reconstructed from a trace file generated by probes placed at $\mathcal{P}$. For the sake of simplicity, a probe placement is either solely to edges or solely to probes. Set of edges $\mathcal{P}_e$ is the placement solution if $\mathcal{P}_e \subset E(G) \wedge trace(G, \mathcal{P}_e)$. Similarly, $\mathcal{P}_v$ is the placement solution of probe placement at vertices if $\mathcal{P}_v \subset V(G) \wedge trace(G, \mathcal{P}_v)$. Furthermore, a probe placement $\mathcal{P}$ is minimal if

$$trace(G, \mathcal{P}) \wedge \nexists \mathcal{P}', trace(G, \mathcal{P}'), |\mathcal{P}'| < |\mathcal{P}|.$$

Similarly, a probe placement $\mathcal{P}$ is optimal if

$$trace(G, \mathcal{P}) \wedge \nexists \mathcal{P}', trace(G, \mathcal{P}'), \sum_{\mathcal{P}'} \omega(p) < \sum_{\mathcal{P}} \omega(p)$$

Nowadays, placing probes on edges is general practice when both minimization and optimization is targeted [14], [11], [10]. Moreover, Ball and Larus [5] claim without any proof that the tracing problem solution with edge probes yields significantly less time and storage overhead. Their conclusion is likely based on the research result on profiling. By contrast, Probert [4] mentions that edge probe implementation is more difficult and generates more execution overhead when advocating for vertex probes in well-delimited programs.

Overall, tracing with vertex probes has not received much of research interest for both minimization and optimization. More precisely, no further research is known to the author that would investigate additional steps to reduce vertex probes for tracing to bring down intrusion.

III. Tracing with Vertex Probes

The simplest solution to $trace(G, \mathcal{P}_v)$ is to place a probe at each vertex. Ball and Larus [5] proved that any edge probe placement $\mathcal{P}_e$ solving $ecount(G, \mathcal{P}_e)$ can be also successfully used for tracing, $trace(G, \mathcal{P}_e)$. Intuitively, the same might hold for vertex counter allocation. Since any solution to $vcount(G, \mathcal{P}_v)$ contains one of the minimal solutions [3], it is sufficient to show that any of the minimal solutions is a solution to $trace(G, \mathcal{P}_v)$.

The Knuth-Stevenson algorithm [3] finds a minimal solution to $vcount(G, \mathcal{P}_v)$ by finding a spanning tree of the reduced graph where edges, called v-edges, represent the flow through

basic blocks. Let G denote the original control flow graph and G_R the reduced graph. Both graphs are tuples $G = \langle V, E\rangle$, $G_R = \langle V_R, E_R\rangle$. It can be assumed without loss of generality that the CFG G contains the edge from $EXIT$ to $ENTRY$, i.e. $EXIT \rightarrow ENTRY \in E$. The vertices V_R, called supervertices, represent grouped original vertices. The edges E_R are the v-edges and there is a one-to-one correspondence between v-edges and all vertices from V captured by a bijective map $f_R : V \rightarrow E_R$.

Lemma 3.1: For each path $p = (v_1, v_1 \rightarrow v_2, \ldots v_{n-1} \rightarrow v_n, v_n)$ in G there is a corresponding path $p_R = (sv_1, f_R(v_1), sv_2, f_R(v_2), \ldots, f_R(v_{n-1}), sv_n)$ in G_R where sv_i are the supervertices and $v_i \in sv_i$.

Proof of Lemma 3.1:

Each v-edge starts at the supervertex containing the vertex whose flow is represented by the v-edge. The v-edge ends at the supervertex aggregating all successors of the v-edge vertex. As a result, the corresponding path exists. For a more formal proof see [20].

The consequence of Lemma 3.1 is that all paths in the original graph are preserved in the reduced graph.

Lemma 3.2: Let $\mathcal{P}_v$ be the minimal solution to $vcount(G, \mathcal{P}_v)$. For any two vertices $u, v \in \mathcal{P}_v$ there is at most one path starting at u and ending at v that doesn't cross any vertex in $\mathcal{P}_v$.

Proof of Lemma 3.2:

By contradiction.

Suppose that there are at least two different paths from a vertex u to vertex v, both $u, v \in \mathcal{P}_v$; each path crossing only vertices that are not in $\mathcal{P}_v$. Consider any two different paths from u to v. Since there is at most one directed edge between any two vertices in the control flow graph G in a single direction, these paths differ in at least one vertex they traverse. Lemma 3.1 implies that there are two corresponding paths in G_R of the form $(sv_u, f_R(u), \ldots, sv_v), u \in sv_u, v \in sv_v$. Since f_R is bijective, the two corresponding paths in G_R differ in at least one v-edge. However, both these paths traverse only v-edges, besides $f_R(u)$, in the reduced graph spanning tree; thus, the spanning tree contains an unoriented loop. By the definition of spanning tree this is a contradiction. Therefore, no such two paths can exist.

Theorem 3.1: Let G be a control flow graph and let $\mathcal{P}_v$ be a minimal solution of $vcount(G, \mathcal{P}_v)$. Then $\mathcal{P}_v$ is also a solution to $trace(G, \mathcal{P}_v)$; i.e $trace(G, \mathcal{P}_v)$ holds.

Proof of Theorem 3.1:

By definition of $vcount(G, \mathcal{P}_v)$, at least one vertex in $\mathcal{P}_v$ is visited during execution. The consequence of Lemma 3.2 is that two consecutive labels in a trace file identify uniquely the traversed path between the probes that had written the labels. Hence, it remains to show that there is at most a single path from $ENTRY$ to any vertex in $\mathcal{P}_v$ and there is at most a single path from any vertex in $\mathcal{P}_v$ to $EXIT$.

Obviously, it is the case if $ENTRY$ or $EXIT$ belongs to $\mathcal{P}_v$. If $ENTRY$ belongs to the G_R spanning tree, then there is still only one path from $ENTRY$ to all other supervertices because otherwise there would be an unoriented loop in the spanning tree, which is a contradiction. Similarly for $EXIT$, there is only a single path from all supervertices to $EXIT$.

It is possible to reconstruct the executed path from the content of the trace file recorded by probes allocated at $\mathcal{P}_v$. The general reconstruction procedure is captured in Corollary 3.1.

Each executed path starts at $ENTRY$. If $ENTRY$ is not in $\mathcal{P}_v$, correct execution is assumed, and when the first probe in $\mathcal{P}_v$ is encountered, the execution must have started at $ENTRY$. Similarly, correct execution means that each execution reaches $EXIT$. If not, then execution path can be reconstructed up to the last correctly recorded label.

In general, the minimal solution $\mathcal{P}_v$ of $vcount(G, \mathcal{P}_v)$ may not contain $ENTRY$, $EXIT$, or both. This depends on the spanning tree construction. As a result, $\hat{\mathcal{P}_v}$ is defined as a superset of $\mathcal{P}_v$ containing both $ENTRY$ and $EXIT$. So,

$$\hat{\mathcal{P}_v} = \mathcal{P}_v \cup \{ENTRY, EXIT\}.$$

Corollary 3.1: Let $\hat{\mathcal{P}_v}$ be as defined above and let $P = \{\emptyset\} \cup \mathbb{P}(\hat{\mathcal{P}_v})$ where $\emptyset$ is a null path (no path) and $\mathbb{P}(\hat{\mathcal{P}_v})$ is a set of all paths in the original CFG, each path starting and ending at a vertex in $\hat{\mathcal{P}_v}$ while crossing no vertices from $\hat{\mathcal{P}_v}$. There is a map $f_t : \hat{\mathcal{P}_v} \times \hat{\mathcal{P}_v} \rightarrow P$ that assigns to each pair (u, v), both $u, v \in \hat{\mathcal{P}_v}$, a path from P.

Corollary 3.1 captures the fact that for a correct trace generated by probing vertices in $\hat{\mathcal{P}_v}$, or simply in $\mathcal{P}_v$ if correct execution is assumed, the path between any two consecutive labels in the trace file can be recognized. In the case of an incorrect trace due to for example faulty execution, there can be a pair of labels with no path between them.

In conclusion, both minimal solutions to $vcount(G, \mathcal{P}_v)$ and $ecount(G, \mathcal{P}_e)$ yield probe placement that can be used for tracing. While $\mathcal{P}_e$, $ecount(G, \mathcal{P}_e)$, might often provide allocation with smaller overhead, in order to limit instrumentation intrusion, vertex probes shall be the preferred option as they do not modify control flow.

Furthermore, the solution yielding fewer probes is also less intrusive. Hence, it is beneficial to compare the number of probes of the minimal solutions of $vcount(G, \mathcal{P}_v)$ and $ecount(G, \mathcal{P}_e)$. Note that for the minimal probe placement $\mathcal{P}_e$, $ecount(G, \mathcal{P}_e)$, $|\mathcal{P}_e| = |E| - |V| + 1$ [2].

Theorem 3.2: Let $G = \langle V, E\rangle$ be a control flow graph and let $\mathcal{P}_v$ be a minimal solution of $vcount(G, \mathcal{P}_v)$. Then $|\mathcal{P}_v| \leq |E| - |V| + 1$.

Proof of Theorem 3.2:

There are $|V| - |S| + 1$ vertex counters where S is the set of supervertices. Supervertices are created by merging end vertices of each vertex outgoing

edges. Every vertex $v_i, v_i \in V$ has k_i outgoing edges. Hence, when reduced graph construction merges the end vertices of outgoing edges, at most $k_i - 1$ supervertices collapse into a single supervertex. Thus,

$$|S| \geq |V| - \sum_{|V|}(k_i - 1)$$
$$|S| \geq |V| - (\sum_{|V|} k_i - |V|)$$
$$|S| \geq |V| - (|E| - |V|)$$
$$|S| \geq 2 \cdot |V| - |E|$$

From this

$$|\mathcal{P}_v| = |V| - |S| + 1$$
$$|\mathcal{P}_v| \leq |V| - (2 \cdot |V| - |E|) + 1$$
$$|\mathcal{P}_v| \leq |E| - |V| + 1$$

Corollary 3.2: Given a control flow graph $G = \langle V, E\rangle$ and both minimal and optimal solutions $\mathcal{P}_e, \mathcal{P}_v$ to $ecount(G, \mathcal{P}_e)$ and $vcount(G, \mathcal{P}_v)$ respectively, it holds for those solutions that

- $|\mathcal{P}_e| = |E| - |V| + 1$
- $|\mathcal{P}_v| \leq |\mathcal{P}_e|$

As a result, the minimal solution to $vcount(G, \mathcal{P}_v)$ is the one with fewer probes and thus less intrusive.

IV. Tracing with Minimal Set of Probes

Given a control flow graph G and a vertex probe allocation $\mathcal{P}_v$ that satisfies $trace(G, \mathcal{P}_v)$, $\mathcal{P}_v$ can be reduced by gradually removing vertices from $\mathcal{P}_v$ while $trace(G, \mathcal{P}_v)$ still holds. Ultimately, $\mathcal{P}_v$ is modified by the removal procedure to the point when it cannot be reduced any further. The probe placement that cannot be reduced any further is called *Locally Minimal Solution*. In this section, such a solution is defined. In addition, algorithms for finding Locally Minimal Solution (LMS) are identified and properties of LMS described.

A. Formal Extension

The following definitions extend the established terminology described in Section II-A. A *complete* path in a control flow graph is a path that starts at $ENTRY$ and ends at $EXIT$; i.e. $(ENTRY, e_1, \ldots, EXIT)$.

A *vertex trace* of a path $p = (v_0, e_1, v_1, \ldots, e_n, v_n)$ is a sequence of vertex labels

$$\tau_v(p) = (label(v_0), label(v_1), \ldots, label(v_n)).$$

In a similar manner, an *edge trace* of path p is a sequence of edge labels

$$\tau_e(p) = (label(e_1), label(e_2), \ldots, label(e_n)).$$

A trace $\tau(p)$ is either a vertex trace or an edge trace; i.e. $\tau(p) = \tau_v(p)$ or $\tau(p) = \tau_e(p)$. Since a label uniquely identifies a vertex or an edge, both vertex trace and edge trace uniquely identify the path.

A *subtrace* of a path p is any subsequence of trace $\tau(p)$ containing the first and the last elements of $\tau(p)$. A subtrace can be derived by deleting some elements from a trace except the first and the last one. A trace is a subtrace of itself. A minimal subtrace is a pair of the first and the last element of its respective trace. If s is a subtrace of a path p, then it is written as $\sigma(s, p)$. If s is an edge subtrace of a path p, then it is written as $\sigma_e(s, p)$, and similarly $\sigma_v(s, p)$ denotes s to be a vertex subtrace of path p.

Given a directed graph G, a path p in G, and a subtrace s of p, s is a *characteristic subtrace* of p, $\bar{\sigma}(s, p, G)$, if for any path q in G, $p \neq q$, s is not a subtrace of q. A characteristic subtrace can be either a vertex characteristic subtrace or an edge characteristic subtrace, denoted $\bar{\sigma}_v(s, p, G)$ and $\bar{\sigma}_e(s, p, G)$, respectively. Clearly, $\bar{\sigma}_v(\tau_v(p), p, G)$ and $\bar{\sigma}_e(\tau_e(p), p, G)$ hold for any path p in G.

Assume an execution path p from Figure 1

$$p = (ENTRY, ENTRY\_A, A, A\_B, B, B\_D, D, D\_E, E, E\_F, F, F\_EXIT, EXIT).$$

Its vertex trace $\tau_v(p)$ is

$$\tau_v(p) = (ENTRY, A, B, D, E, F, EXIT).$$

Subtraces $(ENTRY, B, D, F, EXIT)$ and $(ENTRY, D, EXIT)$ are the characteristic subtraces of p; i.e.

$$\bar{\sigma}((ENTRY, B, D, F, EXIT), p, G) \text{ and}$$
$$\bar{\sigma}((ENTRY, D, EXIT), p, G).$$

B. Locally Minimal Solution for Tracing

Definition 4.1: Let $\mathcal{P}_v$, $ENTRY, EXIT \in \mathcal{P}_v$, be the set of vertices in a control flow graph $G = \langle V, E\rangle$. The vertex probe placement $\mathcal{P}_v$ is the solution of the tracing problem $trace(G, \mathcal{P}_v)$ if and only if for any path p in G that starts and ends at vertices from $\mathcal{P}_v$ the trace recorded by probes at $\mathcal{P}_v$ is a sequence λ such that $\bar{\sigma}_v(\lambda, p, G)$.

The definition applies only to paths that start and end at vertices in $\mathcal{P}_v$. Since $ENTRY \in \mathcal{P}_v$, $EXIT \in \mathcal{P}_v$, the definition covers all complete paths. Execution of any complete path results in a trace file that contains a characteristic subtrace of the path, which is, by the definition of characteristic subtrace, a sufficient and also mandatory condition to reconstruct the path. In the case of an incomplete path, the path can be reconstructed up to the last probe that wrote a label to the trace file.

There are other definitions in related works; in particular, they are due to Ramamoorthy et al. [1] and Larus and Ball [21]. It is shown in [20] that these definitions are equivalent to Definition 4.1, although they target edge probes.

Lemma 4.1: Let G be a control flow graph, $G = \langle V, E\rangle$. The set of probed vertices $\mathcal{P}_v$, $\mathcal{P}_v \subset V$, is a solution to the tracing problem $trace(G, \mathcal{P}_v)$ if and only if for any pair of vertices (u, v), $u, v \in \mathcal{P}_v$, there is at most one path from u to v in G crossing no vertices from $\mathcal{P}_v$.

Proof of Lemma 4.1:

Suppose $\mathcal{P}_v$ is a solution to $trace(G, \mathcal{P}_v)$. If there

is a path from u to v, $u, v \in \mathcal{P}_v$, that does not cross any vertices from $\mathcal{P}_v$, then by Definition 4.1 $\bar{\sigma}_v((label(u), label(v)), p, G), p = (u, \ldots, v)$. Hence, no other path $q, q \neq p, q = (u, \ldots, v)$, such that $\bar{\sigma}_v((label(u), label(v)), q, G)$ can exist.
Conversely, suppose that $\mathcal{P}_v$ is not a solution of $trace(G, \mathcal{P}_v)$. So, there is a sequence λ generated by recording labels of vertices in $\mathcal{P}_v$ along some path p starting and ending at vertices from $\mathcal{P}_v$ for which $\neg\bar{\sigma}_v(\lambda, p, G)$. Thus, there is another path q for which also $\neg\bar{\sigma}_v(\lambda, q, G)$ but $\sigma_v(\lambda, q)$. Since $p \neq q$, the paths cross different vertices not in $\mathcal{P}_v$ between some two vertices in $\mathcal{P}_v$. As a result, there is at least one pair of vertices $(u, v), u, v \in \mathcal{P}_v$, for which there are two different paths from u to v crossing no vertices in $\mathcal{P}_v$.

Lemma 4.1 states the pretty much obvious fact that there can be at most only one path between any pair of vertex probes. Otherwise, the executed path cannot be reconstructed if the two probe labels appear in a trace file one after the other.

Definition 4.2: Let $\mathcal{P}_v$ be a solution to the tracing problem $trace(G, \mathcal{P}_v)$ for some control flow graph G. A vertex v, $v \in \mathcal{P}_v$, is redundant *if it can be removed from the solution without invalidating the solution. In other words, $\mathcal{P}'_v = \mathcal{P}_v - \{v\}$ and $trace(G, \mathcal{P}'_v)$ still holds.*

If $\mathcal{P}_v$ contains a redundant vertex, this vertex can be removed in order to reduce the size of $\mathcal{P}_v$ further. However, $\mathcal{P}_v$ can contain more than one redundant vertex and removing one of them can and often actually does make formerly redundant vertices non-redundant.

Definition 4.3: If a solution $\mathcal{P}_v$ to the tracing problem $trace(G, \mathcal{P}_v)$ contains no redundant vertices, the solution is locally minimal.

Definition 4.4: Let $\check{\mathcal{P}}_v$ satisfy $trace(G, \check{\mathcal{P}}_v)$. If there is no other vertex set $\mathcal{P}_v$ that solves $trace(G, \mathcal{P}_v)$ and $|\mathcal{P}_v| < |\check{\mathcal{P}}_v|$, then $\check{\mathcal{P}}_v$ is a globally minimal *solution.*

Clearly, a globally minimal solution must be a locally minimal solution; i.e. a globally minimal solution contains no redundant vertex.

A locally minimal solution can be found by iteratively identifying and removing redundant vertices. Yet, no mechanism has been described for finding redundant vertices. In the next few paragraphs, the building blocks for finding redundant vertices are established.

Assume again a control flow graph $G = \langle V, E\rangle$ and any solution $\mathcal{P}_v$ to $trace(G, \mathcal{P}_v)$. By Lemma 4.1, for any pair of vertices (u, v), $u \in \mathcal{P}_v, v \in \mathcal{P}_v$ there is at most one directed path from u to v and also at most one directed path from v to u where both paths are not crossing any vertices from $\mathcal{P}_v$. Those paths can be represented by edges $u \rightarrow v, v \rightarrow u$, respectively. Moreover, one can construct a graph $G^v_T = \langle \mathcal{P}_v, E_T\rangle$ where $\mathcal{P}_v$ is the solution to $trace(G, \mathcal{P}_v)$ and edges E_T represent the paths in G between vertices in $\mathcal{P}_v$ not crossing any vertices in $\mathcal{P}_v$.

Definition 4.5: For a control flow graph $G = \langle V, E\rangle$, a tracing solution graph *$G^v_T = \langle \mathcal{P}_v, E_T\rangle$ is a directed graph where $\mathcal{P}_v$ is a solution to $trace(G, \mathcal{P}_v)$ and*

$$E_T = \{u \rightarrow v \mid u \in \mathcal{P}_v, v \in \mathcal{P}_v \text{ and there is a path in } G \text{ from } u \text{ to } v \text{ not crossing any vertex from } \mathcal{P}_v\}$$

A tracing solution graph, or shortly a trace graph, can be constructed by finding all edges for each vertex in $\mathcal{P}_v$. Hence, the construction of the trace graph can be achieved in $O(|V|^2 \cdot |E|)$.

Theorem 4.1: Let G be a control flow graph and let $\mathcal{P}_v$ be the solution of the tracing problem $trace(G, \mathcal{P}_v)$. Let G^v_T be a tracing solution graph, $G^v_T = \langle \mathcal{P}_v, E_T\rangle$. A vertex v, $v \in \mathcal{P}_v$, is redundant if and only if $\neg\exists e, e \in E_T, e = u \rightarrow w, u \in pred(v, G^v_T), w \in succ(v, G^v_T)$. In other words, there is no edge from one of v's predecessors to one of its successors in G^v_T.

Proof of Theorem 4.1:

Let G be a control flow graph and let $\mathcal{P}_v$ be the solution of $trace(G, \mathcal{P}_v)$; i.e. $\mathcal{P}_v$ satisfies Definition 4.1. Let v be a vertex from $\mathcal{P}_v$. Suppose that there are no edges from v's predecessors to v's successors in G^v_T. In other words, for any pair (u, w), $u \in pred(v, G^v_T), w \in succ(v, G^v_T)$, it is that $u \rightarrow v, v \rightarrow w \in E_T$ but $u \rightarrow w \notin E_T$. It is now to be shown that $\mathcal{P}_v - \{v\}$ is still a solution to the tracing problem.
Let $\mathcal{P}'_v = \mathcal{P}_v - \{v\}$. Let u, w be two vertices from $\mathcal{P}_v$ different from v. So, $u, w \in \mathcal{P}'_v$. By Lemma 4.1, there is at most one path from u to w crossing no vertices from $\mathcal{P}_v$.
First, assume that $u \notin pred(v, G^v_T) \vee w \notin succ(v, G^v_T)$. In this case, any path from u to w that crosses v must cross another vertex $z, z \in \mathcal{P}_v, z \neq v$. Therefore, even if v is removed from $\mathcal{P}_v$, there is still at most one path from u to w not crossing any other vertices in $\mathcal{P}'_v$ as any path from u to w going through v must cross vertex $z, z \in \mathcal{P}'_v$.
In the case of $u \in pred(v, G^v_T) \wedge w \in succ(v, G^v_T)$, there is by the condition in Theorem 4.1 no path from u to w crossing no vertices from $\mathcal{P}_v$. By Lemma 4.1, there is only one path from u to v crossing no vertices from $\mathcal{P}_v$ (and also in $\mathcal{P}'_v$). Similarly, there is only one path from v to w crossing no vertices from $\mathcal{P}_v$. Note that these paths are represented by edges $u \rightarrow v$ and $v \rightarrow w$ in G^v_T. As a result, there is only one path from u to w crossing no vertices from $\mathcal{P}'_v$ and this path is a result of concatenation of paths $(u, \ldots, v)$ and $(v, \ldots, w)$ that cross no vertices in $\mathcal{P}_v$. All other paths from u to w cross at least one vertex from $\mathcal{P}'_v$.
Hence, $trace(G, \mathcal{P}'_v)$ holds.
Conversely, if there is an edge from predecessor u of v to successor w in G^v_T, there is a path from u to w crossing no vertices from $\mathcal{P}_v$ including v. If v is removed, there would be two paths from u to

w crossing no vertices from $\mathcal{P}'_v$; one that doesn't cross v and the second one that goes through v. As a result, v cannot be removed from $\mathcal{P}_v$ and thus v is not redundant.

Theorem 4.1 is the climax of the series of constructions in this section. The theorem establishes a method for redundant vertex identification in any solution $\mathcal{P}_v$ to $trace(G, \mathcal{P}_v)$. This allows construction of an algorithm for both finding redundant vertices and finding a locally minimal solution.

Algorithm 4.1: Locally Minimal Solution

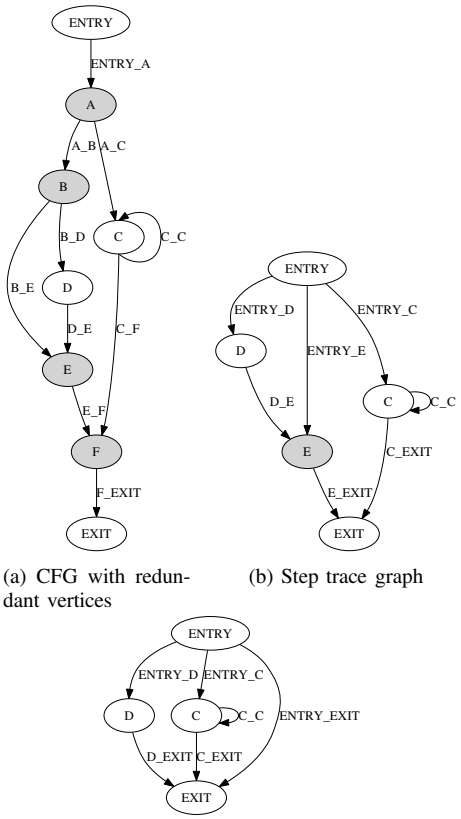
1) `Construct tracing solution graph (trace graph)`
2) `For each vertex in` $\mathcal{P}_v$
   a) `identify vertex predecessors and successors in the trace graph`
   b) `if there is no edge from predecessors to successors, mark the vertex as redundant`
3) `if there is no redundant vertex, return` $\mathcal{P}_v$ `as a locally minimal solution. END.`
4) `else remove one redundant vertex from` $\mathcal{P}_v$ *(possibly with the highest cost).*
5) `return to` *1*

The running time of the algorithm is $\mathrm{O}(|V|^3 \cdot |E|)$. It suffices to check for redundancy only those vertices that were redundant in the previous cycle because a non-redundant vertex cannot become redundant by removing a redundant vertex.

Let's demonstrate the process of finding a minimal solution on the example from Figure 1a. The algorithm starts with $\mathcal{P}_v = V$. In this case, the tracing solution graph is the control flow graph and the algorithm identifies 4 redundant vertices, Figure 2a. The algorithm removes vertex A and continues with the construction of the tracing solution graph again; this time, however, for $\mathcal{P}_v - \{A\}$. Next time, for example, F is removed, then B. This is because these two vertices are still redundant after constructing the trace graph. The tracing solution graph after removing B is in Figure 2b. The vertex E is still redundant and therefore it is removed. In the next step, there are no redundant vertices identified, Figure 2c.

Note that the CFG in Figure 2a is missing the $EXIT\_ENTRY$ edge. This is on purpose, so neither $ENTRY$ nor $EXIT$ is marked as redundant. Otherwise, $ENTRY$ can be removed and then the tracing solution graph would contain edges from $EXIT$ to vertices inside the graph, for instance to D. However, such paths do not really exist. As mentioned in Section II-B, the edge $ENTRY\_EXIT$ is to make the CFG satisfying Kirchhoff's law and represents the procedure invocation. The edge is not needed and is counterproductive for tracing solution graph construction and redundant vertex identification.

The edges in the final trace graph, Figure 2c, represent paths between probes. In order to reconstruct the execution path from a trace, the reconstruction process needs to maintain a map as defined in Corollary 3.1. For example,



(a) CFG with redundant vertices (b) Step trace graph

(c) Final trace graph

Fig. 2. Tracing solution graphs

$$\begin{aligned}
\text{(ENTRY,D)} &\rightarrow \text{(ENTRY,ENTRY_A,A,A_B,B,B_D,D)} \\
\text{(ENTRY,EXIT)} &\rightarrow \text{(ENTRY,ENTRY_A,A,A_B,B,B_E,E,} \\
&\quad \text{E_F,F,F_EXIT,EXIT)} \\
\text{(C,C)} &\rightarrow \text{(C,C_C,C)} \\
\text{(C,D)} &\rightarrow \emptyset
\end{aligned}$$

Hence, the execution path reconstruction algorithm can be like follows

```
1) start = read_label(trace_file)
2) while(start!='EOF') {
     end = read_label(trace_file)
     if(end!='EOF') print(lookup(start,end))
     start = end
   }
3) END.
```

Of course, this is just a sketch of the algorithm as it has no error checking and in the case of impossible path, it just writes it to output. Nevertheless, it demonstrates that the reconstruction is quite simple.

1) More Efficient LMS Algorithm: The LMS algorithm 4.1 finishes in $\mathrm{O}(|V|^3 \cdot |E|)$ time. The following algorithm runs in $\mathrm{O}(|V|^2 \cdot |E|)$ time.

Algorithm 4.2: Locally Minimal Solution

1) `Construct tracing solution graph (trace graph)`
2) `For each vertex in` $\mathcal{P}_v$

a) identify vertex predecessors and successors in the trace graph
b) if there is no edge from predecessors to successors, remove the vertex from $\mathcal{P}_v$ and from the trace graph

3) return $\mathcal{P}_v$ as a locally minimal solution.
END.

LMS algorithm 4.2 removes redundant vertices in the FIFO fashion; i.e. in the order it finds them. Once a redundant vertex is encountered, it is removed and the trace graph changed accordingly. The algorithm doesn't have to start again after the removal of the redundant vertex because non-redundant vertices cannot become redundant by a removal of a redundant vertex.

The drawback of this FIFO approach is that it cannot take into account the weights of the vertices. To defeat this drawback, the vertices in the initial tracing solution would need to be first ordered. If the selection of the to-be-removed redundant vertex was based on a more complicated analysis than just weight, it might be impossible to use Algorithm 4.2.

The process of removing redundant vertices ends when a locally minimal solution is found. However, the solution is not necessarily globally minimal. It is possible that if the algorithm removes vertices in a different order, a smaller locally minimal solution can be found. The problem of whether an algorithm for finding a globally minimal solution even exists is addressed in Section V.

V. Vertex Tracing Probe Placement is NP-Complete

LMS algorithm finds a locally minimal solution by gradually removing redundant vertices. The resulting locally minimal solution depends on the order in which the redundant vertices are being removed.

A globally minimal solution is a vertex probe placement that needs the least number of probes as defined in Definition 4.4. Hence, globally minimal solution is the smallest locally minimal solution. Yet, no algorithm has been described for determining the right order of redundant vertex removals to arrive at the globally minimal solution.

Theorem 5.1: The problem of finding globally minimal solution to tracing vertex probe placement, $trace(G, \mathcal{P}_v)$ for any control flow graph G is NP-complete.

Proof of Theorem 5.1:

It is simple to see that the problem is in NP. If it belonged to P, there would be a polynomial solution. On the other hand, there is a polynomial algorithm to check that $\mathcal{P}_v$ solves $trace(G, \mathcal{P}_v)$. Hence, for any yes-instance of the problem, there is a nondeterministically polynomial algorithm that eventually answers yes.

The vertex cover problem is a well known NP-complete problem, defined by Karp as Node Cover problem [22]. It's decision version is that given an undirected graph $G = \langle V, E\rangle$ and a natural number $k, k > 0$, there is a set $S, S \subseteq V$ such that every edge from E is incident to at least one vertex in S and $|S| \leq k$. Without loss of generality, let G be a simple graph, which is the usual practice. A simple graph is an undirected graph with no self-loop and at most one edge between any two vertices.
Suppose a (control flow) graph $CFG = \langle V_C, E_C\rangle$.

$$V_C = \{ENTRY, EXIT\} \cup V \cup E$$
$$E_C = \{ENTRY \rightarrow e, e \rightarrow e | e \in E\} \cup$$
$$\{e \rightarrow u, e \rightarrow v | e \in E \text{ and } u, v \text{ are the incident vertices of } e \text{ in } G\} \cup$$
$$\{v \rightarrow EXIT | v \in V\}$$
$$k_C = k + |E| + 2$$

Let $\mathcal{P}_v \subseteq V_C$, suppose $trace(CFG, \mathcal{P}_v)$ holds, and $|\mathcal{P}_v| \leq k_C$. Since $ENTRY$ and $EXIT$ have no predecessors and successors, respectively, they can never be redundant and therefore $ENTRY \in \mathcal{P}_v$ and $EXIT \in \mathcal{P}_v$. Since every vertex $e, e \in E$, is its own predecessor and successor, it can be never redundant. Hence, $\forall e, e \in E, e \in \mathcal{P}_v$. Every vertex $e \in E$ has two outgoing edges in the CFG with end vertices that are the vertices incident to the edge e in G. For each e, at least one of those vertices must belong to $\mathcal{P}_v$; otherwise, there would be two edges from e to $EXIT$ in the trace graph defined by $\mathcal{P}_v$ and thus $\mathcal{P}_v$ would not be a solution to $trace(CFG, \mathcal{P}_v)$. Hence, $S = \mathcal{P}_v - \{ENTRY, EXIT\} - E$ and $|S| \leq k$.
Conversely, if S is a vertex cover, then $\mathcal{P}_v = S \cup \{ENTRY, EXIT\} \cup E$ solves $trace(CFG, \mathcal{P}_v)$ because incoming edges of removed vertices, $\bar{S} = V - S$, start at vertices E, which have two outgoing edges where at least one ending at vertex in S. Thus there is at most one path from each vertex in E to $EXIT$ not crossing a vertex from $\mathcal{P}_v$.

The outcome of Theorem 5.1 is that it is unlikely that a polynomial algorithm for finding globally minimal solution can be conceived. Therefore, a globally minimal solution can only be selected out of all locally minimal solutions.

VI. Optimality at Reach

LMS algorithm, Algorithm 4.1 or 4.2, can be started from any probe placement satisfying $trace(G, \mathcal{P}_v)$ including an optimal solution to $vcount(G, \mathcal{P}_v)$. Since the optimal solution to $vcount(G, \mathcal{P}_v)$ eliminates vertices with high costs (weights), the benefit of starting LMS algorithm from this probe placement is obvious. In order to evaluate the benefit of starting LMS algorithm from profiling optimal solution, the relationship of profiling minimal solution found by the Knuth-Stevenson algorithm [3] and locally minimal solution is analyzed.

Theorem 6.1: Let G be a control flow graph and let the vertex probe placement P_{LM} be any locally minimal solution to $trace(G, \mathcal{P}_v)$. Let G_R be the reduced graph of the control flow graph G as defined in Knuth-Stevenson algorithm. The graph $\langle V(G_R), E(G_R) - P_{LM} \rangle$ is then a spanning connected component of graph G_R.

Proof of Theorem 6.1:

By contradiction. See [20].

Since the spanning connected component contains the spanning tree of G_R, the following corollary can be concluded as a consequence of Theorem 6.1.

Corollary 6.1: For every locally minimal solution P_{LM} of $trace(G, P_{LM})$ there is a minimal solution P_C of $vcount(G, P_C)$ such that $P_{LM} \subseteq P_C$.

Proof of Corollary 6.1:

Since each P_{LM} complement forms a spanning connected component in G_R, we can add those vertices from the complement to P_{LM}, so the spanning connected component becomes a spanning tree. Then P_{LM} and the added vertices are a minimal counter-placement solution P_C.

The direct consequence of Corollary 6.1 is that all tracing locally minimal solutions can be found by LMS algorithm starting from minimal counter-placement solutions. Hence, it is likely that the cost of locally minimal solution found from optimal vertex counter placement is in the vicinity of the cost of the actual optimal tracing solution.

VII. Experiments on Performance

The algorithms, LMS algorithm and Knuth-Stevenson algorithm, were implemented in Perl. First, Knuth-Stevenson algorithm was applied to find a minimal solution of $vcount(G, \mathcal{P}_v)$. Next, LMS algorithm was run starting from the vertex probe placement identified by Knuth-Stevenson algorithm.

In order to compare the performance of both algorithms, a control flow graph generating method was developed. The generated graphs are of different size and also of different kind.

The first group of control flow graphs are reducible (well formed) graphs. The graphs are generated from a grammar with production rules replacing nonterminals with well-formed control flow structures: `if-then`, `if-then-else`, `while-loop`, `until-loop`, `sequence`. All those structures contained new nonterminals. Eventually, all nonterminals are simply replaced with a terminal and the generated graph is checked for simple sequences that can be collapsed.

The size of the reducible graphs ranged from graphs of around 10 vertices up to graphs with approximately 100 vertices. In total, 280 reducible graphs were generated. The generation was based on pseudo-random number generator; the randomness is applied to production rule selection. The probability that two same graphs were generated by the generation algorithm is lower than 0.95.

The second group of graphs are irreducible (ill-formed) graphs. The same number of graphs was generated with approximately the same number of vertices. The irreducible graphs are generated from almost the same grammar except that the control-flow structure `until-loop` was replaced with an irreducible loop. Also, the generation assures that the irreducible structure replaces a nonterminal at least once.

TABLE I
Comparison of probe set sizes

	EProf	**KS**	**LMS**
Reducible	50.1%	48.3%	43.2%
Irreducible	54.5%	54.5%	47.9%

Table I contains the relative sizes of reduced probe sets with respect to set of probes placed to every vertex.

EProf	set of edge probes for profiling [2].
KS	set of vertex probes for profiling by Knuth-Stevenson alg. [3]
LMS	set of vertex probes for tracing by LMS alg.

TABLE II
LMS Improvement

	≤ 25	> 25
Reducible	49.6%	88.7%
Irreducible	63.3%	94.7%

Table II shows the percentage of randomly generated graphs for which LMS algorithm found smaller set of probes that Knuth-Stevenson algorithm.The percentages are shown separately for small graphs having at most 25 vertices and larger graphs.

Table I illustrates the outcome of applying probe reduction algorithms to the generated graphs. The table shows in percentage how many probes are needed when different probe reduction algorithms are used. The percentage is calculated with respect to the probe placement to all vertices.

As one can see in the table, LMS algorithm provides improvement compared to already known methods. Yet, LMS algorithm has greater computational complexity. Therefore, whether LMS algorithm shall be applied needs to be judged upon the particular use case. When the reduction of probes is important, for example many traces are being collected or intrusion is critical, then LMS algorithm is useful. However, when one only needs to collect a few traces, profiling vertex probe placement may suffice.

Furthermore, it is also interesting to see how often LMS algorithm actually improves the probe placement found by Knuth-Stevenson algorithm. As described above, LMS algorithm was started from a minimal profiling vertex probe placement to collect such information. Table II shows how often in percentage LMS algorithm improved Knuth-Stevenson solution. As can be seen, LMS algorithm yields improvement only in half of the small graphs. Also, the improvement for small graphs is usually small, typically one vertex probe. However, for larger graphs the improvement starts to be more beneficial.

VIII. Conclusion

This paper revisited the problem of probe reduction for tracing. The major objective of the presented work was to

reduce the number of probes to mitigate intrusion. As a result, the work focused on less intrusive probes, the vertex probes.

It was shown that tracing probe reduction can be achieved by reusing minimal profiling probe placements and that vertex placement yields at most the same or smaller number of probes than edge placement. More importantly, this paper introduced a new algorithm, LMS algorithm, that reduces the number of probes beyond the profiling solution. LMS algorithm is based on finding a locally minimal solution for vertex probe placement, a solution that cannot be further reduced. Moreover, this paper presents the proof that finding globally minimal tracing probe placement is NP-complete.

If vertex cost as assumed execution frequency is considered, the paper suggests to start LMS algorithm from optimal profiling vertex placement. Such an approach shall lead to close-to-optimal probe placement because all locally minimal solutions can be found from minimal profiling probe placements.

The theoretical results were confirmed by an experiment. Vertex probe placement always provided the same or smaller number of probes. First, vertex profiling placement is superior to edge placement when the number of probes is the main criterion. Furthermore, LMS algorithm reduces the number of probes beyond currently known reduction methods. While for smaller graphs LMS algorithm often confirms that the minimal profiling vertex placement is also a locally minimal solution, for bigger graphs the algorithm provides further reduction.

LMS algorithm and its deployment to runtime data collection is protected by filed patent application [23].

IX. Future Work

The research results described in [20] identify additional benefits of Locally Minimal Solution to profiling and code coverage because tracing in general collects data that yield both profiling and code coverage results. Those result should be published. Also, it would be beneficial to verify the experimental results with respect to control flow graphs representing real code.

Furthermore, there are questions that still need to be addressed. First, does the locally minimal solution applied to vertices also extend to edges? Second, how does vertex locally minimal solution compare to known efficient edge probe placements in terms of optimality? Furthermore, as the LMS algorithm still has a high computational complexity, can it be decreased beyond as described in this paper? Finally, can Locally Minimal Solution be also used for path profiling [14] and what would be the efficiency of such a solution?

Acknowledgment

The author would like to extend his gratitude to Dr. David Guaspari for introduction to the world of formal methods and software verification. Further, the author's appreciations go to both his PhD. supervisors Dr. Radek Marik and Associate Professor Dr. Jiri Lazansky. Dr. Radek Marik's assistance in the initial stage of the research effort was very important in order to steer the research aim into right directions. Finally, the author would like to extend his thanks to his employer, Freescale Semiconductor, for sponsoring the initial stages of this research.

References

[1] C. V. Ramamoorthy, K. H. Kim, and W. T. Chen, "Optimal placement of software monitors aiding systematic testing," *IEEE Trans. Software Eng.*, vol. 1, no. 4, pp. 403–411, 1975.

[2] D. E. Knuth, *Fundamental Algorithms*, ser. The Art of Computer Programming. Addison-Wesley, 1969, vol. 1, ch. 2.3.4.1.

[3] D. E. Knuth and F. R. Stevenson, "Optimal measurement points for program frequency counts," *BIT*, vol. 13, no. 3, pp. 313–322, 1973.

[4] R. L. Probert, "Optimal insertion of software probes in well-delimited programs." *IEEE Trans. Software Eng.*, vol. 8, no. 1, pp. 34–42, 1982.

[5] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," in *In Proceedings of the Conference on Principles of Programming Languages*, 1992, pp. 59–70.

[6] ——, "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 4, pp. 1319–1360, July 1994. [Online]. Available: citeseer.ist.psu.edu/article/ball94optimally.html

[7] T. Ball, P. Mataga, and M. Sagiv, "Edge profiling versus path profiling: the showdown," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '98. New York, NY, USA: ACM, 1998, pp. 134–148. [Online]. Available: http://doi.acm.org/10.1145/268946.268958

[8] H. Agrawal, "Dominators, super blocks, and program coverage," in *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, 1994, pp. 25–34. [Online]. Available: citeseer.ist.psu.edu/agrawal94dominators.html

[9] M. Tikir and J. Hollingsworth, "Efficient instrumentation for code coverage testing," 2002. [Online]. Available: citeseer.ist.psu.edu/tikir02efficient.html

[10] K. Vaswani, "Preferential path profiling: Compactly numbering interesting paths," in *In ACM Symposium on Principles of Programming Languages*, 2007, pp. 351–362.

[11] R. Joshi, M. D. B. , and C. Zilles, "Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems," in *In International Symposium on Code Generation and Optimization (CGO*, 2004, pp. 239–250.

[12] M. D. Bond and K. S. Mckinley, "Practical path profiling for dynamic optimizers," in *In International Symposium on Code Generation and Optimization (CGO)*, 2005, pp. 205–216.

[13] S. Tallam, X. Zhang, and R. Gupta, "Extending path profiling across loop backedges and procedure boundaries," in *In International Symposium on Code Generation and Optimization (CGO)*, 2004, pp. 251–264.

[14] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 29. Washington, DC, USA: IEEE Computer Society, 1996, pp. 46–57. [Online]. Available: http://portal.acm.org/citation.cfm?id=243846.243857

[15] S. MAHESHWARI, "Traversal marker placement problems are np-complete," Dept. of Computer Science, Univ of Colorado, Boulder, Colo, Tech. Rep. CU-CS-092-76, 1976.

[16] J. R. Larus, "Efficient program tracing," *Computer*, vol. 26, pp. 52–61, 1993.

[17] T. Ball and J. R. Larus, "Programs follow paths," MICROSOFT RESEARCH, MICROSOFT RESEARCH, Tech. Rep., 1999.

[18] R. C. Prim, "Shortest connection networks and some generalizations." *Bell System Tech. J.*, vol. 36, pp. 1389–1401, 1957.

[19] O. Borůvka, "O jistém problému minimálním (about a certain minimal problem)." *Práca Moravské Přírodovědecké Společnosti*, no. 3, pp. 37–58, 1926.

[20] D. Baca, "Efficient collection of statistics for embedded software verification," Ph.D. dissertation, Czech Tecnical University in Prague, 2012.

[21] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," University of Wisconsin, Madison, Tech. Rep. 1031, 1991.

[22] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds. Plenum Press, 1972, pp. 85–103.

[23] D. Baca, "Software Probe Minimization," Patent application, September 2009, pCT/IB2009/054270.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

A State Alteration and Inspection-based Interactive Debugger

Yan Wang
CSE Department, UC Riverside
wangy@cs.ucr.edu

Min Feng
NEC Laboratories America
mfeng@nec-labs.com

Rajiv Gupta Iulian Neamtiu
CSE Department, UC Riverside
{gupta,neamtiu}@cs.ucr.edu

***Abstract*—Despite significant advances in automated debugging, programmers still rely on traditional interactive debuggers (e.g., GDB) to find and fix bugs. While popular, these debuggers have major deficiencies: they do not guide the programmer in narrowing the source of error, and they only support a limited and low-level set of state-altering commands, hence semantic state alteration requires recompilation and reexecution. To address these shortcomings, we present an interactive debugger that combines capabilities that reduce debugging effort and increase debugging speed. The capabilities that yield these benefits include: *state alteration* commands for dynamically switching the directions of conditional branches and suppressing the execution of statements; *state inspection* commands including navigating and pruning dynamic slices; and *state rollback and checkpointing* commands to allow reexecution of the program from an earlier checkpoint. Our prototype is built on top of GDB using the Pin infrastructure; we also provide a GUI based on KDbg. Our experience shows that our debugger handles many kinds of real bugs effectively and efficiently.**

***Index Terms*—debugging, dynamic slicing, state alteration**

I. Introduction

Debugging is a long and laborious process which takes up to 70% of the total time of software development and maintenance [15]. To assist in the debugging process, programmers make use of an interactive debugger (e.g., GDB) whose use involves: state inspection, state alteration, and code modification. The programmer executes the program on a failing input and uses *state inspection* commands to examine program state at various points (e.g., by setting breakpoints). After finding suspicious values, the programmer may apply *state alteration* to correct these values and see how the program's execution is affected. Alternatively, the programmer may perform *code modification* and then recompile and rerun the program to see how the program behavior is affected (e.g., commenting out suspicious statements [12]). To speed up this process, Visual Studio offers an Edit-and-Continue feature [26] for on-the-fly changes to the program being debugged, but code modifications such as most changes to global or stack data are not supported.

Both state alteration and code modification techniques help the programmer in understanding and locating faulty code. While state alteration is a lightweight technique, code modification slows down debugging as it requires program recompilation and reexecution. This can take a significant amount of time if the program runs for long before exhibiting faulty behavior and the above process is performed repeatedly. A debugger such as GDB supports simple state alteration commands for altering values of variables. Thus the programmers often resort to code modification to locate the bug.

978-1-4673-5739-5/13/$31.00

In this paper we present an interactive debugger which supports commands that reduce debugging effort and increase debugging speed. These commands allow the programmer to narrow his/her focus successively to smaller and smaller regions of code. The state alteration commands allow the programmer to narrow faulty code down to a function. The state inspection techniques allow efficient examination of large code regions by navigating and pruning dynamic slices and zooming in on chains of dependences. Finally, the programmer can zoom to a small set of statements in a slice by breakpointing at those statements and examining program state.

The commands in our debugger speed up the iterative debugging process by reducing the need for code modifications, which require recompilation and reexecution. Its state alteration commands allow the programmer to perform control flow alterations by switching outcomes of conditional branches. Its execution suppression commands allow skipping of statements during execution. That is, these commands effectively simulate the effect of code modifications without the need for recompilation. In addition, since our debugger also supports checkpoint and rollback, the programmer can rollback to an earlier execution point, specify state alterations, and then reexecute the program. A reexecution from a checkpoint instead of from the beginning can greatly reduce the waiting time associated with recompilation and reexecution for long executions.

In summary, our debugger supports new features that greatly improve bugfinding/bugfixing efficiency and speed up the iterative debugging process. Since it is built on top of GDB, all commands that are supported by GDB are still available. Some GDB commands have been enhanced and/or reimplemented for improved efficiency. Setting of breakpoints can be guided by program slices allowing the programmer to single-step from one statement in the slice to the next. Conditional breakpoints allow calls of library functions to be selectively captured. The overhead of the state rollback mechanism is reduced via incremental checkpointing. We have integrated the debugging back-end into the KDbg GUI, hence the new features are augmented by the GUI's visualization and navigation support. We have evaluated our debugger using five kinds of hard-to-find

TABLE I
MAJOR DEBUGGING COMMANDS

Summary	Commands	Description
State Alteration	`switch`	switch the outcome of a predicate
	`suppress`	suppress the execution of a statement
State Inspection	`record`	turn on/off recording for slicing
	`slice`	perform backward dynamic slicing
	`prune`	prune dynamic slices
	`sbreak`	create breakpoints in a slice
	`conditional breakpoint`	conditionally capture memory bug related library calls (e.g., *malloc, free*)
	`instance`	print execution instance of a statement
State Rollback	`checkpoint`	set up an incremental checkpoint
	`rollback`	rollback the program state

real bugs: double free, stack smashing, heap buffer overflow, dangling pointer dereference, and null pointer dereference. Our experience shows that our state alteration and inspection capabilities are effective and efficient.

II. DEBUGGING COMMANDS

We provide three kinds of debugging commands—**state alteration**, **state inspection**, and **state rollback**—listed in Table I. Of these, six commands—`switch`, `suppress`, `slice`, `record`, `prune`, and `instance` are not found in commonly-used interactive debuggers. Other commands are extended to support new debugging features. State alteration commands are used to isolate bugs and help programmers efficiently gain comprehension of program's faulty behavior. State inspection commands help programmers focus on bug-related statements and present unexpected dependences to the programmers and allow them to navigate along dependence edges. State rollback commands enable the quick reexecution of the suspicious code region. Programmers use these commands via the GDB command line or the GUI.

Our debugger can greatly relieve the burden of programmers. First, when a program crashes, it can be difficult for the programmer to reason about the execution flow, e.g., if the crash happens because a library call destroys an auto-maintained stack or heap. Our debugger captures the abnormal data dependences and presents them to the programmers in an intuitive way. Second, it is the programmer's responsibility to find suspicious code and speculate about the root cause. Our debugger enables the programmer to focus on bug-related statements and guides the examination of values and setting of breakpoints guided by dynamic slicing. Third, even after discovering all the bug-related statements, the programmer still has to understand and fix the bug. Our debugger enables the programmer to quickly identify critical bug manifestation condition (e.g., a critical function invocation) by leveraging state alteration and narrowing the fault to a small code region. Fourth, it is common that a variable use is data-dependent on a far-away definition in a different function or a file. Navigating across source files is burdensome. Our debugger enables the programmer to visually navigate captured dependences and reason about the execution.

A. State Alteration Interfaces

State alteration commands provide an easy way to alter the program execution state dynamically and enable programmers to avoid repetitive program compilations and executions.

1) Switching Control Flow: The command `switch file:line [all| once|n]` is designed to switch the outcome of the predicate in the *line*th line in file *file*. Programmers can choose to switch the outcome for every (*all*), next (*once*) or only the nth (*n*) instance of the predicate.

Using `switch`, programmers can dynamically change the outcome of a branch and then check the difference in program state and result. When a program crashes or deviates from the desired behavior, programmers use `switch` to invert the outcome of the predicate dynamically. If the program behaves correctly after inversion, the programmer can infer that there is an error in the predicate or the predicate is critical to bug manifestation. Otherwise the predicate is likely unrelated to the error. If there are several predicates in the execution trace of a failing run, all the predicates with which the program works properly by following the inverted branch compose the critical bug manifestation condition. That is, the bug disappears when the outcome of any of these predicates changes, providing valuable clues to the programmer to understand the bug. With the aid of `switch`, programmers avoid source code modification and recompilation which are time-consuming.

2) Execution Suppression: The `suppress file:line [all|once|n]` command suppresses the execution of statement at line *line* in file *file*. Programmers can choose to suppress every instance (*all*), only the next instance (*once*), or only the nth instance of the statement.

Like `switch`, `suppress` is useful for isolating a bug. A commonly-used debugging strategy is to temporarily comment out a section of code and then check whether the remaining part works as expected [12]. This approach involves recompilation of the source code. The `suppress` command is designed to simplify this procedure. If the programmer suspects that a statement or function is faulty, he can suppress its execution on-the-fly without having to modify the source code and recompile the program. For example, assume that the programmer has forgotten to use a guarding predicate around some statements, causing the program to crash. The programmer can try to suppress the unguarded statements based on his knowledge of the program. If desired results is observed, the programmer can then focus on fixing the code.

Programmers can first suppress a function call to identify a faulty function and then suppress statements in the function to identify faulty code. By reducing the suppression to finer granularities, the root cause of failure can be narrowed to a smaller code section.

B. State Inspection Interfaces

1) Dynamic Slicing: Dynamic slicing commands include the following.

- `record file:line on|off` identifies the code region where dynamic slicing is required.
- `slice stmt i variable|addr [size] | register` constructs a backwards dynamic slice for *variable*, memory region *[addr, addr+size)* or *register*, starting from the i^{th} execution instance of *stmt*. If no variable is specified, we generate a slice for all

the registers and variables used in current execution instance of *stmt*. Our debugger assigns unique numbers to each generated slice and feeds this number back to the programmer.

- `prune id list` is used to prune the id^{th} slice by eliminating from the slice all the dependence edges related to any variable or register in *list*.
- `sbreak id` s_1`[,` s_2`, ...]` is used to insert a breakpoint at $s_1{}^{th}$ (and $s_2{}^{th}$,...) statements in the id^{th} slice. The command `sbreak all id` inserts a breakpoint at each statement in the id^{th} slice. Breakpoints for `sbreak` are triggered when specific execution instances are encountered.
- `sdelete id` is used to delete the id^{th} slice.
- `info slices` is used to print a detailed report of all generated slices that have not been deleted.
- `instance file:line` prints the execution instance of $line^{th}$ line in file *file*.

With traditional debuggers, programmers navigate and conjecture the root cause over the whole execution trace [7]. With our debugger, programmers can infer the root cause in the pruned slices of variables with wrong values. These slices are *much smaller*. The `slice` command is very efficient in locating the root causes of bugs. It is common for a failing program to exhibit abnormal control or data dependences that can be quickly identified by examining a `slice` that captures them. For example, for a null pointer dereference bug, we can locate where the null pointer originates by examining the backwards slice of the null pointer. The slice may also help determine if the pointer was mistakenly set to null.

The `slice` command is also very useful for double free, heap and stack buffer overflow bugs. These memory-related bugs are notoriously hard to find because the source code of library functions is not available and the internal data structures (heap and stack metadata) are transparent to programmers. Our debugger traces into library calls and captures hidden dependences among internal data structures. For example, when a heap buffer overflow bug destroys an internal data structure maintained by the heap allocator, a dependence path from the place where the error is manifested to the overflow point is found. Since library source code is unavailable to programmers, we do not present dependences inside a library code. Instead, we squash the dependence edges to statements inside the library functions to their call sites. For example, consider a stack smashing bug inside a library call that causes a crash when the returning statement is executed. We report a dependence edge from the returning statement to the call site of library function.

During debugging, programmers often have high confidence that the program performs correctly for some execution segments. In that case, they can focus on the most suspicious region first. The `record` command allows recording the concerned code region and slicing based on the partial def-use information. Further, programmers may have high confidence on the correctness of some values. For example, they may know that a loop variable *i* has nothing to do with the failure. The dynamic slice can be pruned to exclude the dependences due to such values. The `prune` command removes dependence edges corresponding to variables or special registers in *list*. Thus, the `record` and `prune` commands greatly limit slice sizes and save programmers' time and effort.

The `sbreak` command facilitates setting breakpoints efficiently. It generates a breakpoint which is only triggered when the specific execution instance in the slice is reached. Programmers frequently step through the program execution to reason about the control/data flow and find faulty code. With the help of `sbreak all`, the programmer is able to only step through the statements and execution instances in the slice. Because all the statements influencing the value of a variable are included in the slice, stepping only through the statements in the slice reduces programmers' effort.

2) Conditional Breakpoints: Existing debuggers (e.g., GDB) provide conditional breakpoints; however, the condition must be defined at source code level that is not available to programmers for library functions. Thus, conditional breakpoints must be set at each call site, that is time consuming and inflexible. Therefore we provide a command `breakpoint lib_func [if condition]` that triggers a breakpoint at the call site of *lib_func* when *condition* is satisfied. The condition allows selective and efficient capture of critical library function invocations. The condition has the forms:

- `if argN|ret==value` triggers a breakpoint when the N^{th} argument or return value equals the given value.
- `if write/read/access addr [size]` triggers a breakpoint when the function writes/reads/accesses specified memory location.

Extended conditional breakpoints are very useful for memory-related bugs. For example, there may be three possibilities if a program crashes at a *free*— double free, unmatched free (i.e., freeing an unallocated pointer), or heap buffer overflow. The programmer can check for a double-free bug using *breakpoint free if arg1==fail_addr* to see if this memory region has been freed before. If a previous free with the same address is caught, this indicates a double-free bug. Otherwise, if no previous deallocation is found, programmers can use *breakpoint malloc if ret==fail_addr* to see if crash is due to deallocation of unallocated memory. Programmers can use *breakpoint strcpy/memcpy if arg1==addr* or *breakpoint strcpy/memcpy if write addr* to find if the specified memory location is modified in *strcpy/memcpy*, to find buffer overflow bugs.

C. State Rollback Interfaces

The `checkpoint` command creates a checkpoint and the debugger assigns an `id` to it. The command `rollback id` is used to go back to a previous checkpoint and re-execute from that point. The command `info checkpoints` prints the list of checkpoints and `cdelete id` deletes a checkpoint.

Traditional checkpointing [34] records/restores memory/register states and is inadequate for us. First, if the programmer rolls back the execution when recording is turned on, the recorded def-use information will wrongly include the rolled-back portion of the execution, thus slices generated

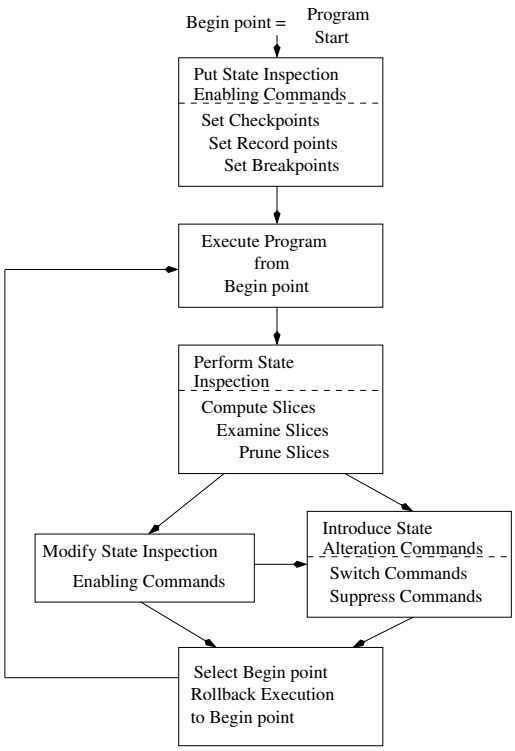


Fig. 1. Typical use of our debugger.

based on this information will be incorrect. Second, rolling back the program state to a previous checkpoint will cause inconsistency between the statement execution instances in the previously-generated slice and in the restored program. Our debugger extends the traditional incremental checkpointing mechanism to support state alteration and state inspection. In addition to recording (restoring) the memory and register states, we also record (restore) the execution instances. This extension maintains the consistency between generated slices and program state.

The `checkpoint` and `rollback` commands are particularly useful for iterative debugging. Without state rollback, programmers have to restart the execution every time they go over the possible faulty area or when they want to modify the program execution (e.g., altering input, switching a predicate, or suppressing a function call). Moreover, on systems such as Linux, the addresses of stack- and dynamically-allocated regions vary from run to run due to address space randomization for security. Therefore it is troublesome to diagnose bugs related to dynamically allocated regions (e.g., double free and heap buffer overflow) and stack (e.g., stack smash). Thanks to the `checkpoint` command, programmers can go back to a previous point and rerun the program from there, while keeping all addresses of dynamically allocated regions unchanged. The `rollback` command keeps the addresses the same when programmers rerun the program from a checkpoint.

III. Usage of Debugging Commands

In this section we first describe how different types of commands are used during the debugging process and then demonstrate their use in context of a set of hard-to-locate bugs from real programs.

Figure 1 overviews the debugging process based upon the supported commands. Let us assume that the execution of the program has failed on an input. First, the programmer enters commands that will later allow detailed state inspection and program reexecution. Then the program is executed from the beginning until execution stops due to an error or a breakpoint is encountered. The user can now perform state inspection, starting with computing a backward dynamic slice. The programmer can prune the slice based on the knowledge of the program; next, internal execution states can be probed by setting breakpoints at statements in the slice. Based upon the insights gained, the programmer may choose to use state inspection commands and/or apply state alteration techniques to further understand program behavior. By rolling back the execution to an earlier checkpoint, and reexecuting the program from that point, the programmer can observe the impact of state alteration by examining program state. This is an iterative process which eventually leads to location of faulty code. This iterative process does not require program recompilation or reexecution from the beginning.

Next we illustrate the process of finding a bug using our debugger. Figure 2 shows a stack buffer overflow bug (also referred to as stack smashing) in version 4.2.4 of the *ncompress* program. Line numbers are shown on the left. The bug is triggered when the length of the input filename (pointed to by *fileptr* at line 880) exceeds the size of array *tempname* defined at line 884 which stores the file name temporarily. The program crashes when the *comprexx* function tries to return to its caller because the return address of *comprexx* is overwritten at line 886 in the *strcpy* function.

Without our debugger it is extremely difficult for programmers to figure out why the program crashes when it executes the *return* statement (at line 946). First, because the program counter is corrupted, existing debuggers (e.g., GDB) cannot report the exact crash point. Our debugger reports the exact location by tracking the modification to the program counter and reporting its current and previous values. Second, the program crashes because a library call destroys the auto-maintained stack, neither of which are visible to the programmer; hence it is difficult to reason about the bug from the source code. Our debugger captures the hidden data dependence and presents it to the programmer.

Returning to our example, with our debugger the programmer knows that the program crashed at line 1252 (shown in Figure 3) and the program counter is modified at this crash point. Next, the program can be restarted and additional checkpoints introduced for later use. The programmer can also enable tracing at the beginning of *main* and turn it off at the crash point to later get the whole slice.

With our enhanced dynamic slicing, if the programmer omits the slice criterion, the debugger computes dynamic slices for all registers and variables used in a statement. This is very useful for memory-related bugs because there is no need for the programmer to figure out which variables or memory regions are used at the crash point. If we use the statement

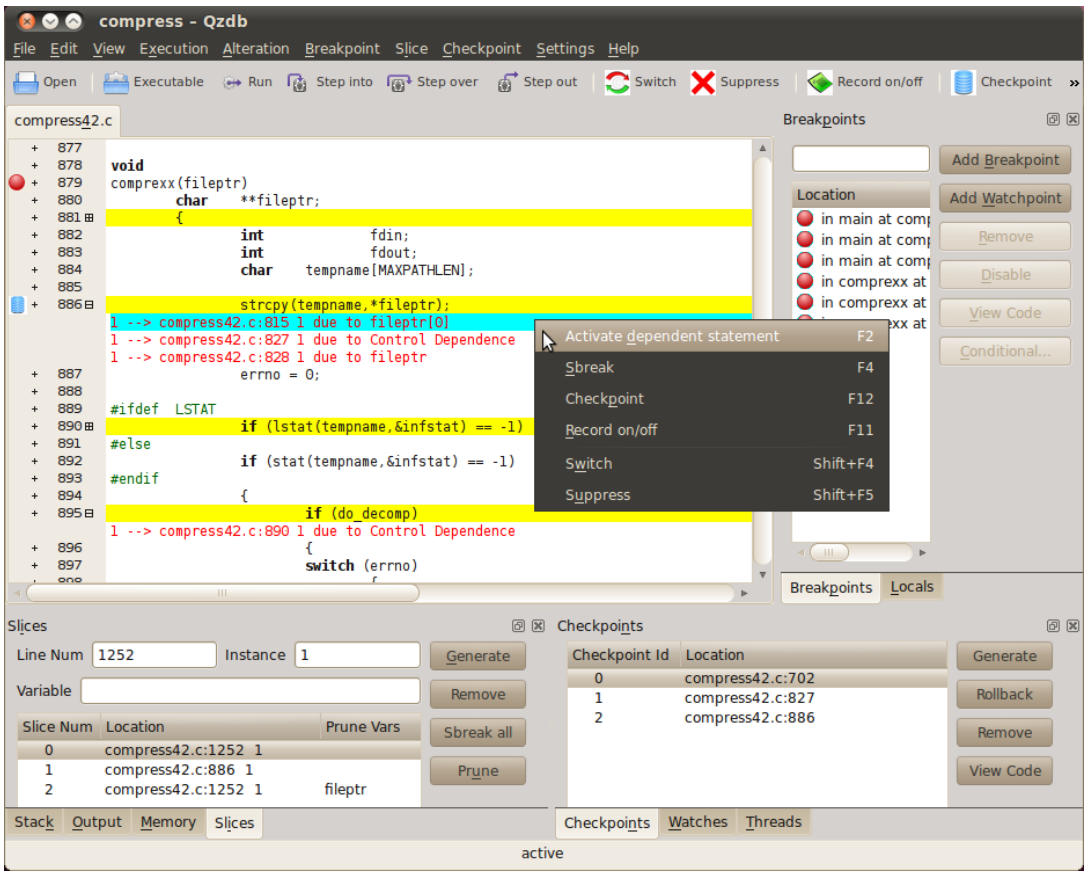


Fig. 2. The main window of our debugger.

line number (1252) and instance (1) as the slice criterion, the generated slice is as shown in Figures 2 and 3. All statements in the slice are highlighted in yellow (e.g., lines 1252 and 886) so programmers can focus on them.

To further help reason about the execution flow, our debugger captures and presents the concrete control/data dependence relationships. Our debugger also allows programmers to navigate the dependence edges and quickly identify unexpected control/data flow. To get the dependence relationships, users click on the left expansion mark of a statement in the slice. The dependence edges from statement $stmt_1$ are shown as follows: $instance_1 \rightarrow file_2 : line_2\ instance_2\ due\ to\ Memory/ControlDependence$, which means that the $instance_1^{th}$ execution of $stmt_1$ is data/control dependent on the $instance_2^{th}$ execution of statement at $line_2$ in $file_2$. For example, by clicking the left expansion mark of line 886 in Figure 2, all the dependence edges originating from this statement in slice 0 are shown just below the source line (in red). From the first line just below line 886, we can see that its first execution instance is data-dependent on the first execution instance of statement at line 815 due to variable *fileptr[0]*. The programmer can navigate backwards along the dependence edge by clicking the "Activate dependent statement" button (e.g., jump directly to the definition point of *fileptr[0]* at line 815). Source code navigation along dependence edges can greatly enhance programmers' debugging efficiency.

Following the dependence edges from the crash point of line 1252, the programmer knows that it is data-dependent on *strcpy* called at line 886 due to an unexpected write access to addresses `0xbf8a9a8c`, `0xbf8a9a88`, and `bf8a9a84` (see the first three dependence edges below line 1252 in Figure 3). Experienced programmers will know that there is something wrong with the invocation of *strcpy*. They can rollback the program state to a previous checkpoint, and then use execution suppression to suppress the abnormal data flow and verify that the root cause is *strcpy* invocation.

Because the crash point is also control dependent on the statement at line 827 (see fourth dependence edge below line 1252, and line 827 in Figure 5), less-experienced programmers may navigate along this dependence edge. If the programmer navigates to line 828, the invocation location of `comprexx`, he can quickly narrow the faulty region by either applying suppression (line 828, Figure 4) or predicate switching (lines 825 or 827, Figure 5). In both cases the crash goes away. Therefore, the programmer will have high confidence that `comprexx` is faulty. Note that `comprexx` may be invoked multiple times, e.g., when *ncompress* detects multiple files in a folder. Using our debugger, the programmer can easily control which instance to alter. With the help of state alteration, the programmer can quickly zoom into the faulty function `comprexx`. Next, the programmer can rollback to a previous

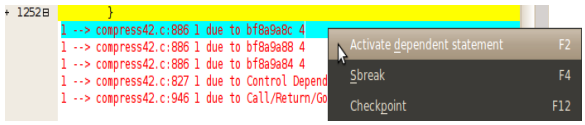


Fig. 3. Slicing from the crash point.

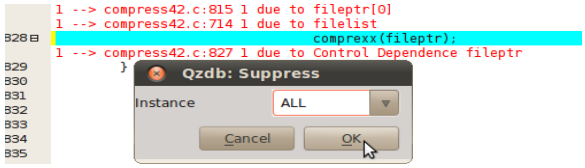


Fig. 4. Execution suppression.

checkpoint and rerun the program up to the beginning of this function, and then efficiently step through `comprexx` with the help of `sbreak all`.

When using slicing, the programmer can use the `prune` command to reduce the size of slices as shown in Figure 6. For example, by pruning the slice by `fileptr`, 17% of the original statements in slice 0 are pruned away, and 37% of the dependence edges are eliminated. Programmers can also generate slices limited to function `comprexx` by simply recording just the execution of `comprexx`—doing so reduces the number of statements in slice 0 by 60% and dependence edges by 62%. Therefore, effective use of partial logging can greatly reduce slice sizes. The above process can be repeated for more complicated applications.

Additional Case Studies: Our case studies are based upon five different kinds of memory-related bugs listed in Table II, taken from BugNet [19]. The stack smashing bug was already discussed; we summarize the four other kinds of bugs due to space limitation.

Tidy-34132: This version contains a double-free memory bug which manifests itself when the input HTML file contains a malformed *font* element, e.g., of the form `<font color="red"<?font>`. The relevant code for this bug is presented in Figure 7. The program constructs a *node* structure for each element (e.g., *font*) in the HTML file. An element may contain multiple attributes corresponding to the *attributes* field of the *node* structure, which is a pointer to the *attribute* structure. The program pushes a deep copy of the *node* structure into the stack when encountering an inline element (i.e., *font* in our test case) by calling *PushInline* (line 057). The deep copy is performed by duplicating the dynamically allocated structure pointed by each field in the *node* structure as well as fields of fields recursively. However, the program makes a shallow copy of the *php* field in the *attribute* structure by mistake in line 033 because of the missing statement, as shown in line 039. All the copies of *node* structure pushed into the stack by *PushInline* will be subsequently popped out in function *PopInline* (line 097), where all the allocated regions will be freed recursively. In some situations, due to the shallow copy, the *php* fields of some *node* structures will contain dangling pointers. If some element in the HTML file is empty and can be pruned out, the program removes the node from the markup tree and discards it by calling *TrimEmptyElement* (line 309) which eventually calls *DiscardElement* on line 316. Node

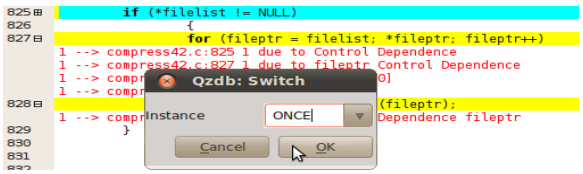


Fig. 5. Predicate switching.

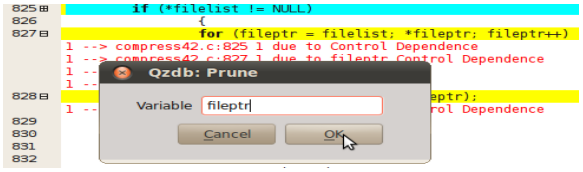


Fig. 6. Pruning a slice.

deletion is just a reverse process of node deep copy, i.e., free all the dynamic allocated memory regions in the *node* structure in a recursive fashion, including the structures pointed by the *php* fields. With some special HTML files as input, the program crashes when it tries to trim the empty *font* element because the *php* field of the *attributes* field of the *font* element has been freed in *PopInline*.

Since the bug is very complicated, debugging is very time-consuming with traditional debuggers. Programmers can identify the bug much easier with the help of our debugger. Although a double-free bug may manifest itself far away from the second *free*, the program happens to crash at the second *free* in our test case. As mentioned before, a program crash at *free* can be caused by three kinds of bugs—double free, unmatched free, or heap buffer overflow. The programmer can use *breakpoint free/malloc/memcpy/strcpy if condition* to identify the exact bug type. In our test case, the command *breakpoint free if arg1==second_free_ptr* captures the position of the first free quickly. The zoom component of our approach now reveals its power, as the reported crash point can be far from the second *free*. The programmer uses the `slice` command to get the dynamic slice of the memory units used at the crash point and then pinpoint the root cause with the state alteration, inspection and rollback interfaces introduced in our debugger.

As we can see, fixing this bug (line 039 in istack.c) calls for far more program comprehension than the positions of the two *free* calls (line 136 in parser.c), which is the best bug report that existing automatic debugging tools (e.g., Memcheck [22]) can achieve. Suppose the programmer has already known the positions of the two *free* calls with the help of either our debugger or automatic debugging tools. To figure out under which condition the bug manifests itself and then remove the defect, the programmer still needs to resort to debuggers. This example illustrates a normal situation where automatic debugging techniques lag far behind the requirements raised from practical debugging. They can only be a supplement to debuggers rather than a substitution.

The programmer can quickly gain program understanding and fix bugs with the help of our debugger in this case. Suppose the programmer has known the position of the two *free*'s, with the help of either our debugger or other automatic

TABLE II
OVERVIEW OF BENCHMARKS.

Program Name	LOC	Error Type	Error Location
ncompress-4.2.4	1.4K	Stack Smashing	compress42.c:886
tidy-34132	35.9K	Double Free	istack.c:031
bc-1.06	10.7K	Heap Buffer Overflow	storage.c:176
ghostscript-8.12	281.0K	Dangling Pointer Use	ttobjs.c:319
tar-1.13.25	28.4K	Null Pointer Use	incremen.c:180

debugging tools. First, the programmer can generate a dynamic slice for the two variables used in the first and second *free* respectively. Then she can easily find out where the shallow copy comes from by following the data dependence edges related to those variables. For example, by following only two hops along the data dependence edges in the generated slice for the variable used in the first *free*, she can find out that the shallow copy comes from line 033. However, the *DupAttrs* function is expected to generate a deep copy of a given attribute, then the programmer figures out that some statements which should generate the deep copy is missed in this function and she can fix this bug quickly.

Of course, the programmer can also leverage state alteration to gain more program comprehension and then fix the bug. For example, she can use the `switch` command to switch some predicates in the dynamic slice of the crash point (the second free) for better understanding the program behavior and crash condition. For this particular bug, switching the last execution instance of the predicate at line 132, 142, or 311 will make the program function properly. Hence, we can infer that a combination of those predicate is the bug manifestation condition and the bug will disappear with any predicate unsatisfied. The programmer can also suppress some functions in the slice of the crash point to isolate the bug. Suppression of the final invocation of *TrimEmptyElement*, *PopInline*, *PushInline* or *DiscardElement* in our test case can eliminate the crash, which suggests that an abnormal data flow is avoided by following any of the execution suppression. From the result of state alteration, the programmer will know that the program crashes if the last processed element is an inline and prunable element. This provides valuable hints to the programmer and helps fix the bug.

Bc-1.06: This version fails with a memory corruption error because a variable *v_count* is misused (instead of the correct *a_count*). The program performs dynamic array expansion, but due to the misuse, the heap object *arrays* is overflowed and the metadata maintained by the heap allocator is corrupted. Because of the metadata corruption, the programmer has difficulty getting any clue using standard debugging methods; with our debugger, checkpointing and reexecution combined with dynamic slicing reveal a hidden dependence in the array expansion and then via suppression the root cause is revealed.

Ghostscript-8.12: This version contains a dangling pointer dereference bug. Using checkpointing and suppression, users of our debugger quickly find out that suppressing function *i_free_object* causes the bug to go away; slicing the metadata that is supposed to be freed by *i_free_object* reveals an illegal write to it earlier on—the root cause of the bug.

Tar-1.13.25: This version dereferences a NULL pointer

```
istack.c:
025: AttVal *DupAttrs( TidyDocImpl* doc, AttVal *attrs) {
033:   *newattrs = *attrs;
034:   newattrs → next = DupAttrs( doc, attrs → next );  ...
       /* miss the following statement*/
039:   newattrs → php=attrs → php? \
         CloneNode(doc, attrs → php):NULL;
041:}
057: void PushInline( TidyDocImpl* doc, Node *node) {...
092:   istack → attributes = DupAttrs( doc, node → attributes ); }
097: void PopInline( TidyDocImpl* doc, Node *node ) {
142:   if (lexer → istacksize > 0) {...
147:       while (istack→attributes){ ...
151:         FreeAttribute( doc, av ); }
parser.c:
128: Node* DiscardElement( TidyDocImpl* doc, Node *element ) {
132:   if (element){...
136:       FreeNode( doc, element); }
140:}
309: Node *TrimEmptyElement( TidyDocImpl* doc, Node *element ) {
311:    if ( CanPrune(doc, element) ){...
316:       return DiscardElement(doc, element); }
```

Fig. 7. Double Free example.

(variable *entry*) which causes a crash. Dynamic slicing of *entry* indicates earlier suspicious predicates; switching one of them suppresses the bug and reveals that the predicate is incorrect—the root cause of the bug.

In the above case studies we have shown the benefits of our approach in context of several widely-used applications containing different kinds of bugs.

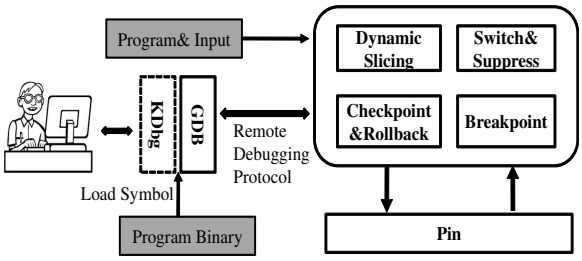


Fig. 8. Components of our debugger.

IV. IMPLEMENTATION

The prototype implementation of our interactive debugging strategy, shown in Figure 8, consists of GDB-based [7] and Pin-based [16], [33] components. The user interfaces with the GDB component via a command line interface or a KDbg [35] based graphical interface. The Pin-based component implements our new debugging commands. The GDB component communicates with the Pin-based component via the remote debugging protocol and it also interprets debug-information in the binary. The Pin-based component implements the new capabilities via dynamic binary instrumentation. The extended KDbg provides an intuitive interface for switching predicates, suppressing execution, setting breakpoints, turning recording on/off, and inspecting and stepping through slices.

TABLE III
RUN CHARACTERISTICS

Program	Test Case Description	Exec. Instr.	Baseline(sec)
ncompress	compress a folder(148KB)	10278947	0.33
tidy	check a HTML file(104 lines)	2125726	0.56
bc	interpret a source file(121 lines)	1846427	0.44
ghostscript	PS to PDF conversion(18KB)	3909749	0.47
tar	create an archive(789K)	4654490	0.51

TABLE IV
SLICING TIME AND SPACE OVERHEAD.

Program	Baseline Time (sec)	Pay-Once Time (sec) and Space (MB) Overhead					Slice Time Overhead (sec)		
		DU		CD		LP			
		Time	Space	Time	Space	Time	AVG	MIN	MAX
ncompress	0.33	5.83	93.63	2.88	63.19	3.28	37.19	13.56	71.52
tidy	0.56	9.45	12.81	4.62	17.60	0.24	31.14	11.43	46.59
bc	0.44	5.58	11.52	2.63	13.51	0.20	14.44	11.53	21.70
ghostscript	0.47	8.28	24.43	4.93	25.58	0.53	20.44	2.95	45.04
tar	0.51	7.23	24.78	3.33	25.14	0.06	6.69	7.74	15.04

Predicate switching. Upon receiving `switch` commands, we use Pin to first invalidate existing instrumentation involving specified code regions and then reinstrument the code to switch the results of predicates by swapping their fall-through and jump targets.

Execution suppression. After the programmer issues a `suppress` command, existing instrumentation is invalidated and new instrumentation is added to skip over the suppressed execution instance of the instruction. The instruction is executed normally if it is not the suppressed execution instance. If all instances are to be suppressed, the instruction is deleted using Pin.

Dynamic slicing. We implement the `slice` command by instrumenting the code to record the PC, dynamic instance, as well as memory region(s) and register(s) read and written by instructions. We instrument both user and library code. We turn off recording when `record off` is encountered. For limiting the time and space overhead of dynamic data dependence graph construction, we use the limited preprocessing (LP) method by Zhang et al. [31]. For accurately capturing dynamic control dependences, we use the online algorithm by Xin and Zhang [27]. The immediate postdominator information is extracted using Diablo [5].

Conditional breakpoint. To implement the extended conditional breakpoint command we invalidate the existing instrumentation and then reinstrument each function call to first check whether the function name is the same as the specified *lib_func*. If so, the instrumentation code evaluates the given *condition* (if any) and triggers a generated breakpoint if it is satisfied.

Checkpointing and rollback. Undo-log based incremental checkpoints [34] are adopted to keep only the modifications between two checkpoints and save space. When a `checkpoint` command is received, we first save the state of all registers maintained by Pin. Subsequently we record the original value of each modified memory cell by instrumenting each memory write operation. Upon a `rollback` command, we restore the logged values to their memory cells and registers. Because Pin cannot track into system calls, we handle system calls and I/O as follows. The system-call side-effects are detected by analyzing commonly-used system calls and recording the memory regions read/written by each system call. For file I/O, whenever a checkpoint is generated, we record the file pointer positions for all open file descriptors. When the program is rolled-back, we restore file pointer positions, so file reads and writes proceed from correct offsets on reexecution. We do not handle interactive I/O specially, but rather offer the expected semantics for reexecution. For example, for the console, after a roll-back, the user must type the input again, and output messages will be printed again. In our experience, this approach works well in practice.

V. PERFORMANCE EVALUATION

Next we show that the time and space overheads for state alteration, inspection, and rollback are acceptable for interactive debugging. To quantify these overheads, we have conducted experiments with the programs from Table II. Since our objective is to measure time and space costs, we used a passing test case to run each application to completion. Table III shows the run characteristics including number of executed instructions and the baseline running time under Pin without instrumentation. All experiments were conducted on a DELL PowerEdge 1900 with 3.0GHz Intel Xeon processor and 3GB RAM, running Linux, kernel version 2.6.18.

Slicing overhead. The time and space overhead for slicing are presented in Table IV. For each program, we turned on the recording to collect definition/use information and detect dynamic control dependences for the whole execution. We then applied limited preprocessing (LP) to the generated def-use information to get a summary of all downward exposed definitions of memory addresses and registers for each trace block. Using the generated trace and summary, we computed slices for the last twenty statements.

The *Pay-Once Time* and *Space Overhead* columns 3–7 in Table IV show the time and space overhead which is only incurred once and amortized over all subsequent slice computations. The pay-once time overhead is further broken down into time overhead for recording definition/use (*DU*), control dependence (*CD*), and preprocessing of the generated def-use information (*LP*). The pay-once space overhead is broken down into space overhead for definition/use information recording (*DU*) and control dependence (*CD*) as the space overhead for LP is relatively insignificant.

The average (*AVG*), minimum (*MIN*), and maximum (*MAX*) slice computation times are given in the *Slice Time Overhead* column. We observe that the time overhead for slice is not greatly dependent on the position of the slice criterion. Instead, it is dominated by the nature of slice criterion and program behavior. Most slice computations can be done in 1 min.,

TABLE V
TIME AND SPACE OVERHEAD: DU & CD

Program	MS/K instructions	KB/K instructions
ncompress	0.85	15.62
tidy	6.62	14.65
bc	4.45	13.88
ghostscript	3.38	13.10
tar	2.27	10.98
average	*3.51*	*13.65*

TABLE VI
CHECKPOINTING AND ROLLBACK TIME AND SPACE OVERHEAD.

Program	Num. of Checkpoints	Time (sec)	MS/K instructions	Space (KB)	KB/K instructions	Rollback Time (millisecond)
ncompress	11	8.93	0.87	28547.6	2.78	356.17
tidy	3	9.31	4.38	233.4	0.11	4.02
bc	2	6.06	3.28	45.1	0.02	0.04
ghostscript	4	8.52	2.38	788.9	0.20	12.30
tar	5	7.40	1.80	189.6	0.04	0.22

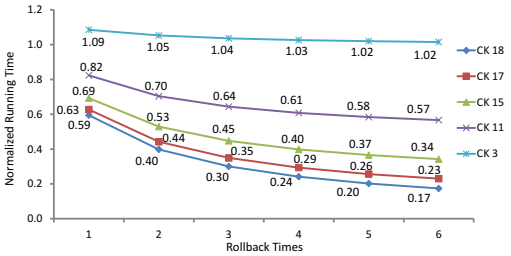


Fig. 9. Runtime savings due to rollback.

which is acceptable considering the large amount of time spent on debugging by programmers.

The time and space overhead of both def-use information recording and control dependence detection per 1K instructions are given in the second and third columns of Table V, respectively. The time overhead ranges from 0.85ms to 6.62ms per 1K instructions and the average overhead is 3.51ms per 1K instructions. The space overhead ranges from 10.98KB to 15.62KB per 1K instructions and the average overhead is 13.65KB per 1K instructions. We believe that the pay-once time and space overheads for dynamic slicing are acceptable.

Checkpointing overhead. The time and space overhead of checkpointing are given in Table VI. This data corresponds to checkpointing every one million instructions. The second column shows the number of checkpoints generated. The total program execution time with incremental checkpointing is given in the third column. The fifth column presents the total space overhead of the generated checkpoints. The time needed to rollback a program from the end to the beginning, which represents the largest distance the programmer can rollback the program, is shown in the last column. The benchmarks reveal that, the larger the size of the generated checkpoints, the longer it takes to rollback the program; *ncompress* incurs the largest space overhead for the 11 checkpoints and it requires the longest time to rollback the program to the beginning.

The time and space overhead of incremental checkpointing per 1K instructions is given in the fourth and sixth columns of Table VI, respectively. As we can see, the time overhead ranges from 0.87ms to 4.38ms per 1K instructions, while the space overhead ranges from 0.02KB to 2.78KB per 1K instructions. Compared to the time and space overhead of recording and control dependence shown in Table V, the time and space overhead per 1K instructions for incremental checkpointing is much lower. This is because only memory write instructions need to be instrumented for incremental checkpointing, while both memory and register read and write instructions need to be instrumented for recording.

Efficiency of state rollback. As mentioned in Section II-C, our state rollback command replaces the rolled-back part of execution by altered program execution (e.g., due to feeding it a different input or switching control flow) in the log. Thus, programmers have no need to rerun the program from the beginning. Of course, to rollback the program state, programmers have to pay the checkpointing overhead during the initial full run. In this experiment, we emulate a traditional debugging process and compare the running time with and without use of state rollback. We consider a run of *bc* that takes 118 seconds in null pin mode and executes 36.2×10^{10} instructions. A checkpoint is made every $2 * 10^{10}$ instructions leading to 19 checkpoints numbered from 0 to 18. We compare the execution time with use of rollback to different checkpoints (CK 3, CK 11, CK 15, CK 17, CK 18) for varying number of times (1 through 6) with the execution time without use of rollback. The execution times with rollback, normalized with respect to the corresponding times without rollback, are shown in Figure 9. We observe that the execution time savings due to use of rollback are substantial and higher for more recent checkpoints (e.g., CK 18) and the savings increase with the number of times rollback is performed (e.g., rollback six times). However, if the rollback is performed to an early checkpoint (e.g., CK 3), its benefit disappears.

TABLE VII
SUPPRESSION TIME OVERHEAD.

Program	Baseline (sec)	Suppress with Recording and Checkpointing (sec)	Suppress with Checkpointing Only (millisecond)
ncompress	0.33	0.60 (182.13%)	3.33 (1.01%)
tidy	0.56	0.13 (22.51%)	24.00 (4.29%)
bc	0.44	0.23 (52.02%)	6.67 (1.52%)
ghostscript	0.47	0.35 (74.04%)	24.80 (5.28%)
tar	0.51	0.09 (18.21%)	1.85 (0.36%)

State alteration overhead. The time overhead of execution suppression is given in Table VII. We consider two scenarios. The first scenario simulates the case where the programmer suppresses a statement with both recording and incremental checkpointing turned on, while in the second scenario only incremental checkpointing is turned on. The data presented is averaged over suppressing 10 statements spread around the middle of the execution. We observe that performing execution suppression in the first scenario incurs substantially higher runtime overhead. This is because in this scenario an execution suppression command invalidates many existing instrumentations, leading to more future reinstrumentation costs, and higher runtime overhead. From Table VII, we can see that the average time overhead incurred by execution suppression ranges from 0.36% to 182.13% compared to the baseline. This overhead is acceptable and a worthy trade-off for the benefits of our approach. We omit presenting the overhead for predicate switching as it is similar to the overhead of execution suppression due to similarity in the implementation.

VI. RELATED WORK

Debugging Assistance. Dynamic slicing has been widely recognized as helpful for debugging [15], [31], [27], [29]. The Whyline [15] tool allows programmers to ask questions about program behavior and it responds with causal chains of events.

Techniques have been developed to automatically generate breakpoints based upon automated fault location [10], [29]. Chern et al. [3] improve breakpointing by allowing control-flow breakpoints. The *vsdb* interactive debugger uses symbolic execution to display all possible symbolic execution paths from a program point [9]. Coca [6] allows programmers to query the execution trace. However, none of these approaches take advantage of state alteration.

Fault Localization. State alteration has been used to automatically localize faults [30], [13], [2]. Zhang et al. [30] proposed predicate switching to constrain the search space of program state changes explored during bug location. Corrupted memory location suppression [13] attempts to identify the root cause of memory bugs by iteratively suppressing the cause of the memory failure. Chandra et al. [2] report repair candidates based on value replacement. Gu et al. [8] propose a bug query language allowing programmers to fix their bugs by referring to similar resolved bugs.

Delta debugging [4], [28] has been applied to automatically identify cause-effect chains [28] and recognize cause transitions [4]. Renieris and Reiss [25] identify faulty code by considering differences in statements executed by passing and failing runs. Tarantula [14] prioritizes statements based on their appearance frequency in failing runs versus passing runs. In general, these approaches rely on a large test suite that may not be available in practice. Limitations of these techniques (e.g., commands) motivate our work.

Memory-related Fault Detection, Tolerance, and Correction. Purify [11] and Valgrind [22] detect the presence of memory bugs via dynamic binary instrumentation. CCured [21] uses type inference to classify pointers and applies dynamic checks according to the classfication for memory safety. Rx [24] recovers from a crash by rolling back execution and reexecuting after changing the execution environment. AccMon [32] detects memory-related bugs by capturing violations of program counter based invariants. DieHard tolerates memory errors through randomized memory allocation and replication [1]. Exterminator dynamically generates runtime patches based upon runtime information [23]. Nagarakatte et al. use compile-time transformation for spatial [17] and temporal safety [18] of C.

VII. Conclusion

We have presented a novel interactive debugger that offers powerful state alteration and state inspection capabilities. State alteration commands enable programmers to narrow down the potential faulty code and ascertain their conjectures efficiently. State inspection commands enable programmers to comprehend program behavior and the nature of the bug rapidly. Case studies on real reported bugs as well as performance evaluation demonstrate the effectiveness and efficiency of our debugger.

Acknowledgment

This research is supported by the National Science Foundation grant CCF-0963996 and a grant from Intel Corporation to the University of California, Riverside.

References

[1] Berger, E. D. and Zorn, B. G. DieHard: Probabilistic memory safety for unsafe languages. *PLDI*, pages 158–168, 2006.
[2] Chandra, S., Torlak, E., Barman, S., and Bodik, R. Angelic debugging. *ICSE*, pages 121–130, 2011.
[3] Chern, R. and De Volder, K. Debugging with control-flow breakpoints. *AOSD*, pages 96–106, 2007.
[4] Cleve, H. and Zeller, A. Locating causes of program failures. *ICSE*, pages 342–351, 2005.
[5] De Bus, B., De Sutter, B., Van Put, L., Chanet, D., and De Bosschere, K. Link-time optimization of arm binaries. *LCTES*, pages 211–220, 2004.
[6] Ducassé, M. Coca: an automated debugger for c. *ICSE*, pages 504–513, 1999.
[7] GDB 2011. http://www.gnu.org/software/gdb/.
[8] Gu, Z., Barr, E., and Su, Z. Bql: capturing and reusing debugging knowledge. *ICSE*, pages 1001–1003, 2011.
[9] Hähnle, R., Baum, M., Bubel, R., and Rothe, M. A visual interactive debugger based on symbolic execution. *ASE*, pages 143–146, 2010.
[10] Hao, D., Zhang, L., Zhang, L., Sun, J., and Mei, H. Vida: Visual interactive debugging. *ICSE*, pages 583–586, 2009.
[11] Hastings, R. and Joyce, B. Purify: Fast detection of memory leaks and access errors. *USENIX Winter Tech. Conf.*, pages 125–136, 1992.
[12] IBM tutorial on debugging 2011. IBM tutorial on debugging–Debugging using comments.
[13] Jeffrey, D., Gupta, N., and Gupta, R. Identifying the root causes of memory bugs using corrupted memory location suppression. *ICSM*, pages 356–365, 2008.
[14] Jones, J. A. and Harrold, M. J. Empirical evaluation of the tarantula automatic fault-localization technique. *ASE*, pages 273–282, 2005.
[15] Ko, A. and Myers, B. Debugging reinvented:asking and answering why and why not questions about program behavior. *ICSE*, pages 301 –310, 2008.
[16] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. *PLDI*, 2005.
[17] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Softbound: highly compatible and complete spatial memory safety for c. *PLDI*, 2009.
[18] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Cets: compiler enforced temporal safety for c. *ISMM*, pages 31–40, 2010.
[19] Narayanasamy, S., Pokam, G., and Calder, B. BugNet: Continuously recording program execution for deterministic replay debugging. *ISCA*, pages 284–295, 2005.
[20] ncompress 2011. Ncompress: a fast, simple lzw file compressor. http://ncompress.sourceforge.net/.
[21] Necula, G. C., McPeak, S., and Weimer, W. CCured: type-safe retrofitting of legacy code. *POPL*, pages 477–526, 2002.
[22] Nethercote, N. and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *PLDI*, pages 89–100, 2007.
[23] Novark, G., Berger, E. D., and Zorn, B. G. Exterminator: Automatically correcting memory errors with high probability. *PLDI*, pages 1–11, 2007.
[24] Qin, F., Tucek, J., Zhou, Y., and Sundaresan, J. Rx: Treating bugs as allergies – a safe method to survive software failures. *ACM TOCS 25,* 3, Article 7 (1–33), 2007.
[25] Renieris, M. and Reiss, S. Fault localization with nearest neighbor queries. *ASE*, pages 30–39, 2003.
[26] Visual Studio Debugger 2011.
[27] Xin, B. and Zhang, X. Efficient online detection of dynamic control dependence. *ISSTA*, pages 185–195, 2007.
[28] Zeller, A. Isolating cause-effect chains from computer programs. *FSE*, pages 1–10, 2002.
[29] Zhang, C., Yan, D., Zhao, J., Chen, Y., and Yang, S. BPGen: an automated breakpoint generator for debugging. *ICSE*, 2010.
[30] Zhang, X., Gupta, N., and Gupta, R. Locating faults through automated predicate switching. *ICSE*, pages 272–281, 2006.
[31] Zhang, X., Gupta, R., and Zhang, Y. Precise dynamic slicing algorithms. *ICSE*, pages 319–329, 2003.
[32] Zhou, P., Liu, W., Fei, L., Lu, S., Qin, F., Zhou, Y., Midkiff, S. P., and Torrellas, J. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. *MICRO*, pages 269–280, 2004.
[33] Lueck, G., Patil, H., Pereira, C. PinADX: an interface for customizable debugging with dynamic instrumentation. *CGO*, pages 114–123, 2012.
[34] King, S. and Dunlap, G. and Chen, P. Debugging operating systems with time-traveling virtual machines. *ATEC*, pages 1–1, 2005.
[35] Kdbg homepage http://www.kdbg.org/.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

Proteum/FL: a Tool for Localizing Faults using Mutation Analysis

Mike Papadakis*, Marcio E. Delamaro†, Yves Le Traon*
*Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg
Email: {michail.papadakis, yves.letraon}@uni.lu
†Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, SP, Brazil
Email: delamaro@icmc.usp.br

***Abstract*—Fault diagnosis is the process of analyzing programs with the aim of identifying the code fragments that are faulty. It has been identified as one of the most expensive and time consuming tasks of software development. Even worst, this activity is usually accomplished based on manual analysis. To this end, automatic or semi-automatic fault diagnosis approaches are useful in assisting software developers. Hence, they can play an essential role in decreasing the overall development cost. This paper presents Proteum/FL, a mutation analysis tool for diagnosing previously detected faults. Given an ANSI-C program and a set of test cases, Proteum/FL returns a list of program statements ranked according to their likelihood of being faulty. The tool differs from the rest of the mutation analysis and fault diagnosis tools by employing mutation analysis as a means of diagnosing program faults. It therefore demonstrates the effective use of mutation in supporting both testing and debugging activities.**

***Keywords*-Mutation analysis; fault localization; software debugging; software testing;**

I. Introduction

Detecting and fixing program faults forms an essential and expensive task of software development. Developers rely on software testing for identifying program flaws and software debugging for diagnosing and removing them. The process of identifying flawed program parts given some failures is usually referred to as fault localization. This process takes place just after identifying program failures and is one of the most expensive activities of software debugging [1].

Mutation analysis forms a powerful technique usually used for driving the testing process. Generally, it seeds some defects into the program under investigation in order to examine its behavior. Considering the testing process, finding artificial defects allows the effective detection of real faults. This practice has been empirically shown to be more effective than using classical test selection criteria (e.g. based on code coverage). In view of this, the present paper explores the idea of using artificial defects as a means of diagnosing real defects. Therefore, mutation analysis is employed in order to guide the fault localization process.

Existing fault localization techniques assist programmers by ranking program locations according to their probability of being responsible for the experienced failures. The underlying idea of these approaches is to compare the similarity between the execution of these code places and the observed failures. Researchers have provided evidence that these approaches are helpful in reducing the cost involved in the debugging activity [2], [3].

Testing using mutation analysis involves designing and executing tests with a set of artificial faults called mutants. The main assumption of the method is that the utilized defects (mutants), despite being artificial defects, behave like real faults. Evidence in support of such a proposition have been given by Andrews et al. [4]. Furthermore, software testing research has successfully demonstrated that detecting mutants results in detecting real faults. Therefore, it can be asked: what is the relation between the mutants' location and the existing faults? Is there any link between the mutants' location with the location of the faults?

The paper presents Proteum/FL a tool that localizes faults using mutation analysis and thus, answering the above questions. The tool was initially used in the work of Papadakis and Le Traon [5] for defining a mutation-based fault diagnosis approach and it can be used to support both testing and diagnosis activities. Previous research on this topic [5] has shown that the implemented approach significantly outperforms the coverage-based fault localization techniques and opens a new direction, mutation-based debugging, to the mutation analysis research.

The remainder of this paper is organized as follows: Section II presents the concepts and details regarding the mutation analysis. Section III and IV respectively introduce fault localization using mutation analysis and describe the implementation details of Proteum/FL. Finally, the relevance of Proteum/FL with other tools and the conclusions made during this research are discussed in Sections V and VI respectively.

II. Mutation Analysis

Mutation analysis injects artificial defects into the program under investigation with the aim of examining its behavior when executed with some test cases. These defects are called *mutants* and they are generated by using a set of simple syntactic rules, called *mutant operators*. Mutants are traditionally utilized to facilitate the testing process here referred to as mutation testing. Proteum [6] is a mutation testing tool that aims at automating the testing process of

978-1-4673-5739-5/13/$31.00

C programs. This section gives a brief description of the mutation testing process, the supported mutant operators and the Proteum mutation testing system.

A. Mutation Testing

Software testing involves the examination of a program by executing a set of test cases. However, in practice there is a need to evaluate the appropriateness of the utilized test sets. Further, in case that the utilized test sets are not adequate, there is a need to guide the design of new test cases. To this end, mutation testing aims at guiding the testers to design sets of test cases and evaluate their adequacy.

Mutation testing requires tests capable of making the seeded defects observable. To this end, a comparison of the programs' outputs is needed in order to decide whether a difference in behavior between the original and the mutant program versions has been triggered. In such cases, the mutants are said *killed*, otherwise they are said *live*. Testing adequacy is measured by the percentage of mutants that are killed by the utilized tests. Unfortunately, not all the mutants can be killed. A mutant for which there is no test data that distinguish its behavior from the original program is said *equivalent*. Therefore, testing adequacy, called *mutation score (MS)*, is measured by the following value:

$$MS = \frac{No.\ killed\ mutants}{No.\ mutants - No.\ equivalent\ mutants}$$

Generally, mutation testing relies on the quality of the involved mutants [7]. The application of the method relies on two hypothesis, the "competent programmer" and the "coupling effect" [7]. The "competent programmer" hypothesis states that the programmers produce programs that are close of being "correct". Thus, only small changes are necessary to effectively exercise the program under test. The "coupling effect" hypothesis states that "Test data that distinguishes all program differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors". Since mutants represent simple faults, the above hypothesis suggests that killing mutants results in revealing both simple and complex faults.

B. Mutation Operators

Proteum employs mutation operators targeting at unit and integration testing faults [8]. The unit-level operators were designed based on the study of Agrawal et al. [9] while the integration testing operators were designed according to the study of Delamaro et al. [8]. Here it should be noted that the focus of the present paper is on locating unit level faults and thus, mutants related to integration testing are not discussed. The employed operators are divided into four main categories (classes): a) STATEMENT b) VARIABLES, c) CONSTANTS and d) OPERATORS. The STATEMENT class contains operators that alter an entire statement or its key syntactic elements. The VARIABLES and CONSTANTS classes contain operators regarding program identifiers and constants, respectively. They model incorrect variable and constant uses. The OPERATORS class contains operators that alter the programming language operator use.

C. Proteum & Proteum/IM

Proteum [6] is among the firstly and most popular mutation testing tools. Originally, it was designed to perform mutation at the unit level by utilizing the mutant operators described in the previous section. It was then extended and named as Proteum/IM, to support both unit and integration testing by utilizing the Interface Mutation [8] approach. A detailed description of the functionality and the implementation of the Proteum/IM tool can be found at [6]. The latest version of Proteum/IM was recently released as an open source software under the "GNU GENERAL PUBLIC LICENSE". It is this version of the tool that it is extended by Proteum/FL and can be found at:

$$http://ccsl.icmc.usp.br/projects/proteum$$

III. Proteum/FL Fault Localization Process

Fault localization is the process of identifying the program places which are responsible for provoking program failures. Typically, fault localization is performed after the testing process. Generally, the testing process involves the design, execution of some test cases and the determination of whether the program behaves as expected. A failure is experienced when the output of a test differs from the expected one, as specified by the tester. The test that results in a failure is called a *failed test*. In the opposite case, the test is called as a *passed test*. To this end, fault localization tries to identify the program places that are responsible for the program failures given a set of failed tests and a set of pass tests. These tests are those used during the testing process. They might have been produced using a test strategy like [10], in a semi-automated way i.e. [11], [12] and [13] or in a manual way. Therefore, given a test suite, fault localization tries to highlight the program statements that are likely to be faulty. Then, the tester will inspect these statements in order to identify the faulty program location.

The underlying idea of most fault localization approaches is to define a suspiciousness metric that measures the probability of a statement to be faulty. The definition of the suspiciousness metric is based on the observation that program failures are the manifestation of faults. Thus, it is natural to expect that failed tests execute program places that correlate with faults. On the contrary, passed tests should execute program places that do not correlate with faults. In view of this, most of the fault localization approaches try to measure the similarity/dissimilarity between the failed/passed tests with the execution of program statements. Among the various similarity measures Proteum/FL uses the Ochiai formula (presented in Table I) [14]. Ochiai is one of the most popular measures and it has been empirically found to

Table I
THE OCHIAI FORMULA

$$Suspiciousness(e) = \frac{failed(e)}{\sqrt{totfailed * (failed(e) + passed(e))}}$$

Where: **Suspiciousness(e)** is the probability of code element e to be faulty. **totfailed** is the total number of failed tests, **failed(e)** is the number of failed tests that cover (kill) the e code element (mutant) and **passed(e)** the number of pass tests that cover (kill) the e code element (mutant).

be one more effective than some other alternatives. Here, it should be noted that other measures could be used as well. The interested reader can refer to [15] for further details.

Despite the research made in the area of fault localization, existing approaches are far from satisfactory in many situations. This is due to the so-called *coincidental correctness problem* [16]. Coincidental correctness occurs when tests execute the faulty elements but fail to manifest it to a failure. Therefore, the similarity measure between the failed/passed tests with the execution of program statements turns to be missleading. To improve the effectiveness of fault localization, it has been proposed to use mutation analysis [5]. Contrary to structural testing criteria, mutation works with the programs' outputs (it requires the mutants to have an effect to the program output). As a consequence, it can simulate well situations having coincidental correctness.

Fault localization using mutation analysis measures the similarity between the test failed/passed with their respective results on kill/live mutants [5]. To accomplish this, any similarity measure can be used [15]. As pointed out before, Proteum/FL uses the Ochiai formula for this purpose by considering the mutants as the formula elements (e). The aim of the process is to measure the similarity between failures and killed mutants. Thus, in Table I the "execute the e code element" requirement represents a mutant that is killed [5]. Therefore, for each mutant (e) a suspiciousness value is assigned. These values are then ranked in order to get a priority list with respect to decreased suspiciousness values. Since every mutant is created based on syntactic changes, a direct mapping from mutants to program statements can be made. If a mutant alters more than one statements, all these are assigned with the same suspicious value.

The overall fault localization approach is outlined in Figure 1. This approach is implemented in the Proteum/FL tool. The description given in the following section concerns the adaptation of Proteum mutation testing tool (see Section II.B) to effectively localize faults.

IV. PROTEUM/FL

Proteum/FL supports the entire fault localization process using mutation analysis as described in [5]. It automatically performs the generation and execution of mutants and it also reports the suspicious program statement list. This section describes the implementation of the tool, its functionality and some important optimization techniques that it uses.

A. Description

Automatically performing mutation analysis with Proteum/FL requires some parameters to be specified. The user can specify the mutant operators that are going to be used and the way (command or script) that the mutants should be compiled. Additionally, the user need to specify a comparison method that will be used to determine the killed and live mutants. By default, a comparison is made based on the programs resulting values and its printable output. This can be extended by defining a program specific method like comparison of output files etc. The definition can be a perl or shell script. The tool will then automatically do all the required tasks to report the suspicious program statements to the tester.

Proteum/FL has been implemented in Perl and works as a command line tool. It implements the mutants generation, execution and fault localization tasks as described in the next section. The high level architecture is depicted in Figure 2. It consists of three main components: Proteum, the Mutant Generation and the Test Execution Engine components. The Proteum component is actually the command line interface of the Proteum/IM tool. It takes as input a C program and produces the mutants' description file [6]. This file contains information containing the definition of each mutant, its program location, its type and its identification number. This file is then used by the Mutant Generator component in order to generate and compile the sought mutants. To accomplish this task, GCC[1] and Gcov[2] tools are used. Gcov is employed for collecting trace information of the original program and the executable program statements. GCC is used for compiling the mutants. The mutants and the original programs are passed to the Test Execution Engine which performs the execution of the mutants with the available tests and produces a suspicious statement report.

B. Functionality of Proteum/FL

The Proteum/FL tool provides the infrastructure for locating program faults by performing the following steps:

Mutant selection: Proteum/FL applies by default all supported mutant operators. However, it is possible to use a smaller set of mutants or operators. This practice often results in similar results with the default one, with a lower computational cost. Thus, the user can specify a selective set of operators to apply. Alternatively the user can select applying a random percentage of mutants.

Generate mutants: Proteum/FL relies on the Proteum/IM [6] to identify the possible mutants by generating the mutants' description file [6]. Proteum/FL then reads both the source code of the program under test and the mutants'

[1]GNU Compiler Collection, http://gcc.gnu.org/
[2]Gcov is a GNU code coverage tool part of GCC

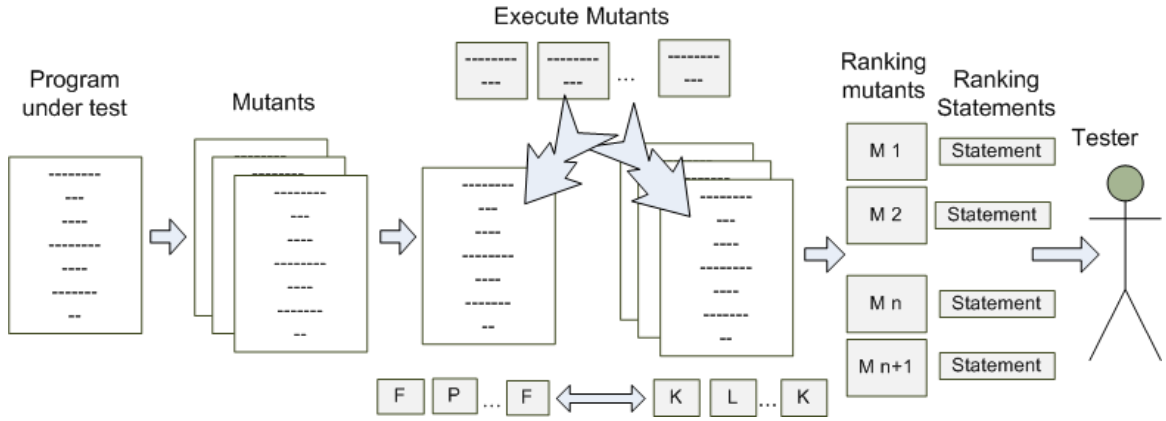


Figure 1. Proteum/FL fault Localization process: Initially, a set of mutants is produced. These mutants are then executed with all the available test cases in order to determine which ones are killed by each one of the utilized test cases. Then a similarity comparison between the failed/passed test cases and the killed/live status of each mutant is performed using the Ochiai formula (I). Then, all the mutants are ranked according to the similarity values computed in previous step. Finally, a ranking list of statements is obtained based on the ranking of mutants and it is reported to the tester.

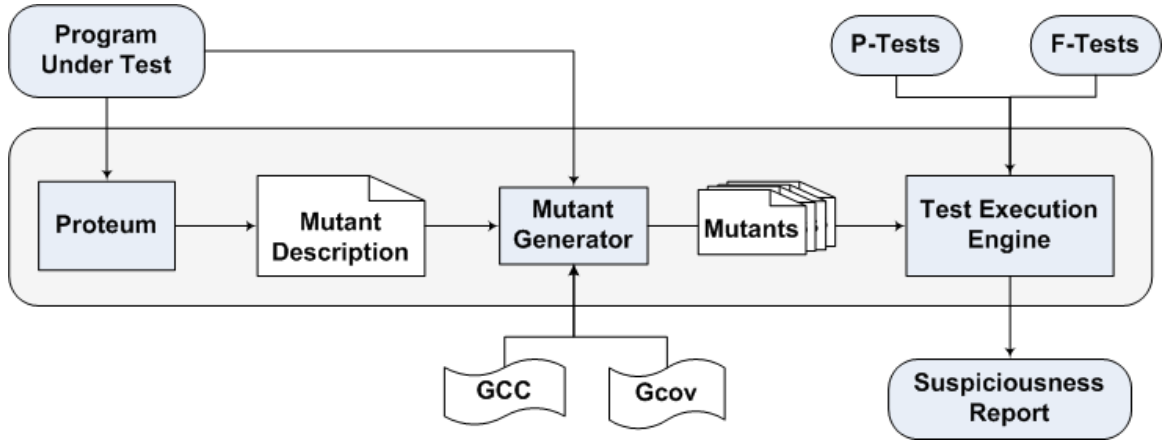


Figure 2. Proteum/FL Architecture

description file and produces the mutants' source code. It actually produces and compiles one source file per mutant. In literature this technique is called *separate compilation* [17] or *compiler-based* [18] technique. Although, more advanced techniques exists like *Mutant Schemata* [7], [17] it was chosen due to its implementation simplicity. Further, by doing so it is possible to construct a test harness. In the *test harness technique* [17] "each mutant is compiled into a shared library that can be dynamically invoked by a test harness". Thus, during the test execution mutants can be efficiently invoked [17] resulting in a reduced execution cost.

Execute mutants: Proteum/FL takes as inputs the failed and passed test cases and executes all the mutants with all tests as required by the mutation analysis technique. It then produces a matrix containing the information of which mutants are killed by each test. This is a computationally expensive part of the process. To efficiently perform it Proteum/FL implements a wide range of optimizations. These are detailed in the next section (Optimizing the Execution of Mutants).

Collect & analyze data: Proteum/FL reads the produced matrices and generates the Suspiciousness Report. This is performed based on the use of the Ochiai formula (Table I).

Rank program statements: Proteum/FL aim is to report the tester a ranked list of program statements. This list assist the tester in locating the program faults by inspecting the most suspicious program statements first. Figure 3 depicts the suspicious statement report produced by the Proteum/FL on a sample program. This report, contains the information about the line of the statements, their suspicious values and the mutants that have these suspiciousness values. It is noted that the statements are ranked according to a decreased suspicious order. Thus, in the report of Figure 3, the most suspicious statement has a suspiciousness value 1.0. This value is assigned to the line 298 based on the suspiciousness of the mutants 1075, 1053 etc.

```
Line 298, Susp:1.0, Mutant=1075,1053,912,951,950,948,953,958,957,956,879,728,
752,743
Line 275, Susp:0.7453559924999299, Mutant=292
Line 301, Susp:0.48666426339228763, Mutant=778,781,780,783,795,796,798,790,
791,811,808,812,806,817,759,758,763
Line 289, Susp:0.47140452079103173, Mutant=461,462
Line 271, Susp: 0.3849001794597505, Mutant=179
Line 366, Susp:0.3636964837266, Mutant=2869,2681,2593,2618,2619,2555,2466
Line 299, Susp:0.3585685828003181, Mutant=1057,702
Line 282, Susp:0.2431083191631576, Mutant=555
Line 307, Susp:0.2401922307076307, Mutant=1051
Line 324, Susp:0.21884405476620425, Mutant=1900
Line 328, Susp:0.2132007163556104, Mutant=1654
Line 329, Susp:0.18534061896456466, Mutant=1583,1630
Line 283, Susp:0.18190171877724975, Mutant=540,526
Line 363, Susp:0.1770844008302866, Mutant=2416
Line 291, Susp:0.1767766952966369, Mutant=550
Line 315, Susp:0.17437145811572893, Mutant=1182
Line 311, Susp:0.16718346377260584, Mutant=1121
Line 337, Susp:0.16539535392599136, Mutant=1777,1427,1426,1240,1239,1238
Line 334, Susp:0.16012815380508713, Mutant=2086
Line 332, Susp:0.13483997249264842, Mutant=2062
Line 362, Susp:0.13456839120487699, Mutant=2804
Line 330, Susp:0.13143238630149706, Mutant=1817
Line 361, Susp:0.12734290799340264, Mutant=2396
Line 323, Susp:0.12067769800636945, Mutant=1796
Line 365, Susp:0.1178511301977579, Mutant=2551,2552
Line 326, Susp:0.11080752827296766, Mutant=1808,1814
Line 339, Susp:0.10027894056973133, Mutant=2250
Line 322, Susp:0.09037128496931669, Mutant=1791
Line 357, Susp:0.08806109155784352, Mutant=2512,2511
Line 345, Susp:0.07552100405338, Mutant=2870,2635,2631,2629,2628,2625,2686,
2620,2622,2715,2714,2719,2718,2717,2716,2720,2721,2722,2723,2724,2725,2496,
2487,2486,2485,2484,2483,2482,2481,2495,2494,2493,2492,2491,2490,2489,2488
Line 325, Susp:0.07552100405338862, Mutant=2175,1731,1771,1396,1397
Line 265, Susp:0.07552100405338862, Mutant=107,20,18,16,15
Line 264, Susp:0.07552100405338862, Mutant=106,6,4,2,0
Line 267, Susp:0.07552100405338862, Mutant=109,48,45,47,43
Line 266, Susp:0.07552100405338862, Mutant=108,32,34,31,29
Line 364, Susp:0.07552100405338862, Mutant=2655,2705,2428
Line 359, Susp:0.07552100405338862, Mutant=2650,2700,2363
Line 360, Susp:0.07552100405338862, Mutant=2651,2701,2376
Line 358, Susp:0.07552100405338862, Mutant=2649,2699,2350
Line 355, Susp:0.07552100405338862, Mutant=2646,2696,2311
Line 356, Susp:0.07552100405338862, Mutant=2647,2697,2324
Line 367, Susp:0.07552100405338862, Mutant=2658,2682,2708
Line 354, Susp:0.07552100405338862, Mutant=2645,2695
Line 368, Susp:0.07552100405338862, Mutant=2709
```

Figure 3. Statement Suspiciousness report

C. Optimizing the Execution of Mutants

Performing mutation analysis requires executing the tests with a huge number of mutants. This process aims at determining the mutants that are killed by each one of the utilized tests and it involves vast computational resources. In order to be efficient, Proteum/FL implements the following optimizations:

Coverage data: Generally, a mutant not executed by a test case has no chance of being killed and thus, its execution results in a waste of time. In view of this, test execution should only focus on mutants that are executed, i.e. test execution reaches the mutated program statement, by a test case. To do it so, Proteum/FL collects execution traces from the original program and executes only those mutants that are reached by each test case.

Parallel Execution: Proteum/FL takes advantage of the parallel and distributed capabilities of the modern computers and thus, it performs a parallel execution of mutants. Since each test is independent of the remaining ones, Proteum/FL assigns one test execution task per utilized process.

Relevant Mutants: Generally, mutants not killed by failed tests do not contribute to the fault localization process and hence they can be ignored. Proteum/FM mutant execution is divided into two stages: the failed tests execution and passed tests execution. It first executes the failed tests and then executes the passed tests by considering the mutants killed by the failed test cases.

The first and second optimizations are usual optimizations implemented in several mutation testing tools. However, none of the existing mutation testing tools targets fault localization which is the main aim of Proteum/FL. The last optimization is dedicated to the fault localization process and thus, there is no other tool doing something similar.

V. Related Work

There is a relatively large number of approaches and tools regarding fault localization. However, there is no other tool to the authors' knowledge that uses mutation analysis to support fault localization.

Mutation analysis is an active research field since 1970s with an increased popularity over the last years [7]. Despite this, only a relatively small number of mutation testing tools exists. Here, a brief description of the most representative tools is given by highlyting their unique characteristics.

As described before, Proteum/IM is one of the first and most popular tools for C programs. It has the main advantage of implementing all the proposed unit [9] and integration [8] level mutation operators. Recently, another tool named Milu [18] has been proposed for C programs. Milu implements a different approach to perform mutation testing based on the notion of higher order mutants. First order mutants are those produced based on one simple syntactic change. Higher order mutants are the combination of one or more first order mutants [7]. Mutation testing tools also exist for the Java programming language. MuJava [17] is one of the most popular tools. Currently it is the only tool that implements specially designed mutation operators for the Java Object Oriented features.

Regarding fault localization, one of the first attempts is Tarantula [2]. Tarantula uses statement coverage informati on to prioritize program statements according to their suspiciousness. Tarantula approach was later extended in order to include various similarity formulas like the Ochiai [14] and [15]. Later, this approach was generalized in order to include other program constructs like program branches [19], definition-use pairs [19] and combinations of them [19]. Other similar tools like Zoltar use Bayesian reasoning [20] for prioritizing suspicious program statements.

VI. Conclusion

The present paper introduced a fault localization tool named Proteum/FL. The innovative part of this tool is the use of mutation analysis for fault localization. The tool supports a comprehensive set of mutant operators specially designed for the C programming language. Additionally, it implements a wide set of optimizations techniques for reducing the mutants' execution cost.

The current version of the tool aims at solving a significant and challenging problem of software debugging. It enables a new direction of research, the use of mutation analysis for supporting software debugging activities like fault localization. We believe that Proteum/FL also opens the way for unifying the testing and debugging activities in a complementary way. In view of this, mutation analysis can be utilized for driving both the testing and debugging processes. This practice merges the application cost of testing and fault localization and results on reducing significantly the debugging effort [5].

Future releases of the tool will integrate recently developed test generation techniques like dynamic symbolic execution [11], [12] search based testing [12] and path based testing [13] towards assisting fault localization [21]. Additionally, the incorporation of the tool with an integrated development environment is also planed.

To learn more about Proteum/FL, refer to:

https://sites.google.com/site/mikepapadakis/proteum-fl

REFERENCES

[1] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459 – 494, 1985.

[2] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05)*, 2005, pp. 273–282.

[3] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the accuracy of fault localization techniques," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, 2009, pp. 76–87.

[4] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.

[5] M. Papadakis and Y. Le Traon, "Using mutants to locate "unknown" faults," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*, 2012, pp. 691–700.

[6] M. E. Delamaro and J. C. Maldonado, "Mutation testing for the new century." Kluwer Academic Publishers, 2001, ch. Proteum/IM 2.0: An Integrated Mutation Testing Environment, pp. 91–101.

[7] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[8] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 3, pp. 228–247, Mar. 2001.

[9] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the c programming language," Purdue University, West Lafayette, Indiana, techreport SERC-TR-41-P, March 1989.

[10] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, ser. APSEC '10, 2010, pp. 300–309.

[11] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10)*, 2010, pp. 121–130.

[12] ——, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Software Quality Journal*, vol. 19, pp. 691–723, 2011.

[13] ——, "Mutation based test case generation via a path selection strategy," *Inf. Softw. Technol.*, vol. 54, no. 9, pp. 915–932, Sep. 2012.

[14] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION '07)*, 2007, pp. 89–98.

[15] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011.

[16] W. Masri and R. A. Assi, "Cleansing test suites from coincidental correctness to enhance fault-localization," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST '10)*, 2010, pp. 165–174.

[17] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, Jun. 2005.

[18] Y. Jia and M. Harman, "Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language," in *Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques (TAIC-PART '08)*, 2008, pp. 94–98.

[19] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, 2009, pp. 56–66.

[20] T. Janssen, R. Abreu, and A. J. van Gemund, "Zoltar: a spectrum-based fault localization tool," in *Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime (SINTER '09)*, 2009, pp. 23–30.

[21] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th international conference on Software engineering (ICSE '06)*, 2006, pp. 82–91.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

GeCoS: A Framework for Prototyping Custom Hardware Design Flows

Antoine Floc'h*, Tomofumi Yuki†, Ali El-Moussawi‡, Antoine Morvan‡, Kevin Martin§, Maxime Naullet†, Mythri Alle‡, Ludovic L'Hours, Nicolas Simon‡, Steven Derrien‡, François Charot†, Christophe Wolinski‡ and Olivier Sentieys†

* Tocea
antoine.floch@tocea.com

† Inria - Rennes
{firstname.lastname}@inria.fr

‡ Irisa (Université de Rennes 1)
{firstname.lastname}@irisa.fr

§ Université de Bretagne Sud
kevin.martin@univ-ubs.fr

***Abstract*—GeCoS is an open source framework that provide a highly productive environment for hardware design. GeCoS primarily targets custom hardware design using High Level Synthesis, distinguishing itself from classical compiler infrastructures.**

Compiling for custom hardware makes use of domain specific semantics that are not considered by general purpose compilers. Finding the right balance between various performance criteria, such as area, speed, and accuracy, is the goal, contrary to the typical goal in high performance context to maximize speed.

The GeCoS infrastructure facilitates the prototyping of hardware design flows, going beyond compiler analyses and transformations. Hardware designers must interact with the compiler for design space exploration, and it is important to be able to give instant feedback to the users.

I. Introduction

In a typical flow of compiler research, new compiler passes are implemented as a "proof-of-concept" and to enable experimental validation. Compiler infrastructures that allow for rapid prototyping of compiler passes can greatly accelerate compiler research and are therefore an important aspect of compiler research.

Several compiler infrastructures have already been developed for this purpose, including PIPS [1], SUIF [2], ROSE [3], Polyglot [4], and CETUS [5]. These compiler frameworks provide the groundwork for implementing custom analyses and transformations with a primary focus on source-to-source transformations. They mainly target high performance computing platforms (multi-core, GPGPUs, distributed memory clusters, etc.) and focus on C/C++ based programs. In such tools, the input program is transformed into a semantically equivalent version, supposedly with better performance.

However, compiler research is not limited to general purpose or high performance computing. Designing and programming ultra low power high performance embedded hardware platforms also requires state-of-the-art compiler technology. As such platforms can be found virtually everywhere in daily life, e.g., cars, Smartphones, TVs, gaming devices, and so on, there is a growing need for compilers tailored to embedded hardware requirements.

978-1-4673-5739-5/13/$31.00

One example of such domain specific compilers are High Level Synthesis tools, also known as C to hardware compilers [6], [7]. Such tools enable the derivation of application specific custom hardware from high-level algorithmic specifications in C or Matlab. Many typical application domains benefit from HLS tools, as most embedded hardware platforms integrate custom hardware accelerators for compute intensive tasks.

The input to HLS tools has significant impact on the synthesized hardware, just like how source level transformations influence general purpose compiler outputs. However, many of these influences are specific to HLS as it depends on how well the input maps to hardware. There is hence a recent trend towards exploring architectural variants through the use of HLS specific source-level transformations [8], [9], [10], [11].

General purpose and custom hardware compilers are used in two very different contexts. In a typical compiler, users expect the output to be optimized for speed. Hardware designers map performance critical parts of embedded systems, described in C/C++, on custom hardware accelerators, where the design process consists in exploring the trade-offs between Quality of Results (QoR) and cost (area), while fulfilling some performance constraint.

A typical example is the choice of fixed-point over floating-point arithmetic, which alters the numerical accuracy of the implemented accelerator, but helps reduce its cost and/or improve its performance. The designer must choose a balance between accuracy and cost, but the desired balance is highly context sensitive.

Assisting the design space exploration process requires more than a compiler. Indeed the designers expect the tool to automate the transformation of the code, but also to provide instant feedback when trying to apply transformations. Designing such a Computer Aided Design (CAD) tool targeting embedded systems leads to two specific needs:

- The intermediate representation must be flexible and

extensible enough to take advantage of *domain specific semantics*, such as bit accurate or fixed point datatypes.
- The tool must support interactions with the programmer, guiding design space exploration. Thus, it is essential for the users to be able to drive the compilation flow, and to receive feedback from the tool.

Standard compiler infrastructures are not necessarily well suited to address such problems. Their Intermediate Representations (IR) often lack the flexibility or extensibility required. For instance, we are not aware of an existing compiler IR with native support for bit-accurate or fixed-point datatypes. Moreover, existing compiler infrastructures lack the interactivity desired in a hardware design flow.

To address these issues, we designed a compiler infrastructure named GeCoS (Generic Compiler Suite) targeting custom hardware synthesis and embedded hardware in general. GeCoS aims to facilitate fast prototyping of complex compiler optimizations and analyses, where these implemented passes can be used interactively by hardware designers to explore trade-offs between hardware performance criteria, such as speed, area, and accuracy.

The GeCoS compiler focuses on *extensibility* and *modularity*, along with powerful IR Query facilities, including an advanced pattern matching engine. These features are brought together by using Model Driven Engineering techniques and by the Eclipse Modeling Framework [12]. Interactions with users are supported through a script-driven interface to customize compilation flows, and through immediate and detailed user feedback in the source editor.

The rest of this paper is organized as follows. Section II describes the architecture of GeCoS and highlights its key features. Several tools for manipulating the intermediate representations of GeCoS are presented in Section III. Section IV illustrates how the IR of GeCoS can be extend along with its front-end and back-end. Section V describes the interactive aspect of GeCoS that allows for user-compiler interactions. Related work is discussed in Section VI and we conclude in Section VII.

II. An Overview of the GeCoS Infrastructure

In this section, we present an overview of the GeCoS infrastructure. GeCoS is not simply a compiler, but rather an environment for developing custom hardware design flows. We achieve tight integration of the compiler into the development flow by taking advantage of the Eclipse IDE.

Figure 1 presents a high level overview of GeCoS and its related toolings. The overall flow is similar to a typical compiler; the input is parsed through the front-end, processed via several analyses and transformations, and then output is generated from the modified IR. In addition to various benefits from integrating GeCoS into the Eclipse IDE, its unique features are a script-driven compilation flow and compiler IRs modeled using the Eclipse Modeling Framework. Although it is not shown in Figure 1, Static Code Analysis Framework (Codan) plugins in Eclipse enable us to provide instant feedback to the editor.

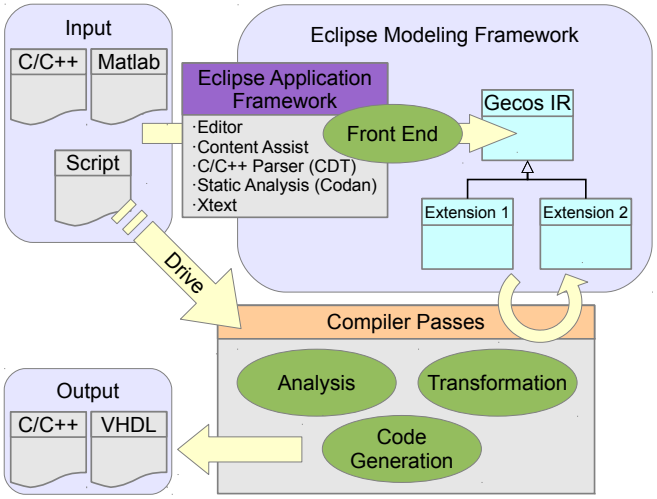


Fig. 1: GeCoS Overview. Many tools and features from the Eclipse Application Framework are used to provide an IDE-like environment for using GeCoS. The GeCoS IR and its extensions are modeled using the Eclipse Modeling Framework. EMF is also used for many components in Eclipse, making the integration easier and tighter. The implemented compiler passes are driven by user-defined scripts to enable detailed customization of the compilation flow.

A. Eclipse Integration

We made the choice to integrate our infrastructure into Eclipse to take advantage of its application framework. Hardware design flow benefits from an interactive and graphical user interface, and it was not an option to develop a full development environment from scratch.

Eclipse has many extension points, or features (e.g., editors), that can be customized through plug-in extensions. We use these extension points to implement custom editors, to take advantage of content assist, provide instant feedback, and so on. The interactive features of GeCoS based on these tools are described in Section V.

The GeCoS front-end supports a superset of C99 including bit-accurate datatypes as defined by the SystemC standard (which are indeed C++ templates). We use the C/C++ Development Tooling (CDT) of Eclipse to parse C/C++ inputs. The CDT parse tree is then converted to the GeCoS IR.

B. Compiler IR as Metamodels

One of our goals during the design of GeCoS was to reduce development and maintenance effort as much as possible. Since GeCoS was expected to be developed by generations of students, being able to reuse prior work was strongly desired. Furthermore, not all developers of GeCoS were expected to have a good background in software engineering, as our research spans between computer science and electronic engineering.

Motivated by these goals, we adopted Model Driven Engineering techniques using the Eclipse Modeling Framework. MDE is a software engineering methodology where the development effort revolves around *models*; abstractions of domain

knowledge. MDE is commonly accompanied by generative approaches, e.g., automatic generation of skeleton codes from models. Generative approaches contribute to standardized code structure, making it easier for the developers to use each others code. Developers with little software engineering background are also forced to follow certain basic principles, such as factory/visitor design patterns.

By using models to specify the *grammar* of compiler IRs, which is called *metamodeling* in modeling terms, we gain access to many benefits from MDE, directly applicable to compiler development [13]:

- Providing structural consistency: The structure of the IR is defined in the metamodel, or the grammar of the IR. Thus, tools to check if the data structure of a model instance (IR of a input source) matches its metamodel (grammar of IR) can be directly used for validating IR transformations. Such structural validation can catch bugs in transformations that deeply impact the IR.
- Access to modeling tools: There are many tools for analyzing and manipulating models. These tools include parser generators and code generators to/from models that are useful for compiler construction. Other tools include basic tree-based editors of IRs, serialization of IRs to XMI, and so on.

III. Querying and Manipulating GeCoS IR

Analyzing and manipulating Intermediate Representations is an important part of any compiler analysis or transformation. In this section we describe a set of tools that we have implemented for manipulating GeCoS IRs.

The base IR of GeCoS is an enriched Abstract Syntax Tree (AST) that represents C programs constructs. There are many forms and auxiliary information associated with the IR, such as control/data flow graphs, static single assignment (SSA) form, and DAG representation of operations within basic blocks.

Many analysis and transformation passes require IR traversal and querying operations to identify patterns ranging in complexity. With the help of MDE, GeCoS provides several tools for reducing the effort required to query its IRs. In this section, we describe two of the tools we have in GeCoS, (i) Tom/Gom term re-writing system, and (ii) graph adapter as a common interface to graph operations.

A. Tree pattern matching and term rewriting with Tom/Gom

While many languages natively support pattern matching (e.g., Scala, Caml, Haskell), the Tom/Gom tool [14] provides much more powerful constructs along with a tight integration into the Java language.

These rules are much easier to express and to understand than visitor based implementations. However, Tom/Gom requires bindings from expressions in its language to Java objects. We have developed a tool where these mappings can be specified in a Domain Specific Language (DSL). From a DSL specification of how elements in a metamodel (IR grammar) map to Tom/Gom terms, the necessary bindings are automatically generated. Our approach is very similar to that of Bach et al. [15] except that it offers more flexibility and customization opportunities. Figure 2 shows a small Tom/Gom use case.

```
%match(expr) -> {
  // factorize a.c+b.c into (a+b).c
  affine(x*,intTerm(a,c),y*,intTerm(b,c),z*) -> {
    return `affine(x*,intTerm(a+b,c),y*,z*);
  }
  // remove 0 coeficient terms
  affine(x*,intTerm(0,c),y*) -> {
    return `affine(x*,y*);
  }
}
```

Fig. 2: Examples of expression simplification rules defined using Tom/Gom. Tom/Gom terms, such as `affine` and `intTerm`, correspond to Java class instances, where the mapping is provided in a separate file. Using these rules, users can define complex re-writing rules. The first rule defines a factorization of affine expressions of the form $ac + bc$ to $(a + b)c$. The second rule defines a simplification rule that removes integer terms that evaluate to 0.

B. Graph Adapter

Graphs are key data structures for compiler analysis, used to capture many different kinds of information. Examples of graphs used in compilers include call graphs, control flow graphs, dataflow dependence graphs, program dependence graphs, and so on.

While the abstract notion of graphs is common across the board, its concrete implementations can be based on different data structures. For example, a graph edge maybe be explicitly represented by an object instance (this is the case for dependence graphs, where edges can be of different types) or simply by a list of vertex-to-vertex cross references. In addition, many graph algorithms can benefit from external optimized library implementations. While algorithmic reuse is highly desirable in such a context, it raises many difficulties.

We use the adapter design pattern, illustrated in Figure 3, to alleviate reuse of graph algorithms. However, when using adapter design patterns for many different graphs, defining the adapter also requires considerable effort.

To address this, we have implemented a toolset to automatically generate adapter patterns enabling the reuse of a large set of graph analysis and traversal algorithms (from breadth-first search to subgraph isomorphism) over many different graph representations. Our approach leverages the structural information provided by the compiler IR metamodel to derive the adapter code. We do so by using an abstract specification (specified in a textual DSL) of a set of IR Classes defining a graph. Our approach builds on the idea of Model Typing [16] developed by the Model Driven Software Engineering community.

IV. Extensible IR

In this section, we illustrate the extensibility of the GeCoS IR through two examples. The first example demonstrates domain specific code generation by using domain specific

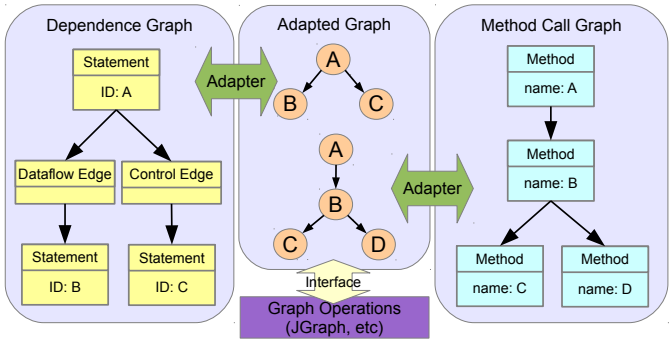


Fig. 3: Illustration of the Graph Adapter. Any graph structure can be first *adapted* to a common abstract representation of graphs. The adapted graph can now make use of available graph operations, including an interface to the JGraph library. All the efforts in analyzing and manipulating graphs can be concentrated on the adapted graph, and can be reused by many different graphs found in a compiler.

representations for regions of the program. We use polyhedral code generation as an example to extend the IR with domain specific information and to take advantage of the IR for code generation.

The second example presents an extension to the front-end to handle a subset of Matlab programs. The base IR of GeCoS can represent most but not all features of Matlab. We hence extended the IR to support new types and operations (e.g, aggregate computations operating on non-scalar data structures).

A. Extensibility by Adapters

The common mechanism used resembles the adapter software design pattern. The GeCoS base IR has several abstract nodes, such as `Block`, `Instruction`, and `Type`. The behavior of analyses and transformations may be different for each instance of these abstract classes. When an analysis or a transformation traverses the AST, it may delegate the handling of specific instances to its handler. In GeCoS delegation usually occurs when the object instance class is not known to the transformation/analysis (i.e. when the class is not part of the base GeCoS IR). Similar design is used in Polyglot [4] for providing a framework to implement Java dialects.

We use an adapter-like mechanism combined with the plug-in framework of Eclipse to achieve a modular and extensible implementation of analyses and transformations. The key in the modularity is that an analysis or transformation does not need to be aware of all possible implementations of an abstract node. Corresponding handlers will be loaded as necessary by Eclipse at run-time, separating domain specific behaviors into its own modules (Eclipse plug-ins).

B. Polyhedral Code Generation

The polyhedral model [17] is a formalism enabling the analysis and transformation of complex loop structures. The model is restricted to a class of loop with affine bounds and array indexing functions, known as Static Control Parts (SCoPs). For this class of loops, the model provides a unified framework for capturing, checking and applying complex combinations of loop and array layout transformations. Polyhedral techniques have proven to be very effective for automatic parallelization [18] on multi-cores, GPGPUs and computing clusters, and are also very relevant for embedded hardware design.

To enable polyhedral transformations within GeCoS, we have extended the IR to represent polyhedral regions (SCoPs) in a program. This was done by introducing an inheritor of `Block`, namely `SCoPBlock`, to model SCoPs. This block models loop nests in polyhedral representation, and goes through different sets of analyses and transformations. Instances of `SCoPBlock` are extracted from the initial program using a combination of complex pattern matching rules and normalization transformations both heavily relying on the Tom/Gom framework (the rewriting rules shown in Figure 2 are one such example).

Although it is possible to convert SCoPs blocks back to the base IR so as to benefit from the existing code generator, doing so prevents us from leveraging the domain specific semantics of SCoP blocks during code generation. For example, it is possible to generate very efficient HDL code from most SCoP based representation, whereas this becomes much more challenging to do from a standard control-flow graph representation. To address this issue, we rely on the extensibility of GeCoS to provide custom code generator extensions which handle code generation stages for all instances of `SCoPBlock`.

C. Matlab Front-end

A more involved extension example is the extension of the IR to support a new language front-end, in our case a subset of Scilab/Matlab. Thanks to the use of an EMF-based IR, we were able to benefit from advanced textual DSL design frameworks such as Xtext [19]. This allowed us to build an IDE and parser front-end for the Matlab language from a single specification, where the parsing result is directly in the form of an extended GeCoS IR.

However, the real challenge stems from the fact that, in addition to being dynamically typed, these languages offer built-in vector and matrix types along with many corresponding operators. Deriving custom hardware from such programs first requires that the compiler is able to infer (at least semi-automatically) the type of each program statement statically. Then arrays and vector operations need to be lowered into simpler loop based constructs that can be supported by a high-level-synthesis back-end.

Extending the IR for such languages hence involves not only adding new extensions to the `Block` and `Instruction` classes, but also extending the type system by introducing new `Type` classes to model vector/matrix types and partial/incomplete typing information which is to be resolved during type inference.

Similarly, we also designed the type inference engine by reusing and extending the initial GeCoS type propagation pass,

by delegating the handling of extended types to a Matlab specific type propagation pass.

V. Interactive Development in GeCoS

In this section, we illustrate the interactive aspect of GeCoS. The two key features for user interaction are (i) script-driven compilation flow, and (ii) static analysis feedback while programming.

A. Script-Driven Compilation Flow

Users of GeCoS "compile" a program by providing a script specifying the sequence of compiler passes to invoke. Figure 4 illustrates an example of a script that runs a sequence of passes in GeCoS. Although the default options are used in the figure, many of the commands in the script have multiple optional parameters to fine tune its corresponding compiler pass.

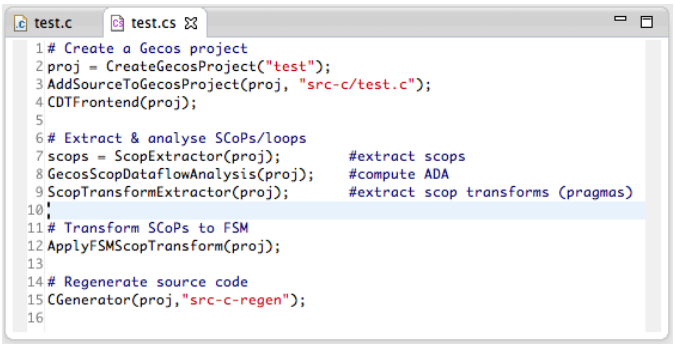


Fig. 4: Example GeCoS Script. First block of commands creates a `GecosProject`, which defines a compilation unit, specifies a source and parses the input using CDT. The second set of instructions performs the analysis and transformations desired by the user. In this case, Static Control Parts of the program are detected and then converted to a Finite State Machine for improved efficiency in hardware. Finally, the code generator is called to reproduce C code.

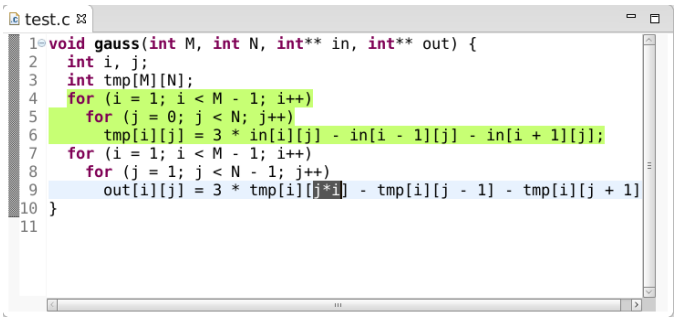
Using the Eclipse framework brings additional features that are crucial to increase productivity of both users and developers of compiler passes. For example, we have taken advantage of the content assist module of Eclipse so that users can access the available list of commands, just like searching for Java methods in Eclipse. As another example, users may use the "Open Definition" feature of Eclipse to locate the implementation of a command in the script.

B. Immediate Feedback to Users

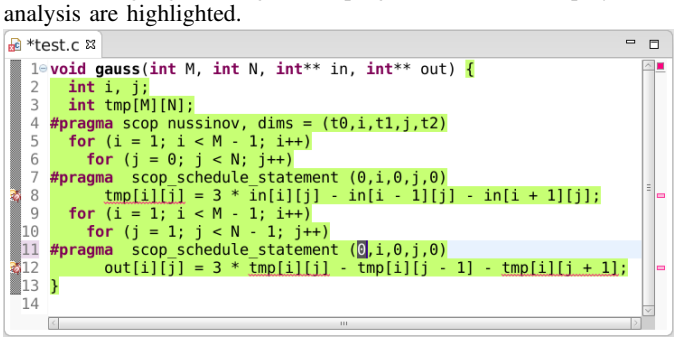
Those who have used Eclipse or any other IDE for programming have experienced how quickly an IDE can flag simple syntactic errors to the user. These tools constantly check for compilation errors/warnings and reflect them in the editor. Even purely syntactic checking can help improve productivity by avoiding going back and forth between the code and compiler outputs.

Optimizing compilers perform a significant amount of analysis to select legal transformations. For instance, automatic parallelization requires sophisticated dependence analysis to find sets of parallel operations. In the interactive compilation flow, it is desirable to reflect such analysis results to the user, in a manner similar to syntactic checks.

The Eclipse Static Code Analysis Framework (Codan) provides an API for providing such feedback. Figure 5 illustrates examples of how we use the Codan framework to give instant feedback to the user.



(a) SCoP highlighter. Regions of programs amenable to polyhedral analysis are highlighted.



(b) Schedule verification. For polyhedral regions of a program, the user may specify *schedules*, which is an encoding of loop transformations. In this example, the schedules specified by the user attempt to merge the two loops. However, the transformation is illegal and dependencies involving underlined array accesses are violated.

Fig. 5: Examples of feedback given by GeCoS through the Codan framework. These analyses can be performed automatically when a user saves the file, or invoked upon user request.

Although the Codan framework only supports C/C++, it is possible to implement such features for other languages with relatively little effort by taking advantage of the highly customizable editors in Eclipse.

VI. Related Work

In this section, we discuss compiler infrastructures in relation to GeCoS. There are many compiler infrastructures with different goals, but we are not aware of other frameworks that target prototyping of hardware design flows.

There are several compiler infrastructures already available, such as PIPS [1], SUIF [2], ROSE [3], and CETUS [5]. These infrastructures aim at offering data structure and APIs that can scale to millions of lines of code, often at the expense of ease of use. To the contrary, hardware synthesis focuses on computational hotspots that rarely consist of more than a few thousands of lines of code, yet it commonly takes minutes if not hours to synthesize.

Moreover, they are intended to be robust compilers, primarily focusing on general purpose processors. Thus, they have little or no support for embedded platforms, or user interaction.

CoSy [20] is a compiler framework for slightly different goals. They have developed a production quality compiler for embedded processors with a high degree of flexibility. They propose to utilize the flexibility to create customized versions tailored for specific architectures. However, they target embedded processors, and not custom hardware.

Polyglot [4] is a compiler framework that focuses on supporting domain specific dialects of Java. They also provide an adapter-like framework to allow users to develop extensions with little effort. HLS tools typically take C/C++ as inputs and not Java, and thus Polyglot is not suitable for compilers targeting hardware design.

CHiLL [21], POET [22], and AlphaZ [23] are program transformation frameworks that allow users to specify transformation sequences via script. However, all of these frameworks target high performance computing (multi-cores, and GPGPUs), and lack support for hardware-oriented compilation. Specifically, these tools primarily work on abstracted representations of the loop nest, and focus on loop transformations. It would require significant changes to their internal data structures to support fixed-point arithmetics and other transformations involving operations within a statement.

VII. Conclusion

We have presented GeCoS, a compiler infrastructure targeting custom hardware design. Compiler research for hardware design has very different goals and requirements, making existing infrastructures difficult to use. In particular, the need to model domain specific semantics, and to support interactions with the user are much stronger in our context.

The tool is open source and a ready to use Eclipse installation is also available on our website[1]. Readers can also watch a 5 minute YouTube video[2] to see GeCoS in action.

Our paper may also be viewed as an advertisement of the Eclipse framework and modeling techniques for compiler development. GeCoS benefits a lot from tools and frameworks coming from Eclipse. Taking advantage of Eclipse can provide an ideal environment for implementing domain specific languages, which even comes with an IDE, with little effort.

Acknowledgements

This work was funded in part by the INRIA STMicroelectronics Nano2012-S2S4HLS project, the French ANR Compa project (ANR-11-INSE-0012), and the European Commission Seventh Framework Programme (FP7/2007-2013) with Grant Agreement 287733.

References

[1] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization: An overview of the pips project," in *Proceedings of the 5th international conference on Supercomputing*, 1991, pp. 244–251.

[2] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam *et al.*, "Suif: An infrastructure for research on parallelizing and optimizing compilers," *ACM Sigplan Notices*, vol. 29, no. 12, pp. 31–37, 1994.

[3] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[4] N. Nystrom, M. R. Clarkson, and A. C. Myers, "Polyglot: An extensible compiler framework for java," in *Proceedings of the 12th International Conference on Compiler Construction*, 2003, pp. 138–152.

[5] S.-I. Lee, T. A. Johnson, and R. Eigenmann, "Cetus–an extensible compiler infrastructure for source-to-source transformation," in *Languages and Compilers for Parallel Computing*, 2004, pp. 539–553.

[6] T. Bollaert, "Leveraging the efficiency of c-based design with catapult-c," in *Forum on specification and Design Languages, FDL 2005, September 27-30, 2005, Lausanne, Switzerland, Proceedings*, 2005, pp. 419–429.

[7] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011, pp. 33–36.

[8] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 51:1–51:10.

[9] A. Morvan, S. Derrien, and P. Quinton, "Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 3, pp. 339–352, 2013.

[10] C. Alias, A. Darte, and A. Plesco, "Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013, pp. 575–580.

[11] J. a. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "Lara: An aspect-oriented programming language for embedded systems," in *Proceedings of the 11th International Conference on Aspect-oriented Software Development*, 2012, pp. 179–190.

[12] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*, 2008.

[13] A. Floch, T. Yuki, C. Guy, S. Derrien, B. Combemale, S. Rajopadhye, and R. France, "Model-driven engineering and optimizing compilers: A bridge too far?" in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, 2011.

[14] M. Pierre-Etienne, C. Ringeissen, and M. Vittek, "A pattern matching compiler for multiple target languages," in *Compiler Construction*, G. Hedin, Ed., vol. 2622, 2003, pp. 61–76.

[15] J.-C. Bach, X. Crégut, P.-E. Moreau, and M. Pantel, "Model transformations with Tom," in *Proceedings of the 12th Workshop on Language Descriptions, Tools, and Applications*, 2012, pp. 4:1–4:9.

[16] J. Steel and J.-M. Jézéquel, "On model typing," *Software & Systems Modeling*, vol. 6, no. 4, pp. 401–413, 2007.

[17] P. Feautrier and C. Lengauer, "The polyhedral model," in *Encyclopedia of Parallel Programming*, D. Padua, Ed., 2011.

[18] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 101–113.

[19] Itemis, "Xtext," online : http://xtext.itemis.com/.

[20] ACE Associated Compiler Experts, "The cosy compiler development system," 2007.

[21] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," *University of Southern California, Tech. Rep*, pp. 08–897, 2008.

[22] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "Poet: Parameterized optimizations for empirical tuning," in *Proceeedings of the 2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[23] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "Alphaz: A system for design space exploration in the polyhedral model," in *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing*, 2012.

[1] `http://gecos.gforge.inria.fr/`

[2] `http://www.youtube.com/watch?v=89Sr1NHaqNs`

Review Efforts Reduction by Partitioning of Static Analysis Warnings

Tukaram B Muske
TRDDC, 54-B,
Hadapsar Industrial Estate,
Pune, MH, 411013, India.
Email: t.muske@tcs.com

Ankit Baid
TRDDC, 54-B,
Hadapsar Industrial Estate,
Pune, MH, 411013, India.
Email: ankit.baid@tcs.com

Tushar Sanas
TRDDC, 54-B,
Hadapsar Industrial Estate,
Pune, MH, 411013, India.
Email: tushar.sanas@tcs.com

***Abstract*—Static analysis has been successfully employed in software verification, however the number of generated warnings and cost incurred in their manual review is a major concern. In this paper we present a novel idea to reduce manual review efforts by identifying redundancy in this review process. We propose two partitioning techniques to identify redundant warnings - 1) partitioning of the warnings with each partition having one leader warning such that if the leader is a false positive, so are all the warnings in its partition which need not be reviewed and 2) further partitioning the leader warnings based on similarity of the modification points of variables referred to in their expressions. The second technique makes the review process faster by identifying further redundancies and it also makes the reviewing of a warning easier due to the associated information of modification points. Empirical results obtained with these grouping techniques indicate that, on an average, 60% of warnings are redundant in the review context and skipping their review would lead to a reduction of 50-60% in manual review efforts.**

***Keywords*—Static Analysis, Data Flow Analysis, Analysis Warnings, Review of Warnings, False Positives.**

I. Introduction

Static analysis used in the verification of software systems has been very successful, however the number of warnings generated and cost incurred in their manual review is a major concern. Static analysis tools report a large number of warnings due to the abstractions used by them [1]. Many of the reported warnings do not represent run-time error, hence such falsely identified warnings are referred to as false positives. Manual review is required to partition the reported warnings into false positives and true errors. This review process is usually tedious and time consuming.

Currently a lot of research in static analysis is directed towards making the analysis more precise using techniques such as abstract interpretation ([2], [3]), model checking [4] and their combinations ([5], [6]). Although, not much has been done to reduce efforts involved in reviewing static analysis warnings. We focus our research in the direction of minimizing user's review efforts by asking the following questions.

- Does the user need to review each reported warning?
- Can we group the warnings such that only one warning from each group needs to be reviewed?
- Can it be useful and, if so, how much?

978-1-4673-5739-5/13/$31.00

The motivation to ask these questions comes from our experience that quite a few of these warnings are similar and all of them need not be reviewed. We illustrate this through the simple code snippet in Figure 1 where six similar Array Index Out of Bound (AIOB) warnings are shown. All these warnings are similar because they refer to the same array and index variable, and will either be all false positives or all errors. Hence only the first AIOB warning needs to be reviewed and review of rest of the warnings can be skipped as their review judgment will be same as the review judgment of the first warning.

```
...
// i, loopBound are integers and their values are
// statically unknown
for( i = 0; i < loopBound; i++)
{
    structArr[i].m1 = structArr[i].m1 + value1;
    structArr[i].m2 = structArr[i].m2 + value2;
    structArr[i].m3 = structArr[i].m3 + value3;
}
...
```

Fig. 1. AIOB Warnings - Code Sample 1

We partition the generated warnings and tag one warning from each partition as *leader warning* (LW) and the rest as *follower warnings* (FWs) such that if the leader is a false positive so are the followers. This grouping approach allows users to avoid individual review of follower warnings if the corresponding LW is identified as a false positive. In case a LW is identified as an error, its associated FWs are also potential errors and need to be reviewed individually. In practice we have found that many LWs are false positives leading to a reduction of review efforts.

```
void f() {
  ...
  i = ...;
  if(...)
        arr[i] = ...;
  else
        arr[i] = ...;
}
```

Fig. 2. AIOB Warnings - Code Sample 2

We further propose a grouping of LWs based on similarity of the modification points of variables referred to in their expressions. The motivation behind such *modification points*

based grouping (MPG) is that multiple warnings get impacted by the same modification points, *i.e.*, many warnings have the same source of values for their expression variables. Figure 2 presents a motivating example where the array indexes get values from same modification point. These AIOB warnings can not be put under same partition as per earlier partitioning approach, since the required leader-follower relationship with respect to their review judgments can not be guaranteed.

The MPG is such that, if any warning from a group is analyzed to be a false positive referring to values provided by the associated modification points then all the warnings in the group will be false positives and need not be analyzed individually. In the other case where the selected warning is not a false positive then user need to analyze each warning from its group separately. The reported information of the modification points avoids code traversals required in locating them during their analysis and further contributes to reduced review efforts.

In this paper, we discuss above grouping techniques using C program static analysis warnings. However, they can be applied to warnings generated on applications in other programming languages as well, when the warnings include review redundancy in themselves. Empirical results by applying these proposed grouping techniques on embedded applications of a total size of over 16 MLOC indicate that, on an average, 45% of the generated warnings are FWs and this identification reduces review efforts by about 20%. MPG further reduces the review efforts by 40-50% with the identification of 20% of redundant warnings amongst the leaders, and also helps the user to review the LWs easily due to the association of the modification points.

The key contributions of this paper are - a) a method to partition warnings with each partition having exactly one leader requiring review and, b) a method to group the leaders based on similarity of the modification points of the variables they refer to.

We discuss in detail the grouping of similar warnings in Section II and MPG in Section III. The implementation details are provided in Section IV and Section V discusses the various experiments performed and observed results. Section VI presents the related work and we conclude with Section VII.

II. Grouping of Similar Warnings

This section discusses in detail the partitioning of warnings, its comparison with other tools and presents an algorithm to partition the warnings.

A. Similar Warnings: Concept

Static analysis tools check certain expressions during code analysis to prove or disprove run-time properties such as AIOB, Zero Division (ZD), Illegal Dereferencing of Pointer (IDP). These tools evaluate an expression for the values it may take and report it as safe or an error or a warning. For example, array index expressions (AIOB) are checked to see if valid array elements are being indexed, denominator expressions (ZD) are confirmed to evaluate to non-zero values, pointers being dereferenced (IDP) are ensured to be pointing to a valid memory. Throughout this paper we refer to these expressions as *expressions of interest* (EOIs).

There exists many scenarios in which static analysis tools are unable to compute values of the EOIs precisely and hence the number of the generated warnings is high. Analysis of these warnings is a major concern due to the efforts involved. Hence, there is a need to reduce the number of warnings that need to be reviewed. We reduce the number of warnings requiring review by partitioning them based on their similarity. Two warnings are similar only if -

1) their EOIs are structurally similar *and*
2) the variables referred to by these EOIs have the same source statements for their values.

The structural similarity of expressions requires that variables used, operators and their order of appearance be the same. For example, given two ZD warnings with their denominator expressions as

- *a+b-c* and *a+b-c* are potentially similar.
- *v+1* and *1+v* will not be similar even if they evaluate to the same values because the operands differ in their positions. This can be viewed as a limitation of our approach to identify similar warnings however, such scenarios have been found to be rare in practice.
- *v1+func()* and *v1+func()* will not be similar since different calls to a function can return different values. A call to any function is treated as unique and will not be similar to any other call to a different or same function.

The warnings which are found to be similar by the above approach are grouped together with a leader selected from them. A warning will be selected as a leader (LW) only if the values resulting from the evaluation of its EOI is a superset of values evaluated by the EOI of other grouped warnings. This ensures that the other grouped warnings are regarded as false positives when the leader is a false positive and, are potential errors when the leader is an error. That is, these other grouped warnings follow review judgment of leader, hence, are referred as follower warnings (FWs).

Consider the sample program in Figure 3 which includes three ZD and five AIOB warnings since values of denominator *factor* and array index *n* expressions are not known statically. We use the notation ZD_n and $AIOB_n$ to denote division and array access points from line *n* respectively. The ZD warnings are similar and can be partitioned together because their denominators (EOIs for ZD) are similar and evaluate to the same values. The ZD_6 point appears on all paths coming to ZD_9 and ZD_{12} and variable *factor* is not modified in between hence, it can be safely selected as the LW of its group. In fact any ZD warning can be selected as the leader and in such cases we select the first warning as leader.

In general, a complete array expression (array and its index) is considered as an EOI for AIOB. The array is required to get the bound (array size) for its index expression. In our analysis, we ignore the array being indexed but consider its size as a part of an EOI. The size of an array is known statically unless it is an *extern* array. Hence, given two AIOB warnings will

```
#define SIZE 10
int rColors[SIZE], gColors[SIZE], bColors[SIZE];

1. void func(int r, int g, int b)
2. {
3.    // Values of 'n' and 'factor' are not known
4.    ...
5.    factor  = getDivFactor();
6.    if((r/factor > rval1) && ...)
7.       rColors[n] = r;
8.
9.    if((g/factor > gval1) && ...)
10.       gColors[n] = g;
11.
12.   if((b/factor > bval1) && ...)
13.       bColors[n] = b;
14.
15.    gradient = getGradient(rColors[n],
16.                           gColors[n], b);
17.    ...
18. }
```

Fig. 3. ZD & AIOB Warnings - Code Sample 3

be similar only if their index expressions have same structure, these indices evaluate to same values and involved arrays are of same size. We consider this change in order to partition more number of warnings together.

The AIOB warnings in Figure 3 are similar warnings and can be partitioned together, however, leader selection for their partition is not straight forward. None of the warnings from $AIOB_7$, $AIOB_{10}$ and $AIOB_{13}$ points can be tagged as leader since their index values may not contain values of indices at other AIOB points. This is because all these points are not always included on the paths coming to other program points. However, $AIOB_{15}$ can be selected as the leader of these warnings and its review is sufficient to review all of them. This is because, it appears on all the paths on which rest warnings are present.

For simplicity, we have presented the ZD EOIs (*factor*) and AIOB EOIs (*n*) in Figure 3 with single variables only. This proposed grouping of similar warnings can be applied even if the EOIs are complex expressions such as $(var1 + var2 - var3)$, $(arr1[0] + arr[i])$.

B. Grouping of Similar Warnings: Need

The way in which similar warnings are handled vary as per static analysis tools. For the example in Figure 3, Astrée [7] reports all the ZD and AIOB points as warnings whereas Polyspace [8] reports warnings for ZD_6, $AIOB_7$, $AIOB_{10}$, $AIOB_{13}$ and $AIOB_{15}$. The reporting of warning for each of the division and array access points by Astrée increases the count of warnings (3 ZD and 5 AIOB warnings) and, thus, the associated review efforts.

Polyspace reports 4 AIOB warnings where reviewing of only one warning from line 15 or 16 would have been sufficient. This is because these warnings include redundancy in themselves. Polyspace reports only one ZD warning (ZD_6) and rest two ZD points as safe, however, such reporting can further lead to multiple verification cycles when these warnings are found as errors on manual review. On finding ZD_6 as an error, a user will correct only the ZD_6 warning and will not pay attention to ZD_9 and ZD_{12} division points as they are reported safe. Reporting such potential error points as safe points can lead to multiple verification cycles.

Partitioning these similar ZD and AIOB warnings with the selected leaders, as proposed, will result in only two LWs (ZD_6 and $AIOB_{15}$) which need to be reviewed initially by the user. When these LWs are found safe, the warnings count is reduced from 8 to 2 in case of Astrée and from 5 to 2 in case of Polyspace. Unlike Polyspace, this approach does not report any potential error point as a safe point. Still, this reduces the number of warnings to be reviewed due to the selection of leaders. It further avoids multiple verification cycles when warnings are observed as errors. For example, if the leader ZD_6 is found as an error, the warnings ZD_9 and ZD_{12}, being followers of ZD_6, would be potential errors too and will get corrected together. Thus, grouping of similar warnings addresses two major issues -

1) large number of generated warnings and associated review efforts *and*
2) multiple verification cycles

C. Data Flow Analysis: Background

Before we present an algorithm to compute LWs and FWs, we provide some background for the *data flow analysis* (DFA).

1) Control Flow Graph: A control flow graph (CFG) is a directed graph for a program depicting paths in the program that might be traversed during its execution. In formal terms, a CFG is a pair $G = (N, E)$ where N is the set of nodes corresponding to basic blocks in the program and E is the set of edges representing (un)conditional transfers of control flow. A basic block is a sequence of program statements having a unique entry and exit, *i.e.*, they do not have conditional jumps coming into the block or going out of it. A CFG includes two special purpose nodes depicting the *entry block* (where the control enters the CFG) and the *exit block* (where the control exits the CFG). Often, the entry and the exit blocks represent, respectively, the start and the end of the function/application under consideration. CFGs are widely used in compiler optimizations and static analysis. In our analysis, there is a separate node in the CFG for each assignment statement and each expression that controls the flow.

2) Data Flow Analysis: Data flow analysis (DFA) computes and propagates information about variables in a program (flow information) at each program point of a function/application, commonly using a CFG. DFA is performed by associating data flow equations for each of the nodes or edges in the CFG and iteratively solving them until a fixed point is reached [9]. A data flow equation for a node *n* in *forward* (backward) analysis is the function of flow information generated at the *exit of its predecessors* (start of its successors) and the flow information locally generated at *n*.

The data flow equations for a node *n* in forward analysis are of this form -

$$Out_n = trans_n(In_n)$$

$$In_n = O_{\ p\ \epsilon\ predecessors(n)}(Out_p)$$

$$trans_n(X) = (Gen_n + X) - Kill_n(X)$$

where,

1) O represents the join operation to merge the information that flows in from multiple paths.
2) In_n is information flowing in at the start of *n*.
3) Out_n is the flow information computed at the exit of node *n* by applying the transfer function $trans_n$ on In_n
4) Gen_n represents information locally generated at *n*.
5) $Kill_n(X)$ represents the part of information X (kill information) which should not be propagated at the exit of *n*

D. LWs Identification Algorithm

As discussed earlier, a warning can be tagged as leader for a group only if possible values of its EOI contains the values of EOIs of its followers. We use *must reachability* and *must liveness* of expressions to obtain the required LWs and associate them with their corresponding FWs. We intend to compute the reaching and live expressions as *must* information where, respectively, it represents the expressions that are definitely reaching or definitely live from the program points at which they appear. These must reaching (must live) expressions can be thought of similar to reaching definitions (live variables) [9] with variables replaced by expressions and computing *must* information instead of *may*.

We compute *must reaching expressions* (MREs) and *must live expressions* (MLEs) with respect to the EOIs of the input warning points and use them in LW identification. It should be noted that the expressions under consideration (EOIs) are unique in themselves and are not similar to expressions as used in *available expressions* [9]. Here, an EOI is uniquely identified by their corresponding top level nodes in an abstract syntax tree built for the input source code.

1) Must Reaching Expression (MRE): An expression *e* at a program point P_e is a MRE at a program point P if every path coming to P passes through P_e and, no path segment between P_e and P contains a *l-value* occurrence of any of the *r-value(s)* of *e*. That is, P_e precedes P and the values evaluated by *e* at P_e contains the values of *e* when evaluated at P.

Few examples of MREs from Figure 3 are -

- The denominator of ZD_6 is MRE for ZD_9 and ZD_{12} as each path to ZD_9 and ZD_{12} is coming through ZD_6 and the variable *factor* is not modified in between.
- The denominator of ZD_9 is MRE for ZD_{12} but not for ZD_6 as ZD_9 does not precede ZD_6.
- The index *n* of $AIOB_7$ is not a MRE to other AIOB points ($AIOB_{10}$, $AIOB_{13}$, $AIOB_{15}$) since there exists a path reaching these points not passing through $AIOB_7$.

2) MREs Data Flow Formalization: Must reachability analysis for expressions requires forward information flow (that tags ZD_6 in Figure 3 as LW). We present below the data flow formalization for MREs computation at *procedure* level. It considers one run-time property at a time while computing MREs, *i.e.*, MREs for grouping of similar ZD and AIOB warnings will be computed separately. The below data flow equations are shown for a node *n* in a CFG.

$$In_n = \begin{cases} \emptyset & n \text{ is the entry block} \\ \bigcap_{p \ \epsilon \ \text{predecessors}(n)} Out_p & \text{otherwise} \end{cases}$$

$$Out_n = (Gen_n + In_n) - Kill_n(Gen_n + In_n)$$

$$Gen_n = \begin{cases} \{e\} & n \text{ has a warning with } e \text{ as its EOI} \\ \emptyset & \text{otherwise} \end{cases}$$

$$Kill_n(X) = \begin{cases} killInfo(X,\text{n}) & n \text{ modifies at least one variable} \\ \emptyset & \text{no variable is modified by } n \end{cases}$$

$$killInfo(X,n) = \{e \in X \mid (usedVars(\text{e}) \cap modifiedVars(\text{n})) \neq \emptyset\}$$

$usedVars(e) =$ *r-values* from expression e

$modifiedVars(n) =$ *l-values* from program statement n

In the above formalization,

- In_n and Out_n represent the MREs flowing in and out at *n* respectively.
- the In_n equation includes intersection because flow information being computed is a *must* information.
- the information flowing in at a point is computed using the information flowing out of its predecessors. This is because we compute the flow information in forward direction.
- the MREs are generated only at the warning points and kill information is computed at each variable modification point.
- the In_n equation indicates that MREs are computed at procedure level. This equation will need a change if the MREs are to be computed at the application level. The change is required to consider the effect of calling contexts and function call points.

In this formalization, the equation for Out_n is different from the standard DFA equation as the kill component is computed from $(Gen_n + In_n)$ instead of In_n. This change is incorporated as MREs can get generated and killed at the same nodes. For instance, we do not want the index expression *val* to be flowing out as MRE after $val = arr[val]$; program point.

3) Must Live Expression (MLE): An expression *e* at a program point P_e is a MLE at a program point P if every path coming out of P also passes through P_e and, no path segment between P and P_e contains a *l-value* occurrence of any of the *r-value(s)* of *e*. That is, P precedes P_e and the values evaluated by *e* at P_e contains the values of *e* when evaluated at P.

Few examples of MLEs from Figure 3 are -

- The index *n* of $AIOB_{15}$ is MLE at $AIOB_7$, $AIOB_{10}$ and $AIOB_{13}$ since all paths coming out of them

also pass through $AIOB_{15}$ and their index n is not modified in between.

- The index of $AIOB_{13}$ is not MLE for $AIOB_7$ and $AIOB_{10}$ since there exists a path coming out of them which does not include $AIOB_{13}$.

4) Data Flow Formalization for MLEs: The must liveness of expressions includes backward information flow (that tags $AIOB_{15}$ in Figure 3 as LW). The formalization for MLEs computation will be similar to that of MREs. The only change here is the direction of information flow which is *backward* in case of MLEs. In order to account for this change, we change the In_n and Out_n equations as under.

$$Out_n = \begin{cases} \emptyset & n \text{ is the exit block} \\ \bigcap_{s \ \epsilon \ \text{successors}(n)} In_s & \text{otherwise} \end{cases}$$

$$In_n = (Gen_n + Out_n) - Kill_n(Gen_n + Out_n)$$

5) Computation of LWs and FWs: Once MREs and MLEs are available at each program point of an application, identification of partitions with their associated LWs becomes easy. We denote the MREs and MLEs flowing out at a program point *n* as $MREs(n)$ and $MLEs(n)$ respectively. Any given two warnings W_1 and W_2, with their EOIs as e_1 and e_2 respectively, will be in the same partition only if e_1 and e_2 are structurally similar and, any of the following hold.

1) $e_1 \in (MREs(W_2) \cup MLEs(W_2))$. In this case, W_2 will be a FW and W_1 can be its leader.
2) $e_2 \in (MREs(W_1) \cup MLEs(W_1))$. In this case, W_1 will be a FW and W_2 can be its leader.

A warning W, with *e* as its EOI, can be selected as a leader of a partition if for every other warning W' in the partition, $e \in MREs(W') \cup MLEs(W')$. If more than one warning in a partition qualify to be a leader, any one of them can be randomly selected as the leader.

E. Similar Warnings Identification: Limitations

In this section we present the scenarios where the proposed approach of grouping similar warnings is unable to identify redundant warnings.

```
void f() {
  ...
  arr[i] = ...;
  g(i);
}

void g(int j) {
  var = arr[j];
}
```

Fig. 4. AIOB Warnings - Code Sample 4

Limitation 1: The requirement of structural similarity of the EOIs of the warnings while finding similar warnings misses on identifying similar warnings which are structurally different. One such example is illustrated in Figure 4 where the warning from function *g* is redundant in its review context if the warning from function *f* is found as a false positive. Our approach will not find them as similar since their index variables are different. However, one can overcome this limitation by improving the MLE and MRE formalizations to include the mapping of arguments to parameters and accordingly computing inter functional MREs and MLEs.

```
int arr[10];

void func(const int* baseAddr, const int* currPtr)
{
   int val = currPtr - baseAddr;
    switch (switchVar){
      case 1:  func1(val); break;
      case 2:  func2(val); break;
      case 3:  func3(val); break;
      default: break;
   }
}

void func1(int p1){
   if(a<b)  arr[p1] = ...;
   else  arr[p1] = ...;
}

void func2(int p2){
   if(c<d)  arr[p2] = ...;
   else  arr[p2] = ...;
}

void func3(int p3){
   if(e<f)  arr[p3] = ...;
   else  arr[p3] = ...;
}
```

Fig. 5. AIOB Warnings - Code Sample 5

Limitation 2: The usage of MREs and MLEs to identify required LWs sometimes misses on grouping the warnings which are similar but are not definitely reachable from one another. One such example is presented in Figure 5 where six AIOB warnings are shown. The two AIOB warnings from function *func1*, *func2* and *func3* are similar, however, they will not be identified as similar with the approach of MREs and MLEs. This is because these warning points can not be reached definitely from one another and there will not be MRE/MLE available at any warning point from another.

III. Modification Points based Grouping

This section describes details of the MPG approach and explains how it helps in reducing the review efforts.

A. MPG: Need

There exist many scenarios where the redundancy of warnings is not captured by grouping similar warnings and these scenarios appear frequently in practice. In these scenarios each of the reported warning is individually reviewed and, therefore, takes a lot of time. This redundancy must be identified to reduce the review efforts further.

Consider Figure 5 where the warnings are reported due to the assignment of result of pointer arithmetic to variable *val* in function *func*. During review of a warning, user needs to know the value(s) of its index expression. A code traversal is required to locate program point(s) contributing to its index values. This includes locating calls to its enclosed function, actual to formal parameters mapping and, finally, getting modification points

for the variable *val*. User needs to manually analyze the pointer arithmetic to get the values assigned to *val*. If these values range from 0 to 9 then the current warning is a false positive, else it represents a run-time error.

This review process needs to be performed at least once for the warnings from each function. The modification points of a variable referred in a warning can appear anywhere in the application and efforts involved in locating them may be significant. In practice, we have found that there exist many warnings having similar EOIs and same modification points for their referred variables (henceforth called as EOI variables). The time spent in reviewing them individually is considerable and such individual reviewing is often redundant. We avoid this by grouping the warnings based on the similarity of the modification points of the EOI variables.

B. Identification of MPG groups

We group the warnings which have the same modification points for their EOI variables and associate these modification points to each of the groups formed. We group any two given warnings only if -

1) their EOIs have the same operators appearing in the same order *and*
2) the EOI variables related by their position in the EOIs have same modification points.

It should be noted that the equality of EOIs ignores variables referred in them so that more warnings get grouped together. The associated information of modification points further reduces the review effort for warnings by allowing user to locate the modification points without performing code traversals.

As part of the review process, user picks any of the MPG grouped warnings and reviews it considering its EOI variable values from reported modification points. If the user finds this warning as a false positive then other warnings from the same group are guaranteed to be false positives. This is because the EOI values at any warning point will always be a subset of its values when it is evaluated with the values of EOI variables at their modification points. In the other case, when the warning is not found to be a false positive, each warning in the group must be reviewed individually.

The MPG approach groups both the warnings from Figure 4 and all six warnings from Figure 5. This leads to a reduction in the review efforts as none of the remaining warnings in a group will be reviewed if the first one turns out to be a false positive.

IV. Implementation

We implemented our proposed grouping techniques in TCS Embedded Code Analyzer (TECA) [10]. TECA is a TCS-internal static analysis tool used to verify C source code. This section covers the implementation details of the proposed techniques.

A. Implementation for MREs/MLEs

We implemented the data flow analysis for MREs/MLEs using the *function summary based* approach described in [9] and the data flow equations presented earlier (in II-D). This information is computed, once for each property, at the application level in a context and flow sensitive manner.

The warnings having any of its EOI variables as *volatile* or *shared* (as in multi-tasking applications) were ignored from MREs/MLEs computation. This is because these EOI variables may get modified outside the context of the application or the task being analyzed. Grouping of such warning with another will not guarantee the correctness of the grouping approach. We also avoided computing the MREs and MLEs with respect to the warnings that had function call or unary pre/post increment/decrement operator in their EOIs. The count of such warnings is considerable and they do not contribute in FWs identification and, hence, were avoided in first place to improve on efficiency of MREs/MLEs computation.

B. Implementation for LWs and FWs

We use a graph based approach to compute LWs and their associated FWs using computed MREs and MLEs. Initially, in this approach, each generated warning is represented as a tree having only one node. In the forest so formed, the number of trees is equal to the number of warnings that are being partitioned. For each MRE/MLE e available at a warning W, we add a directed edge from a node representing W'(say) to the node representing W if -

1) e is an EOI of W',
2) EOI of W and e are structurally similar *and*
3) the node corresponding to W' does not have an incoming edge.

The last constraint is to avoid adding redundant edges since W' having an incoming edge must have already been seen as a FW and hence cannot be a leader to W. Given the nature of computed MREs/MLEs and this constraint, it is clear that a cycle cannot be introduced by adding edges as above. Once all the edges have been added, each tree in the forest represents a partition of similar warnings. For each tree, the node without an incoming edge (root) represents the leader of the partition and the other nodes represent the followers. The number of partitions equals the number of trees in the forest.

The following order of steps optimizes the computation of MLEs:

1) Initially, we compute the LWs using MREs which are computed with respect to the input warnings.
2) The MLEs are computed with respect to the LWs from the previous step. This reduces the information being computed by MLEs DFA as the LWs would be a subset of the input warnings.
3) The LWs are refined further using the MLEs so computed.

We explain the partitioning of warnings using the example in Figure 3. With a slight abuse of notation, we denote a node corresponding to a warning by the warning itself. Initially, none of the nodes have an incoming or an outgoing edge. We add directed edges using MREs computed for ZD and AIOB warnings and the resulting forest is shown in Figure 6. The following must be observed -

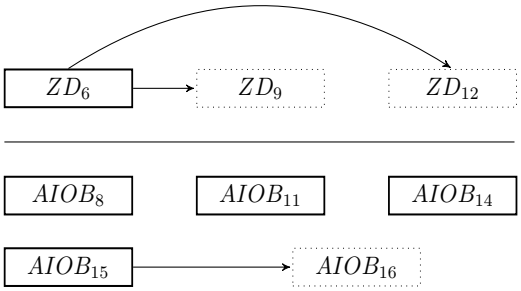


Fig. 6. LWs computed using MREs

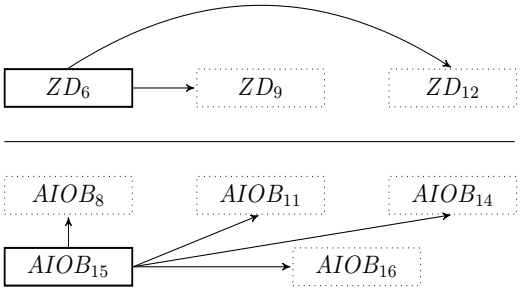


Fig. 7. LWs refined using MLEs

1) There are edges from ZD_6 to ZD_9 and ZD_{12} since the denominator of ZD_6 is MRE at ZD_9 as well as ZD_{12}.
2) Even though the denominator of ZD_9 is MRE at ZD_{12}, its corresponding edge is not added since ZD_9 already has an incoming edge.
3) There are no incoming (outgoing) edges for nodes corresponding to warnings $AIOB_8$, $AIOB_{11}$ and $AIOB_{14}$ as no MREs are computed for (available at) them.

The forest in Figure 6 has 5 LWs (ZD_6, $AIOB_8$, $AIOB_{11}$, $AIOB_{14}$ and $AIOB_{15}$). MLEs are computed with respect to these LWs. The computed MLEs are used to refine these LWs further by adding edges as stated above. Figure 7 shows the forest built after LWs refinement, representing one LW for each property (ZD and AIOB).

C. Implementation for MPG of Warnings

We implemented the *reaching definitions analysis* ([9], [11]) to get the modification points of variables at a given program point. These reaching definitions are computed at an application level in flow and context sensitive manner, using *function summary based* approach [9]. We grouped the LWs based on the similarity of the modification points of their EOI variables.

V. Experiments and Observations

We used TECA to experiment with our proposed grouping techniques and check their impact on review efforts reduction. We selected an embedded application implementing an infotainment system and consisting of 95 tasks running in parallel. These tasks varied from 4 KLOC to 700 KLOC and totaling over 14 MLOC. We verified them for ZD, AIOB and IDP properties as these are very commonly used in software verification.

A. Partitioning based on Similar Warnings

The selected tasks were verified in 5 different settings to observe the impact of MREs and MLEs, and the scope at which they are computed, on redundancy identification. These settings are -

1) W_{np} - verification with no partitioning
2) $Partition_f^R$ - partitioning using *MREs* computed at *function level*
3) $Partition_f$ - partitioning using MREs and MLEs computed at *function level*
4) $Partition_a^R$ - partitioning using *MREs* computed at *application level*
5) $Partition_a$ - partitioning using MREs and MLEs computed at *application level*

We analyzed the called functions as well to compute MREs and MLEs at the function level. Table I provides the results for each of the selected property in these 5 settings. The total number of points for AIOB, ZD and IDP were 73201, 12110 and 825124 respectively. The EOIs of most of the reported warnings referred array elements, directly or indirectly. These were reported as the warnings as TECA does not handle array elements precisely (in order to scale better). We saw a large number of IDP warnings as pointer initialization routines were missing for all the selected tasks.

For a selected property, Table I also compares the number of warnings that are grouped inter-functionally and the analysis time for each verification setting. The number of warnings grouped inter-functionally indicate the instances where a FW and its associated LW are in different functions.

1) Observations from LWs & FWs: Following are some observations from the results in Table I:

- The average percentage of FWs is 45% indicating that nearly half the warnings are followers of others.
- The numbers of warnings grouped inter-functionally are relatively small. This indicates that nearly all the warnings from a partition belong to the same function. This is due to the limitation of arguments and parameters mapping during MREs and MLEs computations. The values for most of the EOIs are propagated through function parameters.
- The MLEs computed at the application level contributed to more FWs (108 for AIOB and 1676 for IDP) when compared with FWs computed using MREs at the application level.
- If we assume review of an FW would have taken 5 seconds on an average, we can expect a reduction of 10.21 hours for AIOB, 0.78 hours for ZD and 186.19 hours for IDP as compared to the manual review efforts in the $Partition_a$ setting.

Some of our other findings from the $Partition_a$ results are as follows:

TABLE I. GROUPING OF WARNINGS - EXPERIMENTAL RESULTS

Comparison Criteria	Run-Time Property	W_{np}	$Partition_f^R$	$Partition_f$	$Partition_a^R$	$Partition_a$	MPG_f	MPG_a
Number of LWs	AIOB	15276	8055	7942	8026	7918	6929	6457
	ZD	1375	814	814	813	813	736	707
	IDP	280971	148721	147047	148587	146911	69095	60479
Percentage of FWs	AIOB	x	47.27	48	47.46	48.16	54.64	57.73
	ZD	x	40.8	40.8	40.87	40.87	46.47	48.58
	IDP	x	47.06	47.66	47.11	47.71	75.40	78.47
Number of warnings grouped inter-functionally	AIOB	x	6	14	55	67	1373	1985
	ZD	x	2	2	4	4	99	205
	IDP	x	0	93	666	711	34375	47643
Analysis Time (Minutes)	AIOB	37.56	53.23	71.28	131.48	161.75	181.3	449.7
	ZD	69.8	102	135.3	158.86	180.48	296.8	439.96
	IDP	861.45	898.26	940.56	979.15	1046.5	1227.48	2835.98

- The maximum FWs for a task was 77.72% (150 out of 193) for AIOB, 68.75% (22 out 32) for ZD and 81.05% (462 out of 570) for IDP.
- There were a number of tasks with more than 60% of FWs: 11 tasks for AIOB, 9 tasks for ZD and 38 tasks for IDP.

Figure 8 compares the counts of AIOB LWs for the largest 20 tasks from W_{np}, $Partition_f$ and $Partition_a$ settings. Figure 9 compares the same counts of ZD LWs. Due to lack of space, we avoid providing such comparison of IDP LWs.

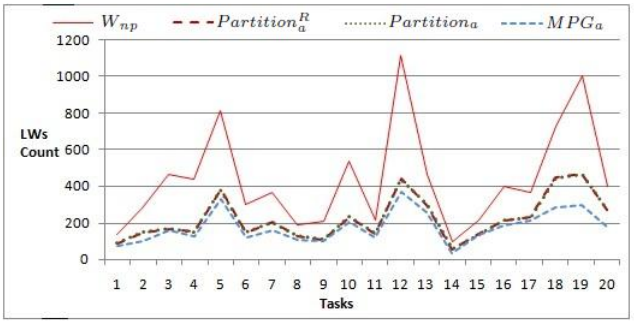


Fig. 8. AIOB LWs for largest 20 tasks

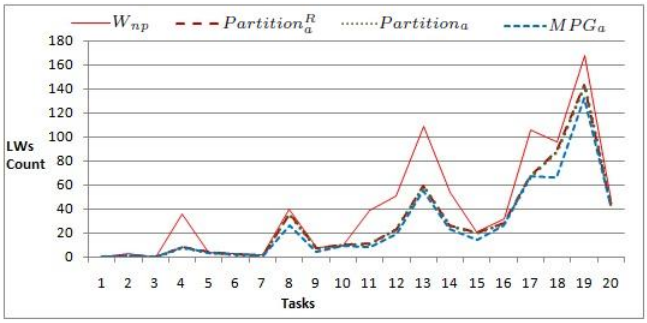


Fig. 9. ZD LWs for largest 20 tasks

2) Performance analysis for LWs computations: The analysis time for each property, as shown in Table I, indicate that the analysis time varies as per the scope at which the MREs and MLEs are computed. Also it can be observed that, the increased analysis time varies as per the property. Figure 10 compares AIOB analysis time for the largest 20 tasks in W_{np}, $Partition_f$ and $Partition_a$ settings. Figure 11 compares the ZD analysis times for those tasks in the same settings.

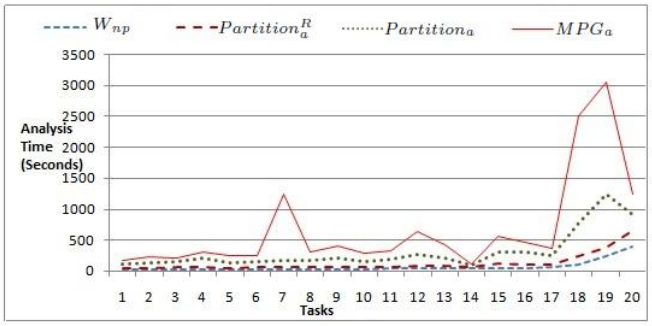


Fig. 10. AIOB analysis time for largest 20 tasks

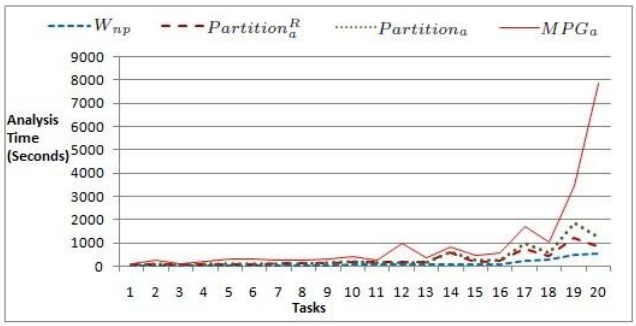


Fig. 11. ZD analysis time for largest 20 tasks

3) Impact on Efforts Reduction: We randomly selected 5 tasks contributing 1000 warnings for each property in the W_{np} setting and got these warnings reviewed by four reviewers. For these tasks, the reviewers also reviewed the LWs from the other 4 grouping settings to observe the impact of partitioning and its scope in efforts reduction. During this process all the LWs were observed to be false positives and their associated FWs, therefore, were not reviewed. This reduced around 15-20% of the review efforts. There are several factors influencing this reduction -

1) the property being verified (for instance, reviewing an IDP warning took longer than reviewing an AIOB or a ZD warning because of the time spent in identifying points-to information for dereferenced pointers)
2) the skills and expertise of reviewer
3) tool support used for code traversal
4) the number of warnings grouped inter-functionally
5) complexity of the code involved (presence of point-

TABLE II. REDUNDANCY IN POLYSPACE WARNINGS

Verification	No. of Common Warnings	No. of FWs	No. of MPG Grouped Warnings
AIOB	56	4	9
IDP	94	11	16

TABLE III. REDUNDANCY IN ASTRÉE WARNINGS

Verification	No. of Common Warnings	No. of FWs	No. of MPG Grouped Warnings
AIOB	210	80	35
IDP	430	130	93

ers, recursive chains, etc.)

B. MPG of Warnings

We performed MPG in the following two settings only for LWs identified in the $Partition_a$ setting -

1) MPG_f - where the modification points are computed at a *function level*
2) MPG_a - where the modification points are computed at the *application level*

Table I also provides the results for these settings where each MPG group is counted as one warning for comparison. The count of inter-functionally grouped warnings indicates the scenarios where a warning program point and the modification points of their EOI variables are in different functions. Their high count highlights the fact that for a lot of warnings in practice, their EOI variables are assigned to in some other function. Further, it can be observed that the analysis time for grouping the warnings as per MPG is affordable, when compared against the redundant warnings identified and the application code size.

Figure 8 also draws the comparison between the counts of AIOB warnings before and after MPG, for the largest 20 tasks. Figure 9 does the same for ZD warnings. Figure 10 and 11 compare the analysis time for AIOB and IDP respectively.

The MPG grouped LWs from the 5 tasks selected earlier were reviewed by the same four reviewers to see its impact on the review efforts reduction. The efforts reduced by more than 40%-50%. Moreover, the MPG technique not only reduced the number of warnings to be reviewed but also helped the users in locating the modification points directly. This could avoid a lot of code traversals which would otherwise have been necessary in order to review the warnings. The combined effect of both the partitioning techniques on review efforts reduction was found to be 50%-60%.

C. Grouping Techniques Against Other Tools

We selected the battery control module of an automotive application of about 60 KLOC to study the impact of our grouping techniques against Polyspace and Astrée. For comparison, we concentrated only on the warnings which were common between TECA and these tools. Tables II and III highlight the observed redundancy in the warnings. The FWs identified with respect to Astrée warnings were computed by the both MREs and MLEs, whereas only MLEs were responsible to find FWs in warnings reported by Polyspace. The warnings from a MPG group belonged to different functions and had their associated modification points in some other function.

VI. RELATED WORK

A lot of work has been done to reduce generated false warnings by improving precision of static analysis using various techniques ([4], [5], [6]). With improved precision, lesser warnings get generated and this reduces the review efforts considerably. The range adjustment approach as used in Polyspace [8], helps in reducing the number of warnings and ultimately efforts in their review. However this approach may run into the issue of multiple verification cycles and sometimes may not even identify some redundant warnings.

Rival targeted helping users in judging a given warning as true error or false warning by proposing a framework for semi-automatic investigation [1]. This framework reduces the burden of tracking the source of alarms in Astrée [7]. Ayewah has proposed use of checklists to enable more detailed review of static analysis warnings [12].

A variety of techniques have been proposed where the priorities of generated warnings are determined based on prior experience of fixing warnings. [13] presents a history-based warning prioritization algorithm by mining warning fix experience obtained from software change history. The work of Spacco et al., as in [14], proposes techniques for tracking the potential defects across the software versions. Some of the other work which targets prioritizing of warnings includes [15], [16], [17] and [18].

A literature survey of actionable alert identification techniques (AAITs) [19] done by Heckman, summarizes 21 AAITs that are used in classifying and prioritizing the actionable alerts. These techniques employ alert type selection, contextual information, graph theory, mathematical and statistical modeling, or machine learning. The classification AAITs divide the generated alerts in two groups: alerts likely to be actionable and alerts likely to be unactionable whereas the prioritization AAITs order the alerts by the likelihood of an alert being actionable.

In contrast, our proposed grouping techniques consider each input warning to be of the same priority and hence all of them are actionable. We identify a warning as redundant only if some other associated warning is found to be a false positive after manual review. None of these AAITs [19] is observed to be similar to our presented grouping techniques.

VII. CONCLUSION

Through our experiments, we got positive answers to the questions which had led to our attempt at review effort reduction. We found that multiple scenarios with redundant warnings occur often in practice. Our experiments demonstrated a high percentage of FWs (nearly 45%) as there are multiple instances of similar expressions of interest (with respect to any given run-time property) in certain code regions. These instances often fall in the same partition and, thus, get eliminated all at once. Since most of the warnings reported by a static tool are false positives, so are the LWs of a number of partitions in our approach. Therefore, such a partitioning of warnings is

quite effective. We observed, on an average, a reduction of at least 20% in the review efforts.

The MPG approach not only helps in avoiding review of some warnings but it also makes it easier to review a warning due to the association of modification points. We found that multiple warnings got values for the variables in their EOIs from the same program points and this helped us avoid reviewing 20% of the warnings. Avoiding multiple code traversals further led to a total of 40-50% reduction in the review efforts as code traversals is a major time-consuming factor during the review process. The techniques, however, fail to identify redundant warnings when a LW or a MPG grouped warning is not found to be a false positive. In these scenarios, review of each grouped warning is required.

We have considered AIOB, ZD and IDP properties in our experiments, however, the proposed grouping algorithms being generic can be applied to the warnings generated for other properties such as overflow-underflow, illegal dereference of pointer. Although the experiments are performed on an embedded domain application written in C, we expect similar benefits on other domain applications as well due to common coding practices. These techniques can be extended further to verify properties in applications coded in other languages too.

Our techniques can be applied in conjunction with some of the existing techniques as well. For instance, we can combine these with the actionable alert identification techniques (AAITs) of [19] wherein we look at an LW only if it is "actionable". This combined approach will be more effective as the unactionable LWs will be skipped from being reviewed. This can be viewed as an additional level of filtering of the warnings. The proposed MPG approach can similarly be applied on top of these AAITs where the actionable or high priority alerts are collected using the AAITs before they are grouped using MPG.

ACKNOWLEDGMENT

We sincerely thank and appreciate Kumar Madhukar, R Venkatesh and Shrawan Kumar for contributing through multiple reviews to make this paper better.

REFERENCES

[1] X. Rival, "Understanding the origin of alarms in astrée," in *SAS*, 2005, pp. 303–319.

[2] P. Cousot and R. Cousot, "Basic concepts of abstract interpretation," in *IN BUILDING THE INFORMATION SOCIETY*. Kluwer Academic Publishers, 2004, p. 4.

[3] P. Cousot, "Abstract interpretation based formal methods and future challenges," in *Informatics*, ser. Lecture Notes in Computer Science, R. Wilhelm, Ed. Springer Berlin Heidelberg, 2001, vol. 2000, pp. 138–156.

[4] M. Junker, R. Huuck, A. Fehnker, and A. Knapp, "Smt-based false positive elimination in static program analysis," in *ICFEM*, 2012, pp. 316–331.

[5] A. Fehnker and R. Huuck, "Model checking driven static analysis for the real world: designing and tuning large scale bug detection," *Innov. Syst. Softw. Eng.*, vol. 9, no. 1, pp. 45–56, 2013.

[6] H. Post, C. Sinz, A. Kaiser, and T. Gorges, "Reducing false positives by combining abstract interpretation and bounded model checking," in *ASE*, 2008, pp. 188–197.

[7] "The Astrée Static Analyzer," http://www.astree.ens.fr/.

[8] P. Munier, *Polyspace*. John Wiley & Sons, Inc., 2012, pp. 123–153.

[9] U. Khedker, A. Sanyal, and B. Sathe, *Data Flow Analysis: Theory and Practice*. Taylor & Francis, 2009. [Online]. Available: http://books.google.co.in/books?id=9PyrtgNBdg0C

[10] "TCS Embedded Code Analyzer (TECA)." [Online]. Available: http://www.tcs.com/resources/brochures/Pages/TCS_Embedded_Code_Analyzer.aspx

[11] J.-F. Collard and J. Knoop, "A comparative study of reaching-definitions analyses," 1998.

[12] N. Ayewah and W. Pugh, "Using checklists to review static analysis warnings," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, ser. DEFECTS '09. New York, NY, USA: ACM, 2009, pp. 11–15. [Online]. Available: http://doi.acm.org/10.1145/1555860.1555864

[13] S. Kim and M. D. Ernst, "Which warnings should i fix first?" in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 45–54. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287633

[14] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking defect warnings across versions," in *Proceedings of the 2006 international workshop on Mining software repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 133–136. [Online]. Available: http://doi.acm.org/10.1145/1137983.1138014

[15] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: an experimental approach," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 341–350. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368135

[16] C. Boogerd and L. Moonen, "Prioritizing software inspection results using static profiling," in *Source Code Analysis and Manipulation, 2006. SCAM '06. Sixth IEEE International Workshop on*, 2006, pp. 149–160.

[17] S. S. Heckman, "Adaptively ranking alerts generated from automated static analysis," *Crossroads*, vol. 14, no. 1, pp. 7:1–7:11, Dec. 2007. [Online]. Available: http://doi.acm.org/10.1145/1349332.1349339

[18] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '07. New York, NY, USA: ACM, 2007, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/1251535.1251536

[19] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363 – 387, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584910002235

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

JSNOSE: Detecting JavaScript Code Smells

Amin Milani Fard
University of British Columbia
Vancouver, BC, Canada
aminmf@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

***Abstract*—JavaScript is a powerful and flexible prototype-based scripting language that is increasingly used by developers to create interactive web applications. The language is interpreted, dynamic, weakly-typed, and has first-class functions. In addition, it interacts with other web languages such as CSS and HTML at runtime. All these characteristics make JavaScript code particularly error-prone and challenging to write and maintain. Code smells are patterns in the source code that can adversely influence program comprehension and maintainability of the program in the long term. We propose a set of 13 JavaScript code smells, collected from various developer resources. We present a JavaScript code smell detection technique called JSNOSE. Our metric-based approach combines static and dynamic analysis to detect smells in client-side code. This automated technique can help developers to spot code that could benefit from refactoring. We evaluate the smell finding capabilities of our technique through an empirical study. By analyzing 11 web applications, we investigate which smells detected by JSNOSE are more prevalent.**

***Index Terms*—JavaScript, code smell, web applications, smell detection**

I. INTRODUCTION

JavaScript is a flexible popular scripting language for developing Web 2.0 applications. It is used to offload core functionality to the client-side web browser and mutate the Document Object Modelling (DOM) tree at runtime to facilitate smooth state transitions. Because of its flexibility JavaScript is a particularly challenging language to write code in and maintain.

The challenges are manifold: First, it is an interpreted language, meaning that there is typically no compiler in the development cycle that would help developers to spot erroneous or unoptimized code. Second, it has a dynamic, weakly-typed, asynchronous nature. Third, it supports intricate features such as prototypes [23], first-class functions, and closures [8]. And finally, it interacts with the DOM through a complex event-based mechanism [29].

All these characteristics make it difficult for web developers who lack in-depth knowledge of JavaScript, to write maintainable code. As a result, web applications written in JavaScript tend to contain many *code smells* [9]. Code smells are patterns in the source code that indicate potential comprehension and maintenance issues in the program. Code smells, once detected, need to be refactored to improve the design and quality of the code.

Detecting code smells manually is time consuming and error-prone. Automated smell detection tools can lower long-term development costs and increase the chances for success [28] by helping to make the code more maintainable.

978-1-4673-5739-5/13/$31.00 © 2013 IEEE

Current work on web application code smell detection is scarce [20] and tools [3], [4], [6], [20] available to web developers to maintain their code are mainly static analyzers and thus limited in their capabilities.

In this paper, we propose a list of code smells for JavaScript-based applications. In total, we consider 13 code smells: 7 are existing well-known smells adapted to JavaScript, and 6 are specific JavaScript code smell types, collected from various JavaScript development resources. We present an automated technique, called JSNOSE, to detect these code smells. Our approach uses a metric-based algorithm, and combines static with dynamic analysis to detect these smells in JavaScript code.

Our work makes the following main contributions:

- We propose a list of JavaScript code smells, collected from various web development resources;
- We present an automated metric-based approach to detect JavaScript code smells;
- We implement our approach in a tool called JSNose, which is freely available;
- We evaluate the effectiveness of our technique in detecting code smells in JavaScript applications;
- We empirically investigate 11 web applications using JSNOSE to find out which smells are more prevalent.

Our results indicate that amongst the smells detected by JSNOSE, lazy object, long method/function, closure smells, coupling between JavaScript, HTML, and CSS, and excessive global variables, are the most prevalent code smells. Further, our study indicates that there exists a strong and significant positive correlation between the types of smells and lines of code, number of functions, number of JavaScript files, and cyclomatic complexity.

II. MOTIVATION AND CHALLENGES

Although JavaScript is increasingly used to develop modern web applications, there is still lack of tool support targeting code quality and maintenance, in particular for automated code smell detection and refactoring.

JavaScript is a dynamic weakly typed and prototype-based scripting language, with first-class functions. Prototype-based programming is a class-free style of object-oriented programming, in which objects can inherit properties from other

objects directly. In JavaScript, prototypes can be redefined at runtime, and immediately affect all the referring objects.

The detection process for many of the traditional code smells [9], [12] in object-oriented languages is dependent on identifying objects, classes, and functions in the code. Unlike most object-oriented languages such as Java or C++, identification of such key language items is not straightforward in JavaScript code. Bellow we explain the major challenges in identifying objects and functions in JavaScript.

JavaScript has a very flexible model of objects and functions. Object properties and their values can be created, changed, or deleted at runtime and accessed via first-class functions. For instance, in the following piece of code a call to the function `foo()` will dynamically create a property *prop* for the object *obj* if *prop* does not already exist.

```
function foo(obj, prop, value){
    obj.prop = value;
}
```

Due to such dynamism, the set of all available properties of an object is not easily retrievable through static analysis of the code alone. Empirical studies [24] reveal that most dynamic features in JavaScript are frequently used by developers and cannot be disregarded in code analysis processes.

Furthermore, functions in JavaScript are first-class values. They can (1) be objects themselves, (2) contain properties and nested function closures, (3) be assigned dynamically to other objects, (4) be stored in variables, objects, and arrays, (5) be passed as arguments to other functions, and (6) be returned from functions. JavaScript also allows the creation (through `eval()`) and execution of new code at runtime, which again makes static analysis techniques insufficient.

Manual analysis and detection of code smells in JavaScript is time consuming, tedious, and error-prone in large code bases. Therefore, automated techniques are needed to support web developers in maintaining their code. Given the challenging characteristics of JavaScript, our goal in this work is to propose a technique that can handle the highly dynamic nature of the language to detect potential code smells effectively.

III. Related Work

Fowler and Beck [9] proposed 22 code smells in object-oriented languages and associated each of them with a possible refactoring. Although code smells in object-oriented languages have been extensively studied in the past, current work on smell detection for JavaScript code is scarce [20]. In this work we study a list of code smells in JavaScript, and propose an automated detection technique. The list of proposed JavaScript code smells in this paper is based on a study of various discussions in online development forums and JavaScript books [19], [20], [21], [22], [10].

Many tools and techniques have been proposed to detect code smells automatically in Java and C++ such as Checkstyle [2], Decor [17], and JDeodorant [27]. A common heuristic-based approach to code smell detection is the use of code metrics and user defined thresholds [17], [7], [18], [25], [13]. Similarly we adopt a metric-based smell detection strategy. Our approach is different from such techniques in the sense that due to the dynamic nature of JavaScript, we propose a code smell technique that combines static with dynamic analysis.

Cilla [15] is a tool that similar to our work applies dynamic analysis but for detecting unused CSS code in relation to dynamically mutated DOM elements. A case study conducted with Cilla revealed that over 60% of CSS rules are unused in real-world deployed web applications, and eliminating them could vastly improve the size and maintainability of the code.

A more closely related tool is WebScent [20], which detects client-side smells that exist in embedded code within scattered server-side code. Such smells can not be easily detected until the client-side code is generated. After detecting smells in the generated client-side code, WebScent locates the smells in the corresponding location in the server-side code. WebScent primarily identifies mixing of HTML, CSS, and JavaScript, duplicate code in JavaScript, and HTML syntax errors. Our tool, JSNose, can similarly identify JavaScript code smells generated via server-side code. However, we propose and target a larger set of JavaScript code smells and unlike the manual navigation of the application in WebScent, we apply automated dynamic exploration using a crawler. Another advantage of JSNose is that it can infer dynamic creation/change of objects, properties, and functions at runtime, which WebScent does not support.

A number of industrial tools exist that aim at assisting web developers with maintaining their code. For instance, WARI [6] examines dependencies between JavaScript functions, CSS styles, HTML tags and images. The goal is to statically find unused images as well as unused and duplicated JavaScript functions and CSS styles. Because of the dynamic nature of JavaScript, WARI cannot guarantee the correctness of the results. JSLint [4] is a static code analysis tool written in JavaScript that validates JavaScript code against a set of good coding practices. The code inspection tends to focus on improving code quality from a technical perspective. The Google Closure Compiler [3] is a JavaScript optimizer that rewrites JavaScript code to make it faster and more compact. It helps to reduce the size of JavaScript code by removing comments and unreachable code.

IV. JavaScript Code Smells

In this section, we propose a list of code smells for JavaScript-based applications. Of course a list of code smells can never be complete as the domain and projects that the code is used in may vary. Moreover, code smells are generally subjective and imprecise, i.e., they are based on opinions and experiences [28]. To mitigate this subjective nature of code smells, we have collected these smells by studying various online development resources [5], [10], [19], [21], [22], [26] and books [8], [31], [32] that discuss bad JavaScript coding patterns.

In total, we consider 13 code smells in JavaScript. Although JavaScript has its own specific code smells, most of the generic code smells for object-oriented languages [9], [12]

can be adapted to JavaScript as well. Since JavaScript is a class-free language and objects are defined directly, we use the notion of "object" instead of "class" for these generic smells. The generic smells include the following 7: *Empty catch blocks* (poor understanding of logic in the try), *Large object* (too many responsibilities), *Lazy object* (does too little), *Long functions* (inadequate decomposition), *Long parameter list* (need for an object), *Switch statements* (duplicated code and high complexity), and *Unused/dead code* (never executed or unreachable code).

In addition to the generic smells, we propose 6 types of JavaScript code smells in this section as follows.

A. Closure Smells

In JavaScript, it is possible to declare nested functions, called `closures`. Closures make it possible to emulate object-oriented notions such as `public`, `private`, and `privileged` members. Inner functions have access to the parameters and variables — except for `this` and `argument` variables — of the functions they are nested in, even after the outer function has returned [8]. We consider four smells related to the concept of function closures in JavaScript.

Long scope chaining. Functions can be multiply-nested, thus closures can have multiple scopes. This is called "scope chaining" [5], where inner functions have access to the scope of the functions containing them. An example is the following code:

```
function foo(x) {
  var tmp = 3;
  function bar(y) {
      ++tmp;
      function baz(z) {
        document.write(x + y + z + tmp);
      }
      baz(3);
  }
  bar(10);
}
foo(2);  // writes 19 i.e., 2+10+3+4
```

This nested function style of programming is useful to emulate privacy, however, using too many levels of nested closures over-complicates the code, making it hard to comprehend and maintain. Moreover, identifier resolution performance is directly related to the number of objects to search in the scope chain [31]. The farther up in the scope chain an identifier exists, the longer the search goes on and the longer time it takes to access that variable.

Closures in loops. Inner functions have access to the actual variables of their outer functions and not their copies. Therefore, creating functions within a loop can cause confusion and be wasteful computationally [8]. Consider the following example:

```
var addTheHandler = function (nodes) {
    for (i = 0; i < nodes.length; i++) {
        nodes[i].onclick = function (e) {
            document.write(i);
        };
    }
};
addTheHandler(document.getElementsByTagName("div"));
```

Assume that there are three `div` DOM elements present. If the developer's actual intention is to display the ordinal of the `div` nodes when a node is clicked, then the result will not be what she expected as the length of nodes, 3, will be returned instead of the node's ordinal position. In this example, the value of `i` in the `document.write` function is assigned when the `for` loop is finished and the inner anonymous function is created. Therefore, the variable `i` has the value of `nodes.length`, which is 3 in this case. To avoid this confusion and potential mistake, the developer can use a helper function outside of the loop that will deliver a function binding to the current local value of `i`.

Variable name conflict in closures. When two variables in the scopes of a closure have the same name, there is a name conflict. In case of such conflicts, the inner scope takes precedence. Consider the following example [5]:

```
function outside() {
   var a = 10;
   function inside(a) {
      return a;
   }
   return inside;
}
result = outside()(20);  \\ result: 20
```

In this example, there is a name conflict between the variable `a` in `outside()` and the function parameter `a` in `inside()` that takes precedence. We consider this a code smell as it makes it difficult to comprehend the actual intended value assignment. Due to scope chain precedence, the value of `result` is now 20. However, it is not evident from the code whether this is the intended result (or perhaps 10).

Moreover, dynamic typing in JavaScript makes it possible to reuse the *same* variable for *different* types at runtime. Similar to the variable name conflict issue, this style of programming reduces readability and in turn maintainability. Thus, declaring a new variable with a dedicated unique name is the recommended refactoring. This issue is not restricted to closures, or to nested functions.

Accessing the `this` reference in closures. Due to the design of JavaScript language, when an inner function in a closure is invoked, `this` becomes bounded to the global object and not to the `this` variable of the outer function [8]. We consider the usage of `this` in closures a code smell as it is a potential symptom for mistakes. As a refactoring workaround, the developer can assign the value of the `this` variable of the outer function to a new variable `that` and then use `that` in the inner function [8].

B. Coupling between JavaScript, HTML, and CSS

In web applications, HTML is meant for presenting content and structure, CSS for styling, and JavaScript for functional behaviour. Keeping these three entities separate is a well-known programming practice, known as *separation of concerns*. Unfortunately, web developers often mix JavaScript code with markup and styling code [20], which adversely influences program comprehension, maintenance and debugging efforts in web applications. We categorize the tight coupling

of JavaScript with HTML and CSS code into the following three code smells types:

JavaScript in HTML. One common way to register an event listener in web applications is via inline assignment in the HTML code. We consider this inline assignment of event handlers a code smell as it tightly couples the HTML code to the JavaScript code. An example of such a coupling is shown below:

```
<button onclick="foo();" id="myBtn"/>
```

This smell can be refactored by removing the `onclick` attribute from the button in HTML and using the `addEventListener` function of DOM Level 2 [29] to assign the event handler through JavaScript:

```
<button id="myBtn"/>

function foo() {
    // code
}
var btn = document.getElementById("myBtn");
btn.addEventListener("click", foo, false);
```

This could be further refactored using the jQuery library as `$("#myBtn").on("click", foo);`.

Note that JavaScript code within the `<script>` tag in HTML code can be seen as a code smell [20]. We do not consider this a code smell as it does not affect comprehension nor maintainability, although separating the code to a JavaScript file is preferable.

HTML in JavaScript. Extensive DOM API calls and embedded HTML strings in JavaScript complicate debugging and software evolution. In addition, editing markup is believed to be less error prone than editing JavaScript code [32]. The following code is an example of embedded HTML in JavaScript [10]:

```
// add book to the list
var book = doc.createElement("li");
var title = doc.createElement("strong");
titletext = doc.createTextNode(name);
title.appendChild(titletext);
var cover = doc.createElement("img");
cover.src = url;
book.appendChild(cover);
book.appendChild(title);
bookList.appendChild(book);
```

To refactor this code smell, we can move the HTML code to a template (`book_tpl.html`):

```
<li><strong>TITLE</strong><img src="COVER"/></li>
```

The JavaScript code would then be refactored as:

```
var tpl = loadTemplate("book_tpl.html");
var book = tpl.substitute({TITLE: name,  COVER: url});
bookList.appendChild(book);
```

Another example this smell is using long strings of HTML in jQuery function calls [22]:

```
$('\#news')
   .append('<div class="gall"><a href="javascript:void(0)↩
        ">Linky</a></div>')
   .append('<button onclick="app.doStuff()">Button</↩
        button>');
```

CSS in JavaScript. Setting the presentation style of DOM elements by assigning their `style` properties in JavaScript is a code smell [10]. Keeping styling code inside JavaScript is asking for maintenance problems. Consider the following example:

```
div.onclick = function(e) {
   var clicked = this;
   clicked.style.border = "1px solid blue";
}
```

The best way to change the style of an element in JavaScript is by manipulating CSS classes properly defined in CSS files [10], [32]. The above code smell can be refactored as follows:

```
\\ CSS file:
.selected{border: 1px solid blue;}
\\ JavaScript:
div.onclick = function(e) {
   this.setAttribute("class","selected");
}
```

C. Excessive Global Variables

Global variables are accessible from anywhere in JavaScript code, even when defined in different files loaded on the same page. As such, naming conflicts between global variables in different JavaScript source files is common, which affects program dependability and correctness. The higher the number of global variables in the code, the more dependent existing modules are likely to be; and dependency increases error-proneness, and maintainability efforts [21]. Therefore, we see the excessive use of global variables as a code smell in JavaScript. One way to mitigate this issue is to create a single global object for the whole application that contains all the global variables as its properties [8]. Grouping related global variables into objects is another remedy.

D. Long Message Chain

Long chaining of functions with the dot operator can result in complex control flows that are hard to comprehend. This style of programming happens frequently when using the jQuery library. One extreme example is shown bellow [22]:

```
$('a').addClass('reg-link').find('span').addClass('inner'↩
     ).end().find('div').mouseenter(mouseEnterHandler).↩
     mouseleave(mouseLeaveHandler).end().explode();
```

Long chains are unreadable specially when a large amount of DOM traversing is taking place [22].

Another instance of this code smell is too much cascading. Similar to object-oriented languages such as Java, in JavaScript many methods calls can be cascaded on the same object sequentially within a single statement. This is possible when the methods return the `this` object. Cascading can help to produce expressive interfaces that perform much work at once. However, the code written this way tends to be harder to follow and maintain. The following example is borrowed from [8]:

```
getElement('myBoxDiv').move(350, 150).width(100).height↩
     (100).color('red').border('10px outset').padding('4px↩
     ').appendText("Please stand by").on('mousedown', ↩
     function (m) {
    this.startDrag(m, this.getNinth(m));}).on('mousemove'↩
         , 'drag').on('mouseup', 'stopDrag').tip("This box↩
          is resizeable");
```

A possible refactoring to shorten the message chain is to break the chain into more general methods/properties for that object which incorporate longer chains.

E. Nested Callback

A callback is a function passed as an argument to another (parent) function. Callbacks are executed after the parent function has completed its execution. Callback functions are typically used in asynchronous calls such as timeouts and XMLHttpRequests (XHRs). Using excessive callbacks, however, can result in hard to read and maintain code due to their nested anonymous (and usually asynchronous) nature. An example of a nested callback is given below [26]:

```
setTimeout(function () {
    xhr("/greeting/", function (greeting) {
        xhr("/who/?greeting=" + greeting, function (who) ↩
            {
            document.write(greeting + " " + who);
        });
    });
}, 1000);
```

A possible refactoring to resolve unreadable nested callbacks is to split the functions and pass a reference to another function [1]. The above code can be rewritten as bellow:

```
setTimeout(foo,1000);
function foo() {
 xhr("/greeting/", bar);
}
function bar(greeting) {
 xhr("/who/?greeting=" + greeting, baz);
}
function baz(who) {
 document.write(greeting + " " + who);
}
```

F. Refused Bequest

JavaScript is a class-free prototypal inheritance language, i.e., an object can inherit properties from another object, called a prototype object. A JavaScript object that does not use/override many of the properties it inherits from its prototype object is an instance of a refused bequest [9] soft code smell. In the following example, the `student` object inherits from its prototype parent `person`. However, `student` only uses one of the five properties inherited from `person`, namely `fname`.

```
var person={fname:"John", lname:"Smith", gender:"male", ↩
     age:28, location:"Vancouver"};
var student = Object.create(person);
...
student.university = "UBC";
document.write(student.fname + " studies at " + student.↩
     university);
```

A simple refactoring, similar to the push down field/method proposed by Fowler [9], could be to eliminate the inheritance altogether and add the required property (`fname`) of the prototype to the object that refused the bequest.

V. Smell Detection Mechanism

In this section, we present our JavaScript code smell detection mechanism, which is capable of detecting the code smells discussed in the previous section.

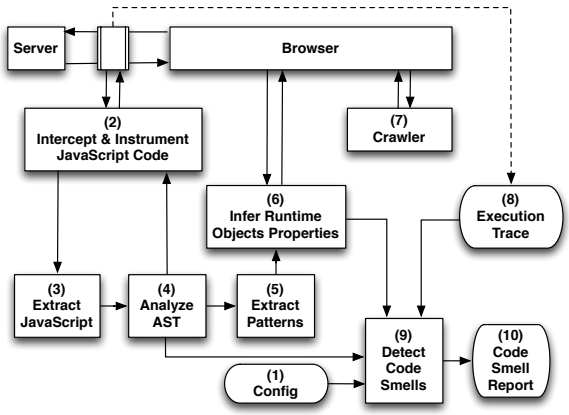


Fig. 1. Processing view of JSNose, our JavaScript code smell detector.

A common heuristic-based approach to detect code smells is the use of source code metrics and thresholds [7], [13], [17], [18], [25]. In this work, we adopt a similar metric-based approach to identify smelly sections of JavaScript code.

In order to calculate the metrics, we first need to extract objects, functions, and their relationships from the source code. Due to the dynamic nature of JavaScript, static code analysis alone will not suffice, as discussed in Section II. Therefore, in addition to static code analysis, we also employ dynamic analysis to monitor and infer information about objects and their relations at runtime.

Figure 1 depicts an overview of our approach. At a high level, (1) the configuration, containing the defined metrics and thresholds, is fed into the code smell detector. We automatically (2) intercept the JavaScript code of a given web application, by setting up a proxy between the server and the browser, (3) extract JavaScript code from all .js and HTML files, (4) parse the source code into an Abstract Syntax Tree (AST) and analyze it by traversing the tree. During the AST traversal, the analyzer visits all program entities, objects, properties, functions, and code blocks, and stores their structure and relations. At the same time, we (2) instrument the code to monitor statement coverage, which is used for unused/dead code smell detection. Next, we (7) navigate the instrumented application in the browser to produce an execution trace, through an automated dynamic crawler, and (8) collect and use execution traces to calculate code coverage. We (5) extract patterns from the AST such as names of objects and functions, and (6) infer JavaScript objects, their types, and properties dynamically by querying the browser at runtime. Finally, (9) based on all the static and dynamic data collected, we detect code smells (10) using the metrics.

A. Metrics and Criteria Used for Smell Detection

Table I presents the metrics and criteria we use in our approach to detect code smells in JavaScript applications. Some of these metrics and their corresponding thresholds have been proposed and used for detecting code smells in object-oriented languages [7], [13], [17], [18], [25]. In addition, we

TABLE I
METRIC-BASED CRITERIA FOR JAVASCRIPT CODE SMELL DETECTION.

Code smell	Level	Detection method	Detection criteria	Metric
Closure smell	Function	Static & Dynamic	$LSC > 3$	LSC: Length of scope chain
Coupling JS/HTML/CSS	File	Static & Dynamic	$JSC > 1$	JSC: JavaScript coupling instance
Empty catch	Code block	Static	$LOC(catchBlock) = 0$	LOC: Lines of code
Excessive global variables	Code block	Static & Dynamic	$GLB > 10$	GLB: Number of global variables
Large object	Object	Static & Dynamic	[30]: $LOC(obj) > 750$ or $NOP > 20$	NOP: Number of properties
Lazy object	Object	Static & Dynamic	$NOP < 3$	NOP: Number of properties
Long message chain	Code block	Static	$LMC > 3$	LMC: Length of message chain
Long method/function	Function	Static & Dynamic	[13], [30]: $MLOC > 50$	MLOC: Method lines of code
Long parameter list	Function	Static & Dynamic	[30]: $PAR > 5$	PAR: Number of parameters
Nested callback	Function	Static & Dynamic	$CBD > 3$	CBD: Callback depth
Refused bequest	Object	Static & Dynamic	[13]: $BUR < \frac{1}{3}$ and $NOP > 2$	BUR: Base-object usage ratio
Switch statement	Code block	Static	$NOC > 3$	NOC: Number of cases
Unused/dead code	Code block	Static & Dynamic	$EXEC = 0$ or $RCH = 0$	EXEC: Execution count RCH: Reachability of code

propose new metrics and criteria to capture the characteristics of JavaScript code smells discussed in Section IV.

Closure smell. We identify long scope chaining and accessing `this` in closures. If the length of scope chain (LSC) is greater than 3, or if `this` is used in an inner function closure, we report it as a closure smell instance.

Coupling JS/HTML/CSS. We count the number of occurrences of JavaScript within HTML tags, and CSS in JavaScript as described in Section IV-B. Our tool reports all such JavaScript coupling instances as code smell.

Empty catch. Detecting empty catches is straightforward in that the number of lines of code (LOC) in the catch block should be zero.

Excessive global variables. We extract global variables in JavaScript, which can be defined in three ways: (1) using a `var` statement outside of any function, such as `var x = value;`, (2) adding a property to the `window` global object, i.e., the container of all global variables, such as `window.foo = value;`, and (3) using a variable without declaring it by `var`. If the number of global variables (GLB) exceeds 10, we consider it as a code smell.

Large/Lazy object. An object that is doing too much or not doing enough work should be refactored. Large objects may be restructured or broken into smaller objects, and lazy objects maybe collapsed or combined into other classes. If an object's lines of code is greater than 750 or the number of its methods is greater than 20, it is identified as a large object [30]. We consider an object lazy, if the number of its properties (NOP) is less than 3.

Long message chain. If the length of a message chain (LMC), i.e., the number of items being chained by dots as explained in Section IV-D, in a statement is greater than 3, we consider it a long message chain and report it as a smell.

Long method/function. A method with more than 50 lines of code (MLOC) is identified as a long method smell [13], [30].

Long parameter list. We consider a parameter list long when the number of parameters (PAR) exceeds 5 [30].

Nested callback. We identify nested functions that pass a function type as an argument. If the callback depth (CBD) exceeds 3, we report it as a smell.

Refused bequest. If an object uses or specializes less than a third of its parent prototype, i.e., base-object usage ratio (BUR) is less than $\frac{1}{3}$, it is considered as refused parent bequest [13]. Further, the number of methods and the cyclomatic complexity of the child object should be above average since simple and small objects may unintentionally refuse a bequest. In our work, we slightly change this criteria to the constraint of NOP>2, i.e., not to be a lazy small object.

Switch statement. The problem with switch statements is duplicated code. Typically, similar switch statements are scattered throughout a program. If one adds or removes a clause in one switch, she often has to find and repair the others too [9], [12]. When the number of switch cases (NOC) is more than three, it is considered as a code smell. This can also be applied to `if-then-else` statements with more than three branches.

Unused/dead code. Unused/dead code has negative effects on maintainability as it makes the code unnecessarily more difficult to understand [21], [14]. Unlike languages such as Java, due to the dynamic nature of JavaScript it is quite challenging to reason about dead JavaScript code statically. Hence, if the execution count (EXEC) of an statement remains 0 after executing the web application, we report it as a candidate unused/dead code. Reachability of code (RCH) is another metric we use to identify unreachable code.

B. Combining Static and Dynamic Analysis

Algorithm 1 presents our smell detection method. The algorithm is generic in the sense that the metric-based static and dynamic smell detection procedures can be defined and used according to any smell detection criteria. Given a JavaScript application A, a maximum crawling time t, and a set of code smell criteria τ, the algorithm generates a set of code smells CS.

The algorithm starts by looking for inline JavaScript code embedded in HTML (line 3). All JavaScript code is then

Algorithm 1: JavaScript Code Smell Detection

```
   input : A JavaScript application A, the maximum exploration time
           t, the set of smell metric criteria τ
   output: The list of JavaScript code smells CS
1  CS ← ∅
   Procedure EXPLORE() begin
2      while TIMELEFT(t) do
3          CS ← CS ∪ DETECTINLINEJSINHTML(τ)
4          code ← EXTRACTJAVASCRIPT(A)
5          AST ← PARSTOAST(code)
6          VISITNODE(AST.root)
7          ASTinst ← INSTRUMENT(AST)
8          INJECTJAVASCRIPTCODE(A, ASTinst)
9          C ← EXTRACTCLICKABLES(A)
10         for c ∈ C do
11             dom ← browser.GETDOM()
12             robot.FIREEVENT(c)
13             new_dom ← browser.GETDOM()
14             CS ← CS ∪ DETECTDYNAMICALLY(τ)
15             if dom.HASCHANGED(new_dom) then
16                 EXPLORE(A)
17     CS ← CS ∪ DETECTUNUSEDCODE()
18     return CS
   Procedure VISITNODE(ASTNode) begin
19     CS ← CS ∪ DETECTSTATICALLY(node, τ)
20     for node ∈ ASTNode.getChildren() do
21         VISITNODE(node)
```

extracted from JavaScript files and HTML `<script>` tags (line 4). An AST of the extracted code is then generated using a parser (line 5). This AST is traversed recursively (lines 6, 19-21) to detect code smells using a static analyzer. Next the AST is instrumented (line 7) and transformed back to the corresponding JavaScript source code and passed to the browser (lines 8). The crawler then navigates (line 9-16) the application, and potential code smells are explored dynamically (line 9 14). After the termination of the exploration process, unused code is identified based on the execution trace and added to the list of code smells (line 17), and the resulting list of smells is returned (line 18).

Next, we present the relevant static and dynamic smell detection processes in detail.

Static Analysis. The static code analysis (Line 19) involves analyzing the AST by traversing the tree. During this step, we extract CSS style usage, objects, properties, inheritance relations, functions, and code blocks to calculate the smell metrics. If the calculated metrics violate the given criteria (τ), the smell is returned.

There are different ways to create objects in JavaScript. In this work, we only consider two main standard forms of using object literals, namely, through (1) the `new` keyword, and (2) `Object.create()`. To detect the prototype of an object, we consider both the non-standard form of using the `__proto__` property assignment, and the more general constructor functions through `Object.create()`.

In order to detect unreachable code, we search the AST nodes for `return`, `break`, `continue`, and `throw` statements. Whatever a statement is found right after these statements that is on the same node level in the AST, we mark it as potential unreachable code.

Dynamic Analysis. Dynamic analysis (Line 14) is performed for two reasons:

1) To calculate many of the metrics in Table I, we need to monitor the creation/update of functions, objects, and their properties at runtime. To that end, a combination of static and dynamic analysis should be applied. The dynamic analysis is performed by executing a piece of JavaScript code in the browser, which enables retrieving a list of all global variables, objects, and functions (own properties of the `window` object) and dynamically detecting prototypes of objects (using `getPrototypeOf()` on each object). However, local objects in functions are not accessible via JavaScript code execution in the global scope. Therefore, we use static analysis and extract the required information from the parsed AST. The objects, functions, and properties information gathered this way is then fed to the smell detector process.
2) To detect unused/dead code we need to collect execution traces for measuring code coverage. Therefore, we instrument the code and record which parts of it are invoked by exploring the application through automated crawling. However, this dynamic analysis can give false positives for non-executed, but reachable code. This is a limitation of any dynamic analysis approach since there is no guarantee of completeness (such as code coverage).

Note that our approach merely reports candidate code smells and the decision will always be upon developers whether or not to refactor the code smells.

C. Implementation

We have implemented our approach in a tool called JSNOSE, which is publicly available.[1] JSNOSE operates automatically, does not modify the web browser, is independent of the server technology, and requires no extra effort from the user. We use the WebScarab proxy to intercept the JavaScript/HTML code. To parse the JavaScript code to an AST and instrument the code, we use Mozilla Rhino.[2] To automatically explore and dynamically crawl the web application, we use CRAWLJAX [16]. The output of JSNOSE is a text file that lists all detected JavaScript code smells with their corresponding line numbers in a JavaScript file or an HTML page.

VI. EMPIRICAL EVALUATION

We have conducted an empirical study to evaluate the effectiveness and real-world relevance of JSNOSE. Our study is designed to address the following research questions:

RQ1: How effective is JSNOSE in detecting JavaScript code smells?

RQ2: Which code smells are more prevalent in web applications?

[1] http://salt.ece.ubc.ca/content/jsnose/

[2] https://github.com/mozilla/rhino/

TABLE II
EXPERIMENTAL OBJECTS.

ID	Name	#JS files	JS LOC	#Functions	Average CC	Average MI	Description	Resource
1	PeriodicTable	1	71	9	12	116	An AJAX-based periodic table of the elements	http://code.jalenack.com/periodic/
2	CollegeVis	1	177	30	11	119	A JavaScript-based visualization tool	https://github.com/nerdyworm/collegesvis
3	ChessGame	2	198	15	102	105	A JavaScript-based simple game	p4wn.sourceforge.net
4	Symbolistic	1	203	20	28	109	A JavaScript-based simple game	http://10k.aneventapart.com/2/Uploads/652
5	Tunnel	0	234	32	29	116	A JavaScript-based simple game	http://arcade.christianmontoya.com/tunnel
6	GhostBusters	0	278	26	45	97	A JavaScript-based simple game	http://10k.aneventapart.com/2/Uploads/657
7	TuduList	4	782	89	106	94	An AJAX-based todo lists manager in J2EE and MySQL	julien-dubois.com/tudu-lists/
8	FractalViewer	8	1245	125	35	116	A JavaScript-based fractal zoomer	http://onecm.com/projects/canopy
9	PhotoGallery	5	1535	102	53	102	An AJAX-based photo gallery in PHP without MySQL	sourceforge.net/projects/rephormer
10	TinySiteCMS	13	2496	462	54	115	An AJAX-based CMS in PHP without MySQL	tinysitecms.com
11	TinyMCE	174	26908	4455	67	101	A JavaScript-based WYSIWYG editor	tinymce.com

RQ3: Is there a correlation between JavaScript code smells and source code metrics?

Our experimental data along with the implementation of JSNOSE are available for download.[1]

A. Experimental Objects

We selected 11 web applications that make extensive use of client-side JavaScript, and fall under different application domains. The experimental objects along with their source code metrics are shown in Table II. In the calculation of these source code metrics, we included inline HTML JavaScript code, and excluded blank lines, comments, and common JavaScript libraries such as jQuery, DWR, Scriptaculous, Prototype, and google-analytics. Note that we also exclude these libraries in the instrumentation step. We use CLOC[3] to count the JavaScript lines of code (JS LOC). Number of functions (including anonymous functions), cyclomatic complexity (CC), and maintainability index (MI) are all calculated using complexityReport.js.[4] The reported CC and MI are across all JavaScript functions in each application.

B. Experimental Setup

We confine the dynamic crawling time for each application to 10 minutes, which is acceptable in a maintenance environment. Of course, the more time we designate for exploring the application, the higher statement coverage we may get and thus more accurate the detection of unused/dead code. For the crawling configuration, we set no limits on the crawling depth nor the maximum number of DOM states to be discovered. The criteria for code smell metrics are configured according to those presented in Table I.

To evaluate the effectiveness of JSNOSE (RQ1), we validate the produced results by JSNOSE against manual code inspection. Similar to [17], we measure precision and recall as follows:

Precision is the rate of true smells identified among the detected smells: $\frac{TP}{TP+FP}$

Recall is the rate of true smells identified among the existing smells: $\frac{TP}{TP+FN}$

[3]http://cloc.sourceforge.net

[4]https://npmjs.org/package/complexity-report/

where TP (true positives), FP (false positives), and FN (false negatives) respectively represent the number of correctly detected smells, falsely detected smells, and missed smells. To count TP, FP, and FN in a timely fashion while preserving accuracy, we only consider the first 9 applications since the last 2 applications have relatively larger code bases. In our manual validation process, we also consider runtime created/modified objects and functions that are inferred during JSNOSE dynamic analysis. It is worth mentioning that this manual process is a labour intensive task, which took approximately 6.5 hours for the 9 applications.

Note that the precision-recall values for detecting unused/dead code smell is calculated considering only "unreachable" code, which is code after an unconditional `return` statement. This is due to the fact that the accuracy of dead code detection depends on the running time and dynamic exploration strategy.

To measure the prevalence of JavaScript code smells (RQ2), we ran JSNOSE on all the 11 web applications and counted each smell instance.

To evaluate the correlation between the number of smells and application source code metrics (RQ3), we use R[5] to calculate the non-parametric Spearman correlation coefficients as well as the p-values. The Spearman correlation coefficient does not require the data to be normally distributed [11].

C. Results

Effectiveness (RQ1). We report the precision and recall in the first 5 rows of Table III. The reported TP_{total}, FP_{total}, and FN_{total}, are the sum of TP, FP, and FN values for the first 9 applications. Our results show that JSNOSE has an overall precision of 93% and an average recall of 98% in detecting the JavaScript code smells, which points to its effectiveness.

We observed that most false positives detected are related to large/lazy objects and refused bequest, which are primitive variables, object properties, and methods in jQuery. This is due to the diverse coding styles and different techniques in object manipulations in JavaScript, such as creating and initializing arrays of objects. There were a few false negatives

[5]http://www.r-project.org

TABLE III
PRECISION-RECALL ANALYSIS (BASED ON THE FIRST 9 APPLICATIONS), AND DETECTED CODE SMELL STATISTICS (FOR ALL 11 APPLICATIONS).

	S1. Closure smells	S2. Coupling JS/HTML/CSS	S3. Empty catch	Number of global variables	S4. Excessive global variables	S5. Large object	S6. Lazy object	S7. Long message chain	S8. Long method/function	S9. Long parameter list	S10. Nested callback	S11. Refused bequest	S12. Switch statement	S13. Unreachable code	S13. Unused/dead code	Number of smell instances	Number of types of smells
TP_{total}	19	171	16	200	-	14	391	87	25	12	1	13	10	0	n/a	959	n/a
FP_{total}	0	0	0	0	-	4	73	0	0	0	0	6	0	0	n/a	83	n/a
FN_{total}	6	0	0	0	-	1	2	8	0	0	0	0	0	0	n/a	17	n/a
$Precision_{total}$	100%	100%	100%	100%	-	78%	85%	100%	100%	100%	100%	68%	100%	n/a	n/a	92%	n/a
$Recall_{total}$	76%	100%	100%	100%	-	94%	99%	92%	100%	100%	100%	100%	100%	n/a	n/a	98%	n/a
PeriodicTable	1	2	0	6	-	4	10	0	0	0	0	0	0	0	28%	23	4
CollegeVis	1	0	0	17	+	0	32	0	2	0	1	0	0	0	22%	53	5
ChessGame	0	7	0	**39**	+	3	9	4	0	2	0	0	0	0	36%	64	6
Symbolistic	0	0	0	4	-	0	17	0	1	0	0	1	0	0	20%	23	3
Tunnel	**9**	0	0	15	+	0	28	0	2	0	0	0	0	0	44%	54	4
GhostBusters	2	0	0	4	-	0	38	0	2	3	0	0	0	0	45%	49	4
TuduList	**6**	**47**	0	**45**	+	**6**	**138**	**78**	**12**	2	0	2	**7**	0	**65%**	343	10
FractalViewer	0	16	0	**40**	+	**7**	**117**	4	5	5	0	**16**	2	0	36%	212	9
PhotoGallery	0	**99**	**16**	30	+	0	73	1	1	0	0	0	1	0	**64%**	221	7
TinySiteCMS	2	7	0	**82**	+	3	13	4	3	**58**	0	0	0	0	22%	172	8
TinyMCE	3	3	1	4	-	5	23	4	0	2	1	3	3	0	**63%**	52	10
Average	2.2	16.5	1.5	26	+	2.5	45.2	8.6	2.6	6.5	0.2	2	1.2	0	40%	115	6.4
#Smelly apps	7	7	2	n/a	7	6	11	6	8	6	2	4	4	0	n/a	n/a	n/a
%Smelly apps	**64%**	**64%**	18%	n/a	**64%**	55%	**100%**	55%	**73%**	55%	18%	36%	36%	0%	n/a	n/a	n/a

in closure smells and long message chain, which are due to the permissive nature of jQuery syntax, complex chains of methods, array elements, as well as jQuery objects created via `$()` function.

Code smell prevalence (RQ2). Table III shows the frequency of code smells in each of the experimental objects. The results show that among the JavaScript code smells detected by JSNOSE, lazy object, long method/function, closure smells, coupling JS/HTML/CSS, and excessive global variables, are the most prevalent smells (appeared in 100%-64% of the experimental objects).

Tunnel and TuduList use many instances of `this` in closures. Major coupling smells in TuduList and PhotoGallary are with the use of CSS in JavaScript. Refused bequest are most observed in FractalViewer in objects inheriting from geometry objects. The high percentage of unused/dead code reported for TuduList, PhotoGallary, and TinyMCE is in fact not due to dead code per se, but is mainly related to the existence of admin pages and parts of the code which require precise data inputs that were not provided during the crawling process. On the other hand, TinyMCE has a huge number of possible actions and features that could not be exercised in the designated time of 10 minutes.

Correlations (RQ3). Table IV shows the Spearman correlation coefficients between the source code metrics and the total number of smell instances/types. The results show that there exists a strong and significant positive correlation between the *types of smells* and LOC, number of functions, number of JavaScript files, and cyclomatic complexity. A weak correlation is also observed between the *number of smell instances* and the aforementioned source code metrics. Surprisingly, for the maintainability index (MI) we do not see any significant correlation with the types or number of code smells.

TABLE IV
SPEARMAN CORRELATION COEFFICIENTS BETWEEN NUMBER OF CODE SMELLS AND CODE QUALITY METRICS.

Metric	Total number of smell instances	Total number of types of smells
Lines of code	$(r = 0.53, p = 0.05)$	$(r = 0.70, p = 0.01)$
# Functions	$(r = 0.57, p = 0.03)$	$(r = 0.76, p = 0.00)$
# JavaScript files	$(r = 0.53, p = 0.05)$	$(r = 0.85, p = 0.00)$
Cyclomatic complexity	$(r = 0.63, p = 0.02)$	$(r = 0.70, p = 0.01)$
Maintainability index	$(r = -0.25, p = 0.74)$	$(r = -0.35, p = 0.86)$

D. Discussion

Here, we discuss some of the limitations and threats to validity of our results.

Implementation Limitations. The current implementation of JSNOSE is not able to detect all various ways of object creation in JavaScript. Also it does not deal with various syntax styles of frameworks such as jQuery. For the dynamic analysis part, JSNOSE is dependent on the crawling strategy and execution time, which may affect the accuracy if certain JavaScript files are never loaded in the browser during the execution since the state space of web applications is typically huge. Since JSNOSE is using Rhino to parse the JavaScript code and generate the AST, if there exists a syntax error in a JavaScript file, the code in that file will not be parsed to an AST and thus any potential code smells within that file will

be missed. Note that these are all implementation issues and not related to the design of our approach.

Threats to Validity. A threat to the external validity of our evaluation is with regard to the generalization of the results to other web applications. We acknowledge that more web applications should be evaluated to support the conclusions. To mitigate this threat we selected our experimental objects from different application domains, which exhibit variations in design, size, and functionality.

One threat to the internal validity of our study is related to the metrics and criteria we proposed in Table I. However, we believe these metrics can effectively identify code smells described in Section IV. The designated 10 minutes time for crawling could also be increased to get more accurate results, however, we believe that in most maintenance environments this is acceptable considering frequent code releases. The validation and accuracy analysis performed by manual inspection can be incomplete and inaccurate. We mitigated this threat by focusing on the applications with smaller sets of code smells so that manual comparison could be conducted accurately.

With respect to reliability of our evaluation, JSNOSE and all the web-based systems are publicly available, making the results reproducible.

VII. Conclusions and Future Work

In this paper, we discussed a set of 13 JavaScript code smells and presented a metric-based smell detection technique, which combines static and dynamic analysis of the client-side code. Our approach, implemented in a tool called JSNOSE, can be used by web developers during development and maintenance cycles to spot potential code smells in JavaScript-based applications. The detected smells can be refactored to improve their code quality.

Our empirical evaluation shows that JSNOSE is effective in detecting JavaScript code smells; Our results indicate that lazy object, long method/function, closure smells, coupling between JavaScript, HTML, and CSS, and excessive global variables are the most prevalent code smells. Further, our study indicates that there exists a strong and significant positive correlation between the types of smells and LOC, cyclomatic complexity, and the number of functions and JavaScript files.

For future work, we intend to extend our list of code smells, increase the accuracy of JSNOSE, add smell detection support for jQuery syntax, and design and implement an automated tool for refactoring detected JavaScript code smells.

Acknowledgment: This work was supported by the National Science and Engineering Research Council of Canada (NSERC) through its Strategic Project Grants programme. Amin Milani Fard is also supported by an Alexander Graham Bell Canada Graduate Scholarship (CGS-D) from NSERC.

References

[1] Callback hell: A guide to writing elegant asynchronous JavaScript programs. http://callbackhell.com/.
[2] Checkstyle. http://checkstyle.sourceforge.net/.
[3] Google closure compiler. https://developers.google.com/closure/.
[4] Jslint: The JavaScript code quality tool. http://www.jslint.com/.
[5] Mozilla developer network's JavaScript reference. https://developer.mozilla.org/en-US/docs/JavaScript/Reference.
[6] WARI: Web application resource inspector. http://wari.konem.net.
[7] Y. Crespo, C. López, R. Marticorena, and E. Manso. Language independent metrics support towards refactoring inference. In *9th ECOOP Workshop on QAOOSE*, volume 5, pages 18–29, 2005.
[8] D. Crockford. *JavaScript: the good parts*. O'Reilly Media, Incorporated, 2008.
[9] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
[10] F. Galassi. Refactoring to unobtrusive JavaScript. JavaScript Camp 2009.
[11] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2002.
[12] J. Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.
[13] M. Lanza and R. Marinescu. *Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.
[14] M. Mäntylä, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 381–384. IEEE Computer Society, 2003.
[15] A. Mesbah and S. Mirshokraie. Automated analysis of CSS rules to support style maintenance. In *Proc. International Conference on Software Engineering (ICSE)*, pages 408–418. IEEE Computer Society, 2012.
[16] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.
[17] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.
[18] M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in Java source-code. In *Proc. International Symposium Software Metrics*, pages 15–15. IEEE, 2005.
[19] R. Murphey. JS minty fresh: Identifying and eliminating JavaScript code smells. http://fronteers.nl/congres/2012/sessions/js-minty-fresh-rebecca-murphey.
[20] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen. Detection of embedded code smells in dynamic web applications. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 282–285. ACM, 2012.
[21] V. Özçelik. o2.js JavaScript conventions & best practices. https://github.com/v0lkan/o2.js/blob/master/CONVENTIONS.md.
[22] J. Padolsey. jQuery code smells. http://james.padolsey.com/javascript/jquery-code-smells/.
[23] S. Porto. A plain english guide to JavaScript prototypes. http://sporto.github.com/blog/2013/02/22/a-plain-english-guide-to-javascript-prototypes/.
[24] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proc. of the conf. on Programming language design and implementation (PLDI'10)*, pages 1–12. ACM, 2010.
[25] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *Proc European Conference on Software Maintenance and Reengineering (CSMR)*, pages 30–38. IEEE, 2001.
[26] K. Simpson. Native JavaScript: sync and async. http://blog.getify.com/native-javascript-sync-async.
[27] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In *Proc. European Conference on Software Maintenance and Reengineering (CSMR)*, pages 329–331, 2008.
[28] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 97–106. IEEE Computer Society, 2002.
[29] W3C. Document Object Model (DOM) level 2 events specification. http://www.w3.org/TR/DOM-Level-2-Events/, 13 November 2000.
[30] L. Williams, D. Ho, and S. Heckman. Software metrics in eclipse. http://realsearchgroup.org/SEMaterials/tutorials/metrics/.
[31] N. C. Zakas. Writing efficient JavaScript. In S. Souders, editor, *Even Faster Web Sites*. O'Reilly, 2009.
[32] N. C. Zakas. *Maintainable JavaScript - Writing Readable Code*. O'Reilly, 2012.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

Determining Dynamic Coupling in JavaScript Using Object Type Inference

Jens Nicolay, Carlos Noguera, Coen De Roover, Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
{jnicolay, cnoguera, cderoove, wdmeuter}@vub.ac.be

***Abstract*—Coupling in an object-oriented context is often defined in terms of access to instance variables and methods of other classes. JavaScript, however, lacks static type information and classes, and instead features a flexible object system with prototypal inheritance. In order to determine coupling in JavaScript, we infer object types based on abstract interpretation of a program. Type inference depends on both structure and behavior of objects, and common patterns for expressing classes and modules are supported. We approximate a set of accessed types per function, and classify every access as either local or foreign. Examples demonstrate that our object type inference, together with some additional heuristics concerning property access, enable determining coupling in JavaScript in a meaningful way.**

I. Introduction

JavaScript applications are becoming increasingly complex, and are no longer confined to the browser. As a result, various stakeholders would benefit from tools that support them in assessing the quality of JavaScript code. Extensive research has resulted in quality assessment tools for traditional, class-based, object-oriented languages [1], [2], [3]. JavaScript, in contrast, has received far less attention in this regard. One possible reason is that static reasoning over JavaScript programs is challenging due to the dynamic nature of the language. Static reasoning forms the foundation for advanced tooling.

One important aspect that determines the quality of a program is coupling. It is considered a good software engineering principle to keep coupling low, as this makes code easier to understand and maintain. In this paper, we describe our approach to determining the dynamic coupling of functions to object types. We define dynamic coupling for a function as the set of object types of which that function accesses properties. As JavaScript has no classes nor static types, we infer object types in such a way that we can measure coupling meaningfully, even in the presence of dynamic features and the commonly used class and module patterns. We demonstrate the usefulness of our approach by applying it to the detection of coupling-related code smells.

A. Approach

We start by approximating the set of all property accesses that might occur in a running program. To this end, an abstract interpreter collects tuples that capture the details of each property access (Section II). Next, we compute a type for the receiver (Section III), enabling us to determine the locality of each access based on the type of the base object and that of the `this` binding that is in effect. By collecting this information for all property accesses in a given function, we determine the dynamic coupling for that function (Section IV). The initially computed types are based only on information available in the tuples returned by the interpreter. When considering the entire set of property accesses, we are able to enhance these types by discovering relations between them based on their flow to base expressions (Section V-A) and their interfaces (Section V-B).

978-1-4673-5739-5/13/$31.00 © 2013 IEEE

B. Contributions

- We describe an abstract interpretation required for approximating all property accesses in a running program.
- We examine how to infer types for objects so that module and class patterns are supported.
- We show how to compute dynamic coupling in JavaScript based on approximations of property access and inferred object types.

II. Abstract Interpretation

At the heart of our approach to determining dynamic coupling is an abstract interpreter [4]. An abstract interpreter executes the program using abstract semantics that approximate the concrete semantics of JavaScript program execution. These abstractions simplify the semantic domain that models the runtime, and guarantees finite execution in both time and space. In this paper we define coupling in terms of types inferred from property access within functions. Therefore, our abstract interpreter is configured to collect a set of tuples that contain information about the property accesses it encounters. This set of tuples is then used to infer object types, which in turn determine coupling of functions.

In what follows, we detail the syntactic elements of JavaScript that are relevant to the abstract interpretation, the abstract domain used to analyse property accesses, and the manner in which property access information is collected during interpretation.

A. Syntactic elements

Figure 1 identifies the syntactic elements of JavaScript that are relevant for reasoning over property accesses with the

$n \in \mathsf{Node}$ = finite set of AST nodes
$f \in \mathsf{Fun} \subseteq \mathsf{Node}$ = function expressions and declarations
$v \in \mathsf{Id} \subseteq \mathsf{Node}$ = identifiers
$e \in \mathsf{Exp} \subseteq \mathsf{Node}$ = expressions
$p \in \mathsf{Prop} \subseteq \mathsf{Exp}$ = e_b`.`v (dot) | e_b`[`e_v`]` (bracket)
$t \in \mathsf{This} \subseteq \mathsf{Exp}$ = `this` (this expression)
$u \in \mathsf{Call} \subseteq \mathsf{Exp}$ = $e_f(e_{arg}*)$ (call expressions)
$c \in \mathsf{New} \subseteq \mathsf{Exp}$ = `new` $e_f(e_{arg}*)$ (new expressions)
$l \in \mathsf{ObjLit} \subseteq \mathsf{Exp}$ = $\{\langle v : e_{init}\rangle * \}$ (object literals)
$r \in \mathsf{ArrLit} \subseteq \mathsf{Exp}$ = `[`e_{init} `*` `]` (array literals)

Fig. 1. Overview of relevant syntactic elements of JavaScript.

$$
\begin{aligned}
\mathit{Store} &= \mathit{Addr} \mapsto \mathit{Val} \\
\mathit{Val} &= (\mathit{Prim} \times \wp(\mathit{Addr})) + \mathit{Benv} \\
\mathit{Benv} &= \mathit{Str} \mapsto \wp(\mathit{Addr}) \\
\mathit{Addr} &= \text{finite set of addresses} \\
\mathit{Prim} &= \text{finite set of abstract primitive values} \\
\mathit{Prim} \supseteq \mathit{Str} &= \text{set of abstract names}
\end{aligned}
$$

Fig. 2. The abstract domain used in our static analysis.

purpose of inferring object types and determining coupling. Besides property access itself, we also distinguish between function expressions and declarations, identifiers, call and `new` expressions, and array and object literals. The set Prop contains two kinds of property access nodes. Dot-based accesses identify the accessed property of base expression e_b through the identifier v, while bracket-based accesses identify the accessed property through the value of another indexation expression e_v. In the second case, JavaScript always coerces base expressions into object types and indexation expressions into strings. We define a function $\mathit{base} : \mathsf{Prop} \mapsto \mathsf{Exp}$ to access the base expression of either kind of property access.

B. Abstract domain

The abstract interpreter models the runtime of the target program with the abstract domain shown in Figure 2. Addresses in Addr are references to values and generated whenever the interpreter allocates values. An abstract object in Benv is a one-to-many mapping from names to addresses. Abstract objects are used to represent objects, arrays, closures, and environments. A store in Store maps addresses onto a value. It mimics the heap and is a key component in performing points-to analysis. Values are either first-class values (primitives and object references) or objects.

In the context of object type inference and dynamic coupling, we are especially interested in those AST nodes that create objects. New expressions, array literals, and object literals are sources of object creation that can be directly traced back to specific nodes. Certain method invocations on built-in objects, like for example `Object.create`, also create objects behind the scenes. Function expressions and declarations give rise to function objects, the evaluation of regular expressions results in `RegExp` objects, and so on. Function nodes also indirectly create a prototype object, accessible through the public `prototype` property on the function object created by the same node. Addresses are generated in the interpreter in such a way that there is a link between the address and the node responsible for creating that address, even if the address in question references an implicitly created object. Thus, we define a function $\mathit{node} : \mathit{Addr} \mapsto \mathsf{Node}$ that establishes this between an address and its corresponding node. Note that this does not limit the number of addresses linked to a particular node, so addresses can be generated in a context-sensitive manner for example.

$$\mathit{Acc} = \mathit{Addr} \times \mathit{Addr} \times \mathsf{Prop} \times \mathit{Addr} \times \mathit{Str} \times \mathit{Store}$$

$a_f \in \mathit{Addr}$	Address of the function enclosing the property access.
$a_t \in \mathit{Addr}$	Current address of `this` in the active execution context.
$p \in \mathsf{Prop}$	Property access node.
$a_b \in \mathit{Addr}$	Address of $\mathit{base}(p)$.
$s \in \mathit{Str}$	Name of the referenced property in p, either name of v or the result of evaluating e_v.
$\sigma \in \mathit{Store}$	Current store.

Fig. 3. Elements of the active execution context in Acc, generated for each property access the interpreter evaluates.

C. Collecting property access information

Examining property access in a JavaScript program on a syntactic level only is not sufficient: it usually does not reveal any useful information about the objects and properties that might be involved at runtime. We therefore rely on an abstract interpreter that is instrumented to collect information on all property accesses it evaluates during abstract interpretation of a program. For the moment we will only consider *read* access. In Section V-C we explain how to extend our approach to also include *write* access, and how to deal with accessing undefined properties. Information regarding a particular read property access is recorded in a tuple $(a_f, a_t, p, a_b, s, \sigma) \in \mathit{Acc}$ consisting of the elements detailed in Figure 3. The evaluation of one property access can result in multiple tuples, depending on the number of addresses in the points-to sets of the enclosing function, the value of `this`, and the base expression. How close these and other computed sets in the remainder of this paper are to their actual runtime equivalents, depends on the combination of the dynamicity of the program and the precision of the underlying abstract interpretation.

III. Object Type Inference

Our approach determines coupling dynamically by considering the types of accessed objects. To compute the type of an

object, we define a function iof that takes an object reference and a store, and returns a set of addresses that represents the type for that object reference.

$$iof : Addr \mapsto Store \mapsto \wp(Addr)$$

Using addresses to represent types entails that the type of an object is another object. Below we discuss which objects we consider as types. Using objects as types carries the added benefit of allowing us to map an object to abstract syntax tree locations (cf. II-B), which can give further information to type inference algorithms. Defining the function iof this way gives rise to a type hierarchy in the form of a tree of objects computed by the transitive closure $iof*$.

$$iof* : Addr \mapsto Store \mapsto \wp(\wp(Addr))$$
$$iof*(a_0, \sigma) = \{\{a_0, \ldots, a_n\} \mid \forall i \in 1..n : a_i \in iof(a_{i-1})\}$$

Function iof encapsulates different possible object type inference algorithms. If we only look at a single tuple at a time to infer types, there are two straightforward definitions of this function.

1) *Prototype-based*, where the type of an object is its prototype.
2) *Object-based*, where each object is its own type.

Below we explore these definitions, and conclude that we need to combine them in order to support some of the most common JavaScript patterns.

A. Prototype-based type inference

Every object in JavaScript has an *internal* [[Prototype]] property that points to that object's prototype, or that has the value `null` if there is no prototype. Similarly, function objects have a *regular* `prototype` property. When a function object is invoked as a constructor using the `new` operator, the result is a newly created object of which the [[Prototype]] property is set to the `prototype` property of the constructor at the time of construction. Therefore, for any function `F`, the following holds:

```
Object.getPrototypeOf(new F()) === F.prototype
```

We consider a first approximation to object type inference in which the iof function is defined in terms of the [[Prototype]] property. In abstract terms the value of [[Prototype]] for an object is a (possibly empty) *set* representing an over-approximation of object references to prototype objects. By defining a function $proto : Benv \mapsto \wp(Addr)$ that returns this set of possible prototype addresses, we can define a prototype-based definition of iof_p as follows:

$$iof_p : Addr \mapsto Store \mapsto \wp(Addr)$$
$$iof_p(a, \sigma) = proto(\sigma(a))$$

A prototype chain of a JavaScript object starts at the [[Prototype]] of that object and then traces out subsequent [[Prototype]] links. In JavaScript the prototype chain of an object would be obtained by repeatedly applying the built-in function `Object.getPrototypeOf`, until `null` is encountered. The `instanceof` operator of JavaScript uses the prototype chain to determine types: an object `x` is an instance of constructor `F` if the prototype chain of `x` contains `F.prototype`. For approximating the object type of an object, the transitive closure of iof_p*, will compute a prototype *tree*, represented as a set of all possible [[Prototype]] chains. Function iof_p defines types by referring to prototype objects (`Object.getPrototypeOf`) instead of taking prototype objects to constructors as `instanceof` does.

```
function Circle(x, y, r) {
  this.x = x;
  this.y = y;
  this.r = r;
}

function Point(x, y) {
  Circle.call(this, x, y, 0);
}
Point.prototype =
  Object.create(Circle.prototype);

var p = new Point(90, 90);
p.x
```

Fig. 4. Example program P_1.

Convention: In the code examples that follow we will translate addresses into equivalent source code expressions, i.e. expressions that evaluate to the same addresses. If necessary, the evaluation context for these expressions will be made clear in the example.

Example 1: In the JavaScript program P_1 (Fig. 4) the following holds:

```
Object.getPrototypeOf(p)
  === Point.prototype
⇒ true
Object.getPrototypeOf(Point.prototype)
  === Circle.prototype
⇒ true
Object.getPrototypeOf(Circle.prototype)
  === Object.prototype
⇒ true
Object.getPrototypeOf(Object.prototype)
  === null
⇒ true
```

Therefore, if σ is the store when evaluating property access `p.x`, then $iof_p(\mathtt{p}, \sigma)$ = {`Point.prototype`} and $iof_p*(\mathtt{p}, \sigma)$ = {{`Point.prototype`, `Circle.prototype`, `Object.prototype`}}. □

Example 2: Example program P_2 (Fig. 5) shows the essence of the module pattern in JavaScript. Functions `pub` and `priv` are defined in the local scope of the immediately invoked function expression (IIFE) enclosing the two functions. Because they are confined to the local scope of the IIFE, they are not directly accessible from outside it. The object literal the IIFE returns serves as the interface of the module, so that the access `module.pub` returns function `pub`. Using iof_p, the type computed for the base object in property access `module.pub` is {`Object.prototype`}. The reason is that object literals are evaluated to objects that are created as if by the expression `new Object()`. The type

```
var module = (function () {
    function pub(x) {return Math.sqrt(x)};
    function priv() {return "bar"};
    return {pub:pub};
  })();

function f() {
  print(module.pub(4));
}
f()
```

Fig. 5. Example program P_2.

```
for (var i = 0; i < 1000; i++) {
  var o = {x:i};
  print(o.x);
}
```

Fig. 6. Example program P_3.

of base object `Math` in property access `Math.sqrt` is also {`Object.prototype`}, because the prototype of this object is `Object.prototype` as well. □
Example 2 shows the limits of iof_p when inferring the types of object literals and built-in objects such as `Math`.

B. Object-based type inference

The finest granularity at which we can assign object types, is by assigning each object its own type. Therefore, we define an object-based version of iof_o:

$$iof_o : Addr \mapsto Store \mapsto \wp(Addr)$$
$$iof_o(a, \_) = \{a\}$$

The type hierarchy computed by the transitive closure iof_o* consists of a single chain containing the object address: $iof_o * (a, \_) = \{\{a\}\}$. In a concrete setting, each object can be allocated at a unique address. In our abstract setting, the set of addresses is finite and typically much smaller than the set of concrete addresses. The address allocation strategy used during abstract interpretation will greatly influence to which degree we can distinguish different objects. Although different configurations and implementation strategies may therefore impact the speed and precision (and therefore quality) of computed results, those results should always be sound.

Example 3: Taking the same module definition as in Example 2, but now using object-based type inference, we compute {`module`} as type of the base object in property access `module.pub`, and {`Math`} as type of the base object in property access `Math.sqrt`.

Example 4:

Consider the example program P_3 (Fig. 6) consisting of a `for` loop: A concrete interpretation will allocate 1000 different objects, but abstract interpretation might distinguish between far less *abstract* objects, usually depending on how quickly it reaches a fixpoint (which in turn might depend on things like context-sensitivity, strong updating, widening, etc.). For example, our abstract interpreter is equipped with a "k last call sites" strategy for distinguishing between different contexts (Section VII), and entering a loop body is considered to be equivalent to a function call in this regard. With $k = 0$, i.e with context-sensitivity turned off, only one address is used for allocating objects resulting from evaluating the object literal, and the property access is evaluated three times before a fixpoint is reached. We therefore have $iof_o(\mathtt{o}, \_) = \{\{\mathtt{x:i}\}\}$ for all property accesses.

When $k = 5$, the abstract interpreter returns seven tuples and generates three distinct addresses for allocating the object literal. In this case we compute three different types for `o` instead of one single type as was the case with $k = 0$.

In both cases the computed types are sound, altough in the latter case they are more precise. □

C. Combining prototypes and object-based inference

Examining the results obtained by typing base objects using either iof_p or iof_o, we observe the following:

- Example 1 shows that prototype-based object type inference works well when objects are explicitly created through constructors using `new`, and we are dealing with "instance" data. However, when accessing properties on non-instance objects, like the property access `Math.sqrt` in Example 2, prototype-based inference is not useful. Example 2 also demonstrates that prototype-based inference is unable to distinguish between accesses to different modules. Objects created from object literals are considered to be instances of the standard `Object` constructor, greatly diminishing the usefulness of the type inference in these cases.
- Examples 3 and 4 shows that object-based object type inference is able to distinguish between different object literals, and, as a consequence, different modules. On the other hand, as in Example 4, object-based inference may be overkill in situations where objects from one or more object literals intuitively represent different instances of the same type.

A better definition for iof therefore would be one that differentiates objects based on their creation. If an object comes into existence as a result of evaluating a constructor invocation (New), an array literal (ArrLit), or the invocation of a built-in method like for example `Object.create` (Call), then the type of that object is its [[Prototype]]. For all other objects, we take the address of those objects as their type. Built-in objects (like `Math` and the global object) will have unique addresses, as will objects created at different source code locations[1].

$$iof_n : Addr \mapsto Store \mapsto \wp(Addr)$$
$$iof_n(a, \sigma) = \begin{cases} iof_p(a, \sigma) & \text{if } n = node(a) \\ & \wedge n \in \mathsf{New} \cup \mathsf{ArrLit} \cup \mathsf{Call} \\ iof_o(a, \sigma) & \text{else} \end{cases}$$

[1]Up to a certain limit because the set of addresses is finite.

IV. Computing coupling

As we have previously explained (cf. I), we are interested in statically assessing the dynamic coupling of JavaScript functions. Thus, we require, for each property access appearing in a given function, the type of the base object (represented by its address $Addr$), and a constant indicating whether that type is local or foreign ($\{L, F\}$) at the time of access. The function *coupling* computes typed accesses specific to a function, starting from the set of tuples collected by the abstract interpreter.

$$coupling : Addr \mapsto \wp(Acc) \mapsto (Addr \times \{L, F\})$$

Whether a type is local or foreign is decided in function of the value of `this` in the function at the time of access. If the property is accessed on a type in the type hierarchy of `this`, then the type is marked as local (L), otherwise, the type is deemed foreign (F).

Our approach can be instantiated with any definition of the function iof that returns a set of addresses representing objects. In what follows, we will use iof_n and its transitive closure from the previous section. Concretely, when the function iof_n* is applied to an address a_t representing `this`, we obtain all possible [[Prototype]] chains consisting of types that are considered local to the active function invocation. Then, the function iof_n is used to compute a type a_o for each object appearing as a base object in a property access. The base object type is local when it is a member of a chain returned by the application of iof_n*, else it is foreign.

$$locality : Addr \mapsto Addr \mapsto Store \mapsto \wp(\{L, F\})$$

$$locality(a_o, a_t, \sigma) = \bigcup_{\vec{a} \in iof_n*(a_t,\sigma)} \begin{cases} L & \text{if } a_o \in \vec{a} \\ F & \text{else} \end{cases}$$

Function *locality* ensures that every access to `this` in functions is always considered to be a local access, since in that case $a_o = a_t$.

We then define a helper function that will filter the information on property access collected in a set $\mathcal{T}$, keeping only tuples pertaining to the specified function.

$$filterFun : \mathsf{Fun} \mapsto \wp(Acc) \mapsto \wp(Acc)$$
$$filterFun(f, \mathcal{T}) = \{(a_f, \_, \_, \_, \_, \_) \in Acc \mid node(a_f) = f\}$$

Filtering the collected coupling information, and applying iof and *locality*, we can determine the dynamic coupling for a function f as follows:

$$coupling(f, \mathcal{T}) = \{(a, \ell) \mid a \in iof_n(a_b, \sigma) \wedge \ell \in locality(a, a_t, \sigma) \wedge (a_f, a_t, p, a_b, s, \sigma) \in filterFun(f, \mathcal{T})\}$$

Example 5: The code in Figure 7 defines two types, `Circle` and `Point`. In the module, a `Point` is made to be a `Circle` with radius 0.

Suppose we have the following client application:

```
function Circle(x, y, r) { ... }

Circle.MP = 0.5;

Circle.prototype.midpoint =
  function (c) {
    return new Point((this.x+c.x) * Circle.MP,
      (this.y+c.y) * Circle.MP);
  }

function Point(x, y) {
  Circle.call(this, x, y, 0);
}
Point.prototype = Object.create(Circle.prototype);
```

Fig. 7. Example program P_4.

```
var c = new Circle(10, 20, 50);
var p = new Point(90, 90);
p.midpoint(c)
```

The abstract interpreter returns the following tuples regarding property access in method `midpoint`:

(midpoint, p, this.x, p, "x", σ)
(midpoint, p, c.x, c, "x", σ)
(midpoint, p, Circle.MP, Circle, "MP", σ)
(midpoint, p, this.y, p, "y", σ)
(midpoint, p, c.y, p, "y", σ)

We compute the following coupling for `midpoint` using iof_n:

(Point.prototype, L)
(Circle.prototype, L)
(Circle, F)

The coupling to `Point.Prototype` is generated by accesses on `this` bound to an instance of `Point`, and accesses of `this` are local by definition. Access to parameter `c` inside method `midpoint` generates coupling to `Circle`, which is also local since `Circle.prototype` is a member of all chains in the type hierarchy iof_n * (p). The access `Circle.MP` is considered to be a foreign coupling to `Circle`, since iof_n(c) = `Circle`, which is not a member of the type hierarchy of `this`. The constructor invocation `new Point(...)` does not generate any coupling, since `Point` is looked up in the lexical scope of function `midpoint`, and we do not track coupling to environment records[2]. □

Example 6: We again start with the code in Figure 7, but now define the following client program:

```
var c2 = {x:10, y:20, r:50};
var p = new Point(90, 90);
p.midpoint(cl);
```

The difference between this client program and the one in Example 5 is that we don't create a circle object using the constructor `Circle`, but instead use an object literal that

[2]We are deliberately glossing over some details here, like the top-level environment being the global object in JavaScript, and the `with` statement blurring the line between objects and environment records, but in our approach *only* member access (at the syntactical level) generates dynamic coupling.

defines the same properties. The abstract interpreter returns the same tuples regarding property access, but now we compute the following coupling for `midpoint` using iof_n:

$$(\texttt{Point.prototype}, L)$$
$$(\texttt{c2}, F)$$
$$(\texttt{Circle}, F)$$

Access to parameter `c` inside method `midpoint` generates coupling to type `c2`, which is foreign since it is not a member of the type hierarchy $iof_n * (\texttt{p})$. □

V. Enhancing Object Types

Having explained how we calculate the types to which a JavaScript function is coupled to, we now explore how to enhance the typing information produced by our object type inferencing for the purpose of coupling assessment. Previous definitions of iof (cf. III) are derived from looking at a single tuple generated by the abstract interpretation of the program, while in what follows, we consider the whole set of tuples. We propose two ways of relating equivalent object types, the first one relates object types by looking at what instances are used together (i.e., flow to the same base objects in a property access); while the second one relates object types by considering which property names are accessed together (i.e., interface types). Finally, as type inference is calculated using property reads alone, we discuss what further information can be obtained by including property write operations in the analysis.

A. Flow of types

The definition of iof_n solves some of the problems we observed at the beginning of Section III-C, but it still does not adequately deal with the situation in which objects resulting from the evaluation of one or more object literals intuitively represent different instances of the same type. One way of relating types is by looking at where instances end up being used as base object. To capture this notion, we first define the function $flowsTo$ that for every type computed using iof_n, yields all the base expression nodes that type flows to. Unlike the previous type inferences, we now take the entire set of tuples returned by the abstract interpreter into consideration.

$$flowsTo : Addr \mapsto \wp(Acc) \mapsto \wp(\mathsf{Exp})$$
$$flowsTo(a, \mathcal{T}) = \{e_b \mid a \in iof_n(a_b, \sigma) \wedge (\_, \_, p, a_b, \_, \sigma) \in \mathcal{T} \wedge base(p) = e_b\}$$

Then we can define a reflexive and symmetric relation $\approx_f$ stating that "two types are equivalent iff they flow to the same base expression node".

$$a_1 \approx_f a_2 \iff flowsTo(a_1, \mathcal{T}) \cap flowsTo(a_2, \mathcal{T}) \neq \varnothing$$

The transitive closure of $\approx_f^*$ partitions the set of types in the range of iof_n into equivalence classes. All types returned by a single application of iof_n are equivalent by this definition.

Example 7: We compute the following coupling for function `area` in example program P_5 (Fig. 8).

```
function Circle(x, y, r) { ... }

Circle.prototype.circumference =
  function () {
    return 2 * Math_a.PI * this_a.r;
  }

function area(c) {
  return Math_b.PI * c_a.r * c_b.r;
}

var c1 = {x:10, y:20, r:30};
var c2 = new Circle(30, -5, 10);
var c3 = new Circle(50, 50, 20);
area(c1)
area(c2)
c2.circumference()
area(c3)
```

Fig. 8. Example program P_5.

$$(\texttt{Math}, F)$$
$$(\texttt{c1}, F)$$
$$(\texttt{Circle.prototype}, F)$$

And for method `circumference` we obtain the following coupling:

$$(\texttt{Math}, F)$$
$$(\texttt{Circle.prototype}, L)$$

For the set of tuples $\mathcal{T}$, we have the following flow of types:

$$flowsTo(\texttt{Math}, \mathcal{T}) = \{\texttt{Math}_a, \texttt{Math}_b\}$$
$$flowsTo(\texttt{c1}, \mathcal{T}) = \{\texttt{c}_a, \texttt{c}_b\}$$
$$flowsTo(\texttt{Circle.prototype}, \mathcal{T}) = \{\texttt{c}_a, \texttt{c}_b, \texttt{this}_a\}$$

Consequently, the equivalence relation $\approx_f^*$ partitions the set of computed types as follows:

$$\{\{\texttt{Math}\}, \{\texttt{c1}, \texttt{Circle.prototype}\}\}$$

This means that types `c1` and `Circle.prototype` are equivalent under $\approx_f^*$, and we could substitute the equivalence class for the computed types when determining coupling. If we denote the equivalence class of type a as $[a]$, we have:

$$[\texttt{Math}] = \{\texttt{Math}\}$$
$$[\texttt{c1}] = [\texttt{Circle.prototype}] = \{\texttt{c1}, \texttt{Circle.prototype}\}$$

This however can have consequences for the locality of the access. The equivalence class of prototype `Circle.prototype` in our example also contains an object-based type `c1`, the latter which does not belong to the hierarchy of receiver `c2`. If the type hierarchy of an equivalence class is the join of the type hierarchies computed for every type in that equivalence class, the locality for the coupling to `Circle.prototype` both $\{L, F\}$. This also breaks the invariant that an access to `this` inside a method is a purely local access. One solution could be to replace types by their equivalence class, but keep the locality computed using the original type. Also, if an equivalence class contains a prototype, one could assume that "the programmer knows

best" and that these types, most often constructed by `new`, should somehow take precedence over other types in that equivalence class. We regard clarifying these issues as future work.

Relating types by their flow may make types too coarse-grained. Two types may flow to the same base expression(s), thereby sharing a very small part of their interface, but actually may be unrelated for all other intents, including in the context of coupling. We'll have more to say on this when we discuss interface types next.

B. Interface types

Just like $flowsTo$ induces a partitioning on the types computed by iof_n based on the flow of types, we can also partition the set of types based on the properties that are accessed. Intuitively, this follows the notion of duck typing. We define the $interface$ of a type as the set of names that are accessed on it. As with $flowsTo$, we consider the entire set of property access information returned by abstract interpretation.

$$interface : Addr \mapsto \wp(Acc) \mapsto \wp(Str)$$
$$interface(a, \mathcal{T}) = \{s \mid a \in iof_n(a_b, \sigma) \wedge (\_, \_, \_, a_b, s, \sigma) \in \mathcal{T}\}$$

We define a reflexive and symmetric relation $\approx_i$ stating that "two types are equivalent iff they have the same interface".

$$a_1 \approx_i a_2 \iff interface(a_1, \mathcal{T}) = interface(a_2, \mathcal{T})$$

The transitive closure of $\approx_i^*$ partitions the set of types in the range of iof_n into equivalence classes, and here too all types returned by a single application of iof_n are equivalent by this definition.

Relating types only by interface is usually not sufficient, since it may relate types that should be considered unrelated. Two types can have the same interface, say $\{$`"x"`, `"y"`, `"z"`$\}$, but one type might express a 3D coordinate, while the other represents the code for a 3-cipher lock. As was previously the case with the definition of iof_n, where we combined prototype inference with object-based type inference, here too a combination of flow information and interfaces may be the best approach.

Example 8: In program P_6 (Fig. 9) the relation $\approx_f^*$ induces the following partition on the set of types:

$$\{\{\texttt{Math}\}, \{\texttt{Label.prototype}, \texttt{Circle.prototype}\}\}$$

Types `Label.prototype` and `Circle.prototype` are related because they flow to function `hash`. However, intuitively they represent different types and have different interfaces, even though they share $\{$`"x"`, `"y"`$\}$ in this respect. Based on interfaces, we may split the equivalence class based on flow, thereby obtaining a set of unrelated types again:

$$\{\{\texttt{Math}\}, \{\texttt{Label.prototype}\}, \{\texttt{Circle.prototype}\}\}$$

```
function Label(x, y, n) {
  this.x = x;
  this.y = y;
  this.n = n;
}

function Circle(x, y, r) { ... }

function hash(p) {
  return ((p.x + p.y)*(p.x + p.y + 1)/2) + p.y;
}

Circle.prototype.circumference =
  function () {
    return 2 * Math.PI * this.r;
  }

function display(label) {
  return label.x + label.y + label.n;
}

var c = new Circle(30, -5, 10);
var l =  {x: 50, y:50, n:"Destination"};
hash(c);
c.circumference();
hash(l);
display(l);
```

Fig. 9. Example program P_6.

C. Writing properties

So far we have only addressed coupling as a consequence of reading properties. When considering writing properties' impact on coupling, we observe that:

1) Writing properties can extend or modify an object. When the property is not present, the property is added, else the value of the property is changed.
2) Writing of properties does not involve prototype lookup. The base object is the object that is extended or modified.

In order to properly reason about property writes, the abstract interpreter must be extended to keep track of extra information for each property access. We therefore extend the definition of our tuples in Acc by adding a tag indicating whether a property write (W) or read (R) occurred.

$$Acc_{RW} = Addr \times Addr \times \mathsf{Prop} \times \{R, W\} \times Addr \times Str \times Store$$

If the abstract interpreter takes a snapshot of the store before the property access itself is effectuated, it becomes possible to determine whether the accessed property exists or not. That way we can discern between the following scenarios with regards to the coupling induced by the property access:

1) Read of an existing property: this is the scenario that we have considered so far in the paper.
2) Read of a non-existing property, or property with an `undefined` value: programmers sometimes use the presence or absence of a certain property as a type test. It may be argued that reading an undefined property does not generate coupling by itself. Programmers might try to read undefined properties as a kind of "instance of"

check, in which case the check is normally followed by uses of the object, which in turn will be detected as coupling.

3) Update of an existing property: this scenario generates coupling in the same way as reading an existing property, thus it is detected in the same manner as scenario 1 above.
4) Adding a property: again we argue that this does not generate coupling on the level of types, since adding properties always "works", regardless of the inferred type of the receiving object.

VI. Applications

We now present how, using the coupling detection approach we have outlined above we can detect the Feature Envy bad smell as defined by Lanza, Marinescu and Ducasse in [5], adapted to functions in JavaScript.

A. Detecting Feature Envy

Feature Envy is a bad smell related to coupling. Informally, it can be described as "a design disharmony [that] refers to methods that seem more interested in the data of other classes than that of their own class" [5]. To detect feature envious functions, we implement the three metrics that Lanza et. al. use to detect Feature Envy: Access To Foreign Data ($atfd$), Locality of Attribute Access (laa), and Foreign Data Providers (fdp). Given those functions, they propose the following detection strategy for Feature Envy, given a function f:

$$\begin{aligned} featureEnvy(f) \iff & \; atfd(f) > FEW \\ & \wedge \; laa(f) < \frac{1}{3} \\ & \wedge \; fdp(f) \leq FEW \end{aligned}$$

where FEW stands for a small positive integer like 2 or 3.

We use the set of tuples computed by function $coupling$ from Section IV to distinguish between different data providers, and between foreign and local attributes, Note however, that a type is returned only once by the $coupling$ function regardless the amount of times that type is referred to in the function. When detecting feature envy, this distinction becomes relevant. In order to approximate the different accesses to the same type, we include the name of the accessed property s in the coupling information computed by $coupling_s$:

$$\begin{aligned} coupling_s &: Addr \mapsto \wp(Acc) \mapsto (Addr \times Str \times \{L, F\}) \\ coupling_s(f, \mathcal{T}) &= \{(a, s, \ell) \\ &\quad \mid \tau \in iof(a_b, \sigma) \wedge \ell \in locality(a, a_t, \sigma) \\ &\quad \wedge (a_f, a_t, p, a_b, s, \sigma) \in filterFun(f, \mathcal{T})\} \end{aligned}$$

B. Calculating the metrics

It is straightforward to compute the proposed coupling metrics using the results from function $coupling_s$. In what follows, $\#S$ denotes cardinality of S.

Function atd counts all distinct property accesses contained in function f, given a set of tuples $\mathcal{T}$.

$$\begin{aligned} atd &: \mathsf{Fun} \mapsto \wp(Acc) \mapsto \mathsf{Int} \\ atd(f, \mathcal{T}) &= \#coupling_s(f, \mathcal{T}) \end{aligned}$$

Function $atfd$ only counts the foreign property accesses.

$$\begin{aligned} atfd &: \mathsf{Fun} \mapsto \wp(Acc) \mapsto \mathsf{Int} \\ atfd(f, \mathcal{T}) &= \#\{(\_, \_, F) \in coupling_s(f, \mathcal{T})\} \end{aligned}$$

Function laa is the ratio of local property accesses to total number of property accesses.

$$\begin{aligned} laa &: \mathsf{Fun} \mapsto \wp(Acc) \mapsto \mathsf{Int} \\ laa(f, \mathcal{T}) &= \frac{atd(f, \mathcal{T}) - atfd(f, \mathcal{T})}{atd(f, \mathcal{T})} \end{aligned}$$

Finally, the function fdp counts the number of distinct foreign types that are accessed.

$$\begin{aligned} fdp &: \mathsf{Fun} \mapsto \wp(Acc) \mapsto \mathsf{Int} \\ fdp(f, \mathcal{T}) &= \#\{a \mid (a, \_, F) \in coupling_s(f, \mathcal{T})\} \end{aligned}$$

To illustrate how feature envy can be detected, consider the following example.

Example 9: Suppose we have the following client program (identical to the one in Example 5) that uses the code in Figure 7.

```
var c = new Circle(10, 20, 50);
var p = new Point(90, 90);
p.midpoint(c);
```

Using function $coupling_s$ and a set of property access tuples $\mathcal{T}$, we compute the following dynamic coupling for method `midpoint`:

$$\begin{aligned} &(\texttt{Point.prototype}, \texttt{"x"}, L) \\ &(\texttt{Point.prototype}, \texttt{"y"}, L) \\ &(\texttt{Circle.prototype}, \texttt{"x"}, L) \\ &(\texttt{Circle.prototype}, \texttt{"y"}, L) \\ &(\texttt{Circle}, \texttt{"MP"}, F) \end{aligned}$$

The set of foreign data providers is $\{\texttt{Circle}\}$ being accessed once. The remaining types are local, with in total 4 distinct data accesses. The metrics would then be calculates as:

$$\begin{aligned} atfd(\texttt{midpoint}, \mathcal{T}) &= 1 \\ laa(\texttt{midpoint}, \mathcal{T}) &= \frac{4}{5} \\ fdp(\texttt{midpoint}, \mathcal{T}) &= 1 \end{aligned}$$

From this we conclude that method `midpoint` does not exhibit Feature Envy.

C. Dealing with arrays

As explained in Section VI-A, function $coupling_s$ takes the name of the accessed properties into account in its output. JavaScript, unlike Java for example, treats indexing into an array as any other property access. A property name s is an array index if it is the string representation of an integer between 0 and $2^{32} - 1$ [6]. This can have undesirable consequences,

since functions accessing arrays can have very strong coupling to array objects if every array access is counted as a distinct property access.

To address this, we could choose to disregard array index access altogether or treat all array indexing for a particular base array object as one distinct access. Instead, we define a predicate function idx that discriminates between property names that are indexes and those that are not. If the name of the access is computed, loss of precision in the abstract domain can lead to a situation where the concrete representation of an abstract name can be both an array index name and not. Instead of reflecting this by returning a set of booleans, we simplify and idx returns `false` in this situation.

$$idx : Str \mapsto \mathsf{Bool}$$

The behavior of generic JavaScript methods on `Array.prototype` assume that any object accessed using array indexes is in fact array-like. We therefore do not check in any way whether the base object actually is of type `Array.prototype`. This has as an advantage that we can identify array index access on the level of tuples in Acc returned by the abstract interpreter. We can then remove these accesses, or we can define a function $mapIdx$ that maps the property name of every index access onto a unique global symbol IDX so that array indexing is reflected in the metrics at most once for every base object.

$$\begin{aligned} mapIdx &: \wp(Acc) \mapsto \wp(Acc) \\ mapIdx(\mathcal{T}) &= \{(a_f, a_t, p, a_b, s', \sigma) \\ &\quad \mid (a_f, a_t, p, a_b, s, \sigma) \in \mathcal{T} \\ &\quad \wedge s' = \begin{cases} IDX & \text{if } idx(s) \\ s & \text{else} \end{cases} \end{aligned}$$

VII. Implementation

We implemented our approach on top of a generic abstract interpreter called JIPDA, which was influenced by work of Might et al. [7]. JIPDA is capable of handling a large subset of JavaScript semantics, and allows for a large range of configuration options, allowing us to plug-in the abstract domain depicted in Fig. 2 which generates addresses as described in Section II-B. To generate and collect the tuples central to our approach, we instrumented the evaluator by overriding the two methods that deal with member expressions: one method for member expressions that are in operator position, and the other method for all other cases. To determine the enclosing function, we implemented a function that walks the stack to find the most recent invocation.

Our current implementation computes the functions that calculate coupling ($coupling$ and $coupling_s$), and the $flowsTo$ and $interface$ relations defined in Section V. We consider the development of the algorithms that enhance the type inference based on these partitions the subject of future work.

The generic abstract interpreter, its configuration, and the code for implementing our approach, are written in JavaScript. The implementation is publicly available on the project's webpage (`https://code.google.com/p/jipda/`); the configuration of the abstract interpreter and the supporting code used in this paper can be found in folder `instances/mod`.

Limitations: Most limitations in the implementation of our approach are a direct consequence of limitations in the underlying abstract interpreter.

Some of these limitations are inherent to abstract interpretation itself. By using overapproximation, we are sure that we do not exclude any objects and relations between objects that may exist at runtime, but false positives (computed results that will never actually occur in a running program) will diminish the usefulness of the type inference. Therefore, the abstract interpreter that underlies an implementation of our approach should be precise enough to be useful in practice, while being able to compute results in an acceptable amount of space and time, where "useful" and "acceptable" depend on the context. Such a good trade-off between speed and precision is difficult to achieve.

Our implementation of the abstract interpreter supports a set of ECMAScript 5 features [6] large enough, and models this with sufficient precision, to be able to experiment with interesting examples. However, we are currently not able to handle real-world JavaScript frameworks and applications. The main shortcoming is that many global objects and primitives are not yet modeled in the abstract domain. Another limitation is that we do not yet support getter/setter properties and strict mode. These limitations must be addressed before we can attempt to validate the usefulness of our approach on real-life JavaScript programs.

VIII. Related Work

A. Static and dynamic coupling metrics

We have found no previous work that concerns itself with coupling assessment of JavaScript functions, as most existing work has been carried out in the context of object-oriented systems with a static type-system. Within this body of work, two main approaches have been explored: those that rely on the structure of the *source code* (static) and those that rely on (dynamic) *runtime* information, of which the former have received most of the attention in literature. Most of them measure the manner in which classes are linked to each other by observing method invocations or attribute references. This information is readably available in statically-typed languages such as Java or C++. In more dynamic languages as is the case with JavaScript, calculating static coupling metrics is more difficult since obtaining the receiver type of a method invocation requires static analyses. This difficulty is further compounded with the fact that no single way of implementing classes is imposed by JavaScript, so the metric calculation must accommodate for different patterns used to construct objects.

Dynamic coupling metrics, first introduced by Yacoub et al. [8], measure the degree of interaction between objects from a dynamic rather than static context. Beszedes et al. [9] Dynamic Function Coupling (DFC) considers two functions

as coupled if their behaviors are "close" to each other. This is in contrast to our approach, where we consider the coupling between functions and (local or foreign) types. Hassoun et al.'s Dynamic Coupling Measure (DCM) considers the coupling between objects as varying in time [10]. Parallels can be drawn between DCM's time-variance and the set of tuples generated by our abstract interpreter. Both DFC and DCM are metrics calculated from actual runs of a program. The work of Harman et. al. [11] is the only instance we have found that uses static analysis to calculate coupling, using program slicing to measure the coupling of a system. The intuition behind their work is that code fragments that share a slice are coupled.

B. Abstract Interpretation and type inference of JavaScript

Hackett and Guo developed a fast type inference for JavaScript that uses abstract interpretation [12]. Similar to our approach, they assign types to objects according to their prototype, except for plain `Object` and `Array` objects, which have the same type if they were allocated at the same soure location. They do not attempt to relate types based on flow properties or their interfaces, as we do. Logozzo and Venter perform atomic type analysis based on abstract interpretation [13]. Their focus is not on objects, but instead on numerical analysis and domains with the aim of program optimization. Anderson et al. give an operational semantics and static type system using structural types over an idealized version of JavaScript [14]. They consider subtyping for objects based on *implementation* (a subtype must contain all members of its supertype), whereas we formulated equivalence between types based on having the same *interface*. Jensen et al. describe a type analysis for JavaScript based on abstract interpretation, with the goal of checking for the absence of common programming errors, and to provide type information for program comprehension [15]. The abstract domain for TAJS is more complex than ours and models all possible primitive data types, while our approach focuses on object types. Static analysis performed for the Self language is interesting in the context of static analysis for JavaScript, because Self influenced the design of JavaScript. Agesen et al. designed and implemented a constraint-based type inference algorithm for Self, based on implementation types, to guarantee the safety of message sends [16].

IX. Conclusion

We have demonstrated that it is possible to statically determine dynamic coupling of JavaScript functions to object types in a manner that is useful. We also showed that our dynamic coupling can be used to compute metrics with the aim of detecting bad smells related to coupling.

Object type inference is challenging for a dynamic language without classes and static typing. Manual code inspection and even simple AST analysis do not suffice. Value and control flow need to be tracked simultaneously for non-trivial JavaScript programs. Our object type inference therefore is based on information collected by an abstract interpreter that approximates run-time objects and property accesses with sufficient precision. The actual object type inference already takes common JavaScript patterns for classes and modules into account. Relating types based on flow and interfaces are promising ideas to further enhance the type inference, but they need to be further explored to become useful in practice.

Acknowledgments

Jens Nicolay is funded by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen). Coen De Roover is funded by the *Cha-Q* project also sponsored by IWT Vlaanderen. Carlos Noguera is funded by the AIRCO project of the "Fonds Wetenschappelijk Onderzoek".

References

[1] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *Software, IEEE*, vol. 25, no. 5, pp. 22–29, 2008.

[2] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. IEEE, 2004, pp. 245–256.

[3] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, 2010.

[4] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[5] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[6] ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*, 5th ed., June 2011. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[7] M. Might and T. Prabhu, "Interprocedural dependence analysis of higher-order programs via stack reachability," in *Proceedings of the 2009 Workshop on Scheme and Functional Programming, Boston, Massachussetts, USA*, 2009.

[8] S. M. Yacoub, T. Robinson, and H. H. Ammar, "Dynamic metrics for object oriented design," in *Proc. International Symposium on Software Metrics*, 1999, pp. 50–58.

[9] A. Beszedes, T. Gergely, S. Farago, T. Gymothy, and F. Fischer, "The dynamic function coupling metric and its use in software evolution," in *Proc. European Conference on Software Maintenance and Reengineering*, 2007, pp. 103–112.

[10] Y. Hassoun, R. Johnson, and S. Counsell, "A dynamic runtime coupling metric for meta-level architectures," in *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*. IEEE, 2004, pp. 339–346.

[11] M. Harman, M. Okunlawon, B. Sivagurunathan, and S. Danicic, "Slice-based measurement of coupling," in *IEEE/ACM ICSE workshop on Process Modeling and Empirical Studies of Software Evolution*, Boston, Massachsetts, 1997, pp. 28–32.

[12] B. Hackett and S.-y. Guo, "Fast and precise hybrid type inference for javascript," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, 2012, pp. 239–250.

[13] F. Logozzo and H. Venter, "Rata: rapid atomic type analysis by abstract interpretation–application to javascript optimization," in *Compiler Construction*. Springer, 2010, pp. 66–83.

[14] C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for javascript," in *ECOOP 2005-Object-Oriented Programming*. Springer, 2005, pp. 428–452.

[15] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *Static Analysis*. Springer, 2009, pp. 238–255.

[16] O. Agesen, J. Palsberg, and M. I. Schwartzbach, *Type inference of SELF*. Springer, 1993.

CodeMetropolis – code visualisation in MineCraft

Gergő Balogh and Árpád Beszédes

Department of Software Engineering
University of Szeged
Hungary
{geryxyz, beszedes}@inf.u-szeged.hu

***Abstract*—Data visualisation with high expressive power plays an important role in code comprehension. Recent visualization tools try to fulfil the expectations of the users and use various analogies. For example, in an architectural metaphor, each class is represented by a building. Buildings are grouped into districts according to the structure of the namespaces. We think that these unique ways of code representation have great potential, but in our opinion they use very simple graphical techniques (shapes, figures, low resolution) to visualize the structure of the source code.**

On the other hand, computer games use high quality graphic and good expressive power. A good example is Minecraft, a popular role playing game with great extensibility and interactivity from another (third party) software. It supports both high definition, photo-realistic textures and long range 3D scene displaying.

Our main contribution is to connect data visualisation with high end-user graphics capabilities. To achieve this, a conversion tool was implemented. It processes the basic source code metrics as input and generates a Minecraft world with buildings, districts, and gardens. The tool is in the prototype state, but it can be used to investigate the possibilities of this kind of data visualisation.

I. Introduction

Software systems could reach virtually infinite complexity by their nature. In theory, there is no limit of control flow embedding, or the number of methods, attributes, and other source code elements. In practice, these are bound to the computational power, time and storage capacities. To comprehend these systems, developers have to construct a detailed mental image. These images are gradually built during the implementation of the system.

Often, these mental images are realised as physical graphics with the aid of data visualisation software. For example, different kinds of charts are used that emphasize the difference among various measurable quantities of the source code, or UML diagrams which are able to visualise complex relations and connections among various entities in the system.

A. Classical visualisation

People are different, each of them having their own point of views. They use various "tools" to comprehend the world. Some of them need numbers, others use abstract formulas, but most of them need to see to visualize the information as colours, shapes, and figures. To fulfil the expectations of people, a lot of data visualization techniques and tools were designed and implemented. It exceeds the purpose of this article to exhaustedly evaluate these techniques and tools, but in our opinion traditional visualisation tools like Rigi [11], sv3D [5] and SHriMP Views [8] are built on innovative ideas but often it is difficult to interact with them, and they usually fall behind in terms of graphics from today's computer games, for instance.

978-1-4673-5739-5/13/$31.00

B. About CodeCity and EvoSpace

The most closely related approaches to our tool are CodeCity [10] and EvoSpace [4] which use the analogy of skyscrapers in a city. CodeCity simplifies the design of the buildings to a box with height, width, and colour. The quantitative properties of the source code – called metrics – are represented with these attributes. In particular, each building represents a class where height shows the number of methods, width shows the number of attributes, and color shows the type of the class. The buildings are grouped into districts as classes are tied together into namespaces. The diagram itself resembles to a 3D barchart with grouping. EvoSpace uses this analogy in a more sophisticated way. The buildings have two states: closed – when the user can see the large scale properties like width and height, and open – when we are able to examine the low, small scale structure of the classes, see the developers and their connections. It also provides visual entity tagging and quick navigation via the connections and on a small overview map.

Despite their appealing appearance and great potential in general, these tools still use relatively low fidelity graphics compared to today's most advanced computer games. In this paper we introduce our approach for visualising source code using the same metaphor but employing a sophisticated game engine called Minecraft.

C. About Minecraft

Minecraft [6] is a popular role-playing game. It is written in Java language and uses OpenGL graphical engine to display the scenes. Both of these technologies are widely supported on major platforms. It is distributed both as free and as commercial software with support. There are several binary formats used to describe the game scene – called world – which are either open standards or free formats.

The game itself does not have a strict game-flow. Its main focus is creativity and the joy of creation. Only the available computation power and the storage capacity can limit the fantasy of the player.

Figure 1. JUnit project visualized by CodeMetropolis

The main concept in the game is the block. It is a box with about one meter long sides, compared to the player. Almost everything is built up of it, so the whole World is a 3D matrix filled with blocks of various types. The player can collect the blocks, create (craft) new ones and interact with them. The game is similar to a virtual Lego with infinite playground and an infinite number of building blocks.

The player has its own backpack which can be filled with blocks and tools. The tools are used to accomplish or speed up various tasks like mining. Every tool has its own properties. There are some common materials like dirt, stone and sand, and some special blocks have more sophisticated purposes, for example cheast, which can store items, and various electronic (red wire) devices like pressure sensitive plates, buttons, and switches.

Due to its extensibility, its simple yet sophisticated functions, and its rich palette of possibilities Minecraft can display complex structures with a low overhead.

II. Data visualisation in CodeMetropolis

CodeMetropolis is a command line tool written in C#. It takes the output graph of Columbus Tool [1] and creates a Minecraft world from it. The output is given with a unique binary format, but the related tools and the format itself are under development and not yet published. For these reasons we could not give a detailed specification of the output format. The world uses the metropolis metaphor, which means that the source code metrics are represented with the various properties of the different kinds of buildings. Figure 1 shows an example world.

The representation has two main levels. On the data level, each entity has its own property set – for example metrics. In the current version, these are loaded from the previously mentioned graph, but we plan to support other data sources, for example XML files. These data are displayed on the metaphor level. All buildings in the metropolis belong to this. The buildings and the world itself has a couple of attributes which control visual appearance. The properties are mapped to the attributes in order to visualise the data. However, in the current version this mapping is hardcoded, the further versions will support customisation with a sophisticated mapping language.

A. Considered metrics and properties

As mentioned before, source code metrics were used to express the various properties of the system. Our source code analysis toolset, Columbus, produces a number of different source code metrics for various languages. Some of the most commonly used metrics are the following:

NOI Number of outgoing invocations, **NII** Number of incoming invocations, **LOC** Lines of code, **TLLOC** Total logical lines of code, **LLOC** Logical lines of code, **NUMPAR** Number of parameters, **NL** Nesting level, **McCC** McCabe's cyclomatic complexity, **NOS** Number of statements.

These common metrics were considered during the experiments, however the current version of CodeMetropolis does not use all of them (we detail the used ones below).

To create a visualization with sufficient expressive power, the structure of the system has to be displayed as well. The graph input contains this information, expressed with the edges of the graph. From several types of edges only the containment relation was used.

Figure 2 shows a simple input graph. It represents a small Java program which only consists of a couple of source files. The source code and the graph of this example were included in the sample inputs.

B. Data mapping

The current version of the converter uses the following entities and attributes to visualise the source code. These items

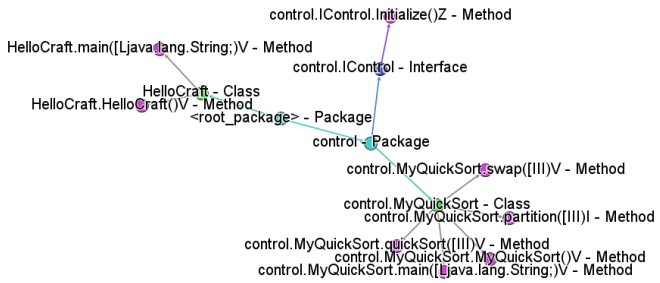


Figure 2. Simple graph input

are highlighted on Figure 3.

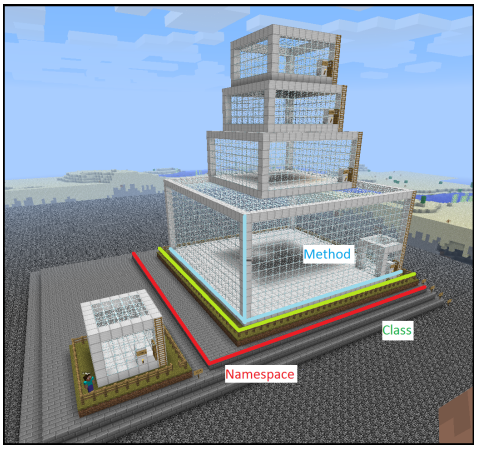


Figure 3. Items of the metaphor level

- A Plate is able to group various type of entities including other plates. It is used to display the namespace hierarchy like a tree-map. In the generated world, it is displayed as a solid rectangle of stone blocks. Its width and length were adjusted automatically to fit to its contents. Attributes:
  - Name
- Districts are similar to Plates. It is used to represent individual classes. It is displayed as a plate of grass blocks surrounded with fences. Attributes:
  - Name
- A Building is another compound entity. It consists only of floors which are placed on the top of each other ordered according to their width and length. The converter uses this entity to group floors, however it has no meaning on the data level. Attributes:
  - Name
- Floor (Figure 4) is a simple box of glass blocks with iron lattice and bottom. It represents a single method. Its width and length are mapped to McCC and its height represents the size of the method in the logical lines of the code. Attributes:
  - Name
  - Width
  - Length
  - Height

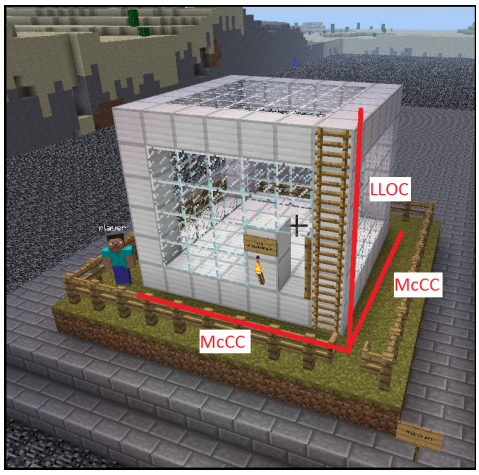


Figure 4. Attributes of a floor

All of these entities have a minimal size, to let the player walk in and are labelled with post or wall signs showing the name of the represented source code item (Figure 5). Because these sign could be seen only from short distance, we plan to provide some location information via a mini-map of the city.

Figure 5. In-game labels

Figure 6 shows a concrete class from JUnit represented with a single building on a plate. The values of the mapped metrics are shown as well, which are normalized. The values are scaled between a minimum (4 blocks) and a maximum (25 blocks) values. The source code of this class can be seen in Listing 1.

```
public class InvokeMethod extends Statement {
    private final FrameworkMethod fTestMethod;
    private Object fTarget;

    public InvokeMethod(FrameworkMethod testMethod,
        Object target) {
        fTestMethod = testMethod;
        fTarget = target;
    }

    @Override
    public void evaluate() throws Throwable {
        fTestMethod.invokeExplosively(fTarget);
    }
}
```

Listing 1. Source of `InvokeMethod` class

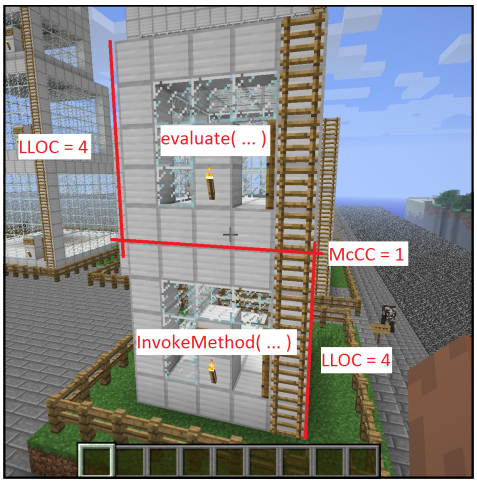


Figure 6. Example class visualization

C. Multilevel data visualisation

One of the problems of understanding complex systems is the flow of information. In this case, the users have too much low level information and they are unable to filter out the irrelevant data. To overcome this problem, we introduced multilevel data visualisation. With this technique, the users have the opportunity to decide how much details they would like to see.

In Minecraft, this is simply done by looking at the entities from different distances. Since Minecraft allows one to fly in a so called creative mode, the user can inspect the metropolis from the bird's eye view thus seeing the large scale size of the classes and the namespaces. Then, one can walk on the streets and compare the size and the complexity of the methods. It is even possible to go inside the methods and – in the future version – explore their inner structure represented with furniture.

III. Benefits and Use cases

The usefulness of a tool like CodeMetropolis depends on various factors including the experience and personal values of the users. People who naturally use similar metaphors to understand the world this could be a straightforward way of visualisation, while for many others the approach would be merely an interesting but generally an experimental idea.

With these considerations kept in mind, a couple of use cases will be explained. There are two main applications of the data visualisation tools – and CodeMetropolis. It can be used on real life projects, or in classrooms, to help students understand complex relations and concepts. In the use cases, we were focusing on the first one, but the corresponding use cases of the second one can be easily found.

A. Bad smells detection

Bad smells are the parts of the code which will probably cause an error or are difficult to maintain. [2] Developers usually identify bad smells by reading the source code.

With CodeMetropolis, developers are actually able to see some of the smells. For example, if the logical lines of the methods are mapped to the height of the floors – as in the current version – a giant class can be easily found by searching the tallest building of the metropolis. Then, by walking or flying around this building the count and size of the methods could be examined. If there are only a couple of them, these are large methods that need to be split up. On the other hand, when there are a lot of small methods, the class might be a so called god or brain class which needs to move its functionalities into another class, maybe a new one.

B. Inspect the structure of code

When new developers are assigned to an ongoing project they have to understand the large scale structure of the system. They usually browse via many directories and source files, or in better cases they look at the different diagrams and documentations of the system. But any case simply browsing through a myriad of code and documents is tedious and the developers' motivation rapidly decreases. Furthermore, documentations is often outdated or incomplete.

The worlds of CodeMetropolis are generated from the source code, so they always reflect the current state of the system. Developers can fly over the metropolis and see the clusters of the classes and the namespaces.

C. Annotating entities

When developers are inspecting the source code, they often leave comments to mark its parts.

The future version of our conversion tool will support code annotation. When developers put a wall or post a sing on some entities (floors, buildings), the text on it will be inserted into the source code as a comment. Furthermore, in the multi-player mode developers can see and interact with each other, so some parts of a code review meeting can be held in the Minecraft world.

D. Present code history

There are several open-source Minecraft servers which support extendibility. We plan to use these servers to visualize the code history gathered from the version controlling system.

With the use of historical data, representations can be generated for each revision, then several computer controlled players will be used to literally build these states of the metropolis. Each player represents a developer and will build the parts of the buildings that they coded. For example, if a developer inserts a new method into a class, the player will go to the corresponding building and build a new floor representing the new method. Project managers and other developers can join the server and see the evolution of the system.

E. Identify untested code

Test coverage data will be integrated into a future version of the converter. For example, torches and glowstones can be used to illuminate the building representing the tested parts of the code. The height of the fences around the classes can represent the number of test cases associated with the given class. The colour of these fences can be used to indicate the result of the test case (pass, fail).

F. Understanding inter-metrical relations

Since source code can be complex having many different properties and attributes and relations to other entities, classical visualisation techniques like graphs and charts cannot represent so much information at once. However, to understand the relation and connection among the various metrics, the global context has to be analysed.

To address this problem, CodeMetropolis will use various sophisticated metaphors. For example, a floor represents a method. Its width and length are mapped to its complexity and its height indicates its size. Furthermore, the number of windows and doors visualise the count of its parameters. There are torches on the wall if the method is tested and the fences around it indicate the number of passed test cases. Even if the developers do not know the formal definition of these metrics, they are able to see the consequences of their actions while writing the code. Sooner or later, they will assign informal meanings to the different kinds of graphical elements in the metropolis and they will perceive the represented source code as a whole.

IV. Current state

CodeMetropolis is in a prototype state, and we have a lot of plans for various additional functionalities to be implemented. However, with this paper we intend to publish current results and further ideas as a proof of the implementability of the concept.

This version was written in C# using the .Net framework. It is a command line tool which takes the previously mentioned graph as input and creates a Minecraft world from it. There are a couple of open-source API-s for every major language which support editing or creations over these worlds. Our tool uses the Substrate [9] library for .NET Framework.

Currently, only the previously mentioned complexity and size metrics are visualised with the entities listed in Section II-B and only the namespaces, classes and methods are represented. However, even with this limited toolset we were able to visualise complex, real life systems and gain useful insights into the systems. As an example, Figure 1 shows the view of the JUnit [3] metropolis.

A. User feedback

> "It makes software metrics such fun that you want to do it." *an user*

We performed a number of interviews among our colleagues that included university lecturers, students and developers working on industrial code, altogether six users was asked. We wanted to gain early feedback on the converter and the visualisation technique in general. In this section, both the negative and the positive impressions about CodeMetropolis will be summarised.

The negative opinions are grouped into several topics. The first of these concerns is the problem of great distances. The metropolis of a large or medium scale project can be huge and the players need a lot of time to navigate in it. We plan to solve this problem by implementing a quick map and a navigation system. These will be included in the multi-player server enabling the players to see their location and quickly teleport to other places.

Other opinions were concerned about the learning curves, either about Minecraft or about the visualisation. In our opinion, Minecraft has very simple control logic and in-game physics, no cryptic keyboard short-cuts, or complex machinery. It can be learned quickly and easily while playing or with the use of sophisticated online resources [7] covering every detail. Minecraft supports two special items among others: cheats, which can store other items and book them, and quill, which allows the players to take notes and write books in the game. CodeMetropolis will also support these to create in-game explanation notes for the items of the metaphor level.

The last problem was the lack of simultaneous data visualisation. The users could identify only three attributes: width, length, and height. This limited set is not enough to visualise the complex items of the data level, but as explained above we will be able to extend the tool with additional properties easily.

The positive impressions are also grouped into categories. The first of these is about the expressive power provided by the metaphor and the in-game logic of Minecraft. The second one is that the "work while you play" approach can maintain or even increase the motivation of the users, especially students. Finally, the users mentioned one of our further plans, the round-trip source code management. For example, if the players destroy a floor in the metropolis, the corresponding method will be removed from the source code.

V. Future works

In this section, we will enumerate some of our further plans. Since almost all of these were explained in depth in the previous sections, here only a short list is given.

Extending the palette of the entities and attributes The future version of our converter will use an extended palette of the blocks supported in Minecraft. For example, flowers to decorate beautiful code and zombies (hostile creatures) to indicate bad practices.

Navigation support We plan to implement a mini-map and a teleportation system. The related classes will be connected with railways allowing the users to navigate and see the connections.

Round-trip source code management The changes between the source code and the metropolis will be propagated to each other.

In-game explanations Post and wall sings and books will be used to explain the meaning of the various attributes and to show the source code of the corresponding element.

Visualize source code history The functionality of open-source multi-player severs will be extended to visualize source code history. For example, computer controlled players (npc-s or bots) will build the metropolis as the developers commit their changes into the version control system.

VI. Conclusion

In some cases, developers need to step away from the source code and inspect the system from a different perspective. We believe that CodeMetropolis will be able to

maintain motivation without sacrificing productivity thanks to its intuitive and, for many people, already known graphical surface. The provided metropolis metaphor has enough expressive power to represent the complex items of the source code. Combined with high quality graphical techniques provided by today's computer games, it is able to offer a rich graphical interface, an easy to learn controlling, and a rich user experience. It is probably easier to fit in classrooms than in a commercial project. However, we will continue its development to integrate the functionalities which are useful for developers, for students, and for teachers.

APPENDIX A
DEMONSTRATION

The demonstration will be carried out live, which means that a small example project written in Java will be visualised from the source code to the metropolis in Minecraft. We will go through the following steps:

1) The problem will be presented in a couple of slides to warm up the audience.
2) A sample world will be generated by Minecraft to illustrate the possibilities of the game.
3) A small sample code will be introduced.
4) The example project will be converted to a Minecraft world.
5) Every attribute mentioned in this paper will be explained on the metropolis generated in the previous step.
6) The authors and the audience will explore together the metropolis of JUnit.

APPENDIX B
EXECUTABLES AND SAMPLES

The current version of CodeMetropolis can be downloaded from the following url: http://www.inf.u-szeged.hu/~geryxyz/code-metropolis.html. The published package contains the executables and two sample projects: JUnit and HelloCraft, both of them were mentioned in this paper, together with sample inputs and outputs.

The tool is distributed in two ways: in portable binaries and in a setup package. It requires the 4.5 version of the .NET framework. After installing or copying the executables into the desired location, the conversion can be started with the `CodeMetropolis.exe <input-file>.graph` command from the command-line prompt. It will prompt to press the Enter-key when the conversation is finished. The results will be produced in the CodeWorld directory under the current directory. It will contain the generated Minecraft world which needs to be copied to `<user-home>\AppData\Roaming\.minecraft\saves` under Microsoft Windows 7. Then the user will be able to open it with Minecraft as a usual world. The tool was tested with 1.5.2 version of Minecraft.

REFERENCES

[1] Rudolf Ferenc et al. "Columbus – Reverse Engineering Tool and Schema for C++". In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*. IEEE Computer Society, Oct. 2002, pp. 172–181.

[2] Martin Fowler et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999, p. 464. ISBN: 0201485672.

[3] *JUnit Official Website*. URL: http://junit.org/.

[4] D Lalanne and J Kohlas. *Human Machine Interaction: Research Results of the MMI Program*. 2009.

[5] A. Marcus, D. Comorski, and A. Sergeyev. "Supporting the evolution of a software visualization tool through usability studies". In: *Proceedings of the 13th International Workshop on Program Comprehension* (May 2005), pp. 307–316. DOI: 10.1109/WPC.2005.34.

[6] *Minecraft Official Website*. URL: http://minecraft.net/.

[7] *Minecraft Wiki - The ultimate resource for all things Minecraft*. URL: http://www.minecraftwiki.net/wiki/Minecraft_Wiki.

[8] Margaret-anne Storey, Casey Best, and Jeff Michaud. "SHriMP views: an interactive environment for information visualization and navigation". In: *CHI '02 extended abstracts on Human factors in computing systems - CHI '02* (Apr. 2002), p. 520. DOI: 10.1145/506443.506459.

[9] *substrate-minecraft A .NET SDK for editing Minecraft worlds*. URL: https://code.google.com/p/substrate-minecraft/.

[10] Richard Wettel and Michele Lanza. "CodeCity". In: *Companion of the 13th international conference on Software engineering - ICSE Companion '08*. New York, New York, USA: ACM Press, May 2008, p. 921. ISBN: 9781605580791. DOI: 10.1145/1370175.1370188.

[11] Kenny Wong. "Rigi user's manual". In: *Department of Computer Science, University of Victoria* (1998).

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

MetricMiner: Supporting Researchers in Mining Software Repositories

Francisco Zigmund Sokol, Mauricio Finavaro Aniche, Marco Aurélio Gerosa
Department of Computer Science
University of Sao Paulo (USP) - Brazil
E-mail: francisco.sokol@usp.br, {aniche, gerosa}@ime.usp.br

***Abstract*—Researchers use mining software repository (MSR) techniques for studying software engineering empirically, by means of analysis of artifacts, such as source code, version control systems metadata, etc. However, to conduct a study using these techniques, researchers usually spend time collecting data and developing a complex infrastructure, which demands disk space and processing time. In this paper, we present MetricMiner, a web application aimed to support researchers in some steps of mining software repositories, such as metrics calculation, data extraction, and statistical inference. The tool also contains data ready to be analyzed, saving time and computational resources.**

***Index Terms*—mining software repositories; supporting tool; code metrics**

I. Introduction

Techniques of mining software repositories enables researchers to study software engineering practices empirically. Practitioners by means of these techniques uncover useful information for the software development team, such as frequently changed or error-prone classes, or the identification of core developers in order to transfer knowledge. With this information exposed, teams can take actions to improve their code and processes.

To develop a study in the area, researchers need to gather large amounts of data sometimes from many different projects and store them in their own workstations or servers. Then, manually run code metrics, and perform statistical calculations. This process requires the installation of several tools and libraries, making the process complex and slow. Besides the complexity, this kind of research consumes many computational resources. To start with, the repositories download uses a reasonable amount of bandwidth. After being processed and persisted in a database, the data occupies a huge disk volume. To calculate metrics on a lot of artifacts a large amount of processing time is required. Finally, after all these steps, it is possible to extract information, and evaluate them by means of statistical analysis. It means that researchers spend a lot of time working on the tools, rather than in analyzing the data and interpreting the results.

Based on all these difficulties, we decided to develop MetricMiner, a web application that performs all these steps without requiring great effort from the researcher. With it, researchers can write new metrics, and extract information from a reasonable quantity of different projects. In this paper, we present the tool, its functionalities, and architecture decisions. We also present a replication study that was developed using the tool.

II. MetricMiner: A Web Application to Support Research in MSR

Understanding the process of software evolution is a hard task. Large systems tend to have a long development history, with many different developers working on different parts of the system. It is common that developers do not know the entire source code of the project. Because of that, the idea of a manual analysis of all software is impracticable.

Mining Software Repositories (MSR) analyses the software evolution in an automated way, through the application of data mining techniques into the development history data. Studies in this field reveal useful information to the development of a particular project or even find patterns in software evolution that can be generalized to other software systems.

The term *"software repository"* comprises all artifacts created during the development of a software system. From source code files that are stored in a source control manager, such as Git or SVN, to messages that developers exchange in mailing lists. Such repositories contain useful information, which can be explored to comprehend the software evolution and contribute to its development.

MetricMiner is a web application that aims to support researchers when working with mining software repositories. As a web application, MetricMiner makes use of the power of cloud computing to scale. This way, researchers do not have to worry about resources. Resarchers also do not spend time installing different tools and libraries. Currently, MetricMiner is running over a cloud infrastructure and it is currently available at http://metricminer.org.br/.

MetricMiner was based on rEvolution[1], a command-line tool that extracts data from a local repository and persists them in a database. rEvolution was limited to collect data from just one project, requiring researchers to execute it manually for each repository. In addition, the configuration of the tool was complex. It was necessary to configure database, source control tools, and all external applications that the tool uses in a single XML file.

The tool automatically clones source code repositories, processes all repository metadata, stores it in a database,

[1]http://github.com/mauricioaniche/rEvolution.

	Quantity
Total of projects	317
Total of committers	5,274
Total of commits	887,493
Total of artifacts	1,453,123
Minutes of work spent with tasks	39,203

TABLE I
CURRENT NUMBERS OF PROJECTS IN METRICMINER

allows researchers to create queries, to manipulate data, and even to run statistical tests on the data set.

In the subsections below, we better explain the functionalities as well as the architectural decisions. All the source code and development history is hosted on Github, under an open source license[2].

A. Current Features

When researchers decide to do a study in MSR, they first choose projects. When using MetricMiner, the researcher only needs to insert the name of the project and the URL to its repository into the web application. Currently, MetricMiner supports Git and SVN repositories. As soon as the researcher inserts the project, the tool automatically starts to clone the repository, and to run all the existent metrics for each version of the project. In addition, researchers can access repositories that were previously inserted by other researchers. As it is all done in the cloud, MetricMiner currently contains a reasonable number of projects that were already processed. For example, all Apache open source projects are already there. In Table I, we show the current numbers of projects and commits in MetricMiner.

As all steps in the process usually take time, all tasks are executed asynchronously. MetricMiner has an internal queue, in which tasks are executed. Researchers can monitor the web application to find out when their tasks finished.

At the end of the cloning process, all relevant information, such as source code, list of committers, date/time information, code metrics, are persisted in the database. MetricMiner already contains several metric implementations: cyclomatic complexity [6], lack of cohesion of methods (LCOM) [3], efferent coupling [5], amount of lines of code [1], and amount of method invocations [4].

Researchers are able to query all the data that is in the tool by just typing a SQL query. MetricMiner execute the query asynchronously, store the result, and send the researcher an email notification. Then, the researcher can download the results in a standard CSV format[3], and refer to the results in their papers by an unique URL that MetricMiner provides. Currently, CSV is the only format implemented to export query results.

As new projects can be added at any time, queries can be re-executed at any moment. However, no data is lost; older results are saved and their unique URL persists, even after the query is executed again. In Figure 1, we show the screen in which developers can create SQL queries.

[2]http://github.com/metricminer-msr/metricminer. Last access on June, the 9th, 2013.

[3]To preserve the identity of the developers, the tool replaces the developers' name and emails by a hash string.

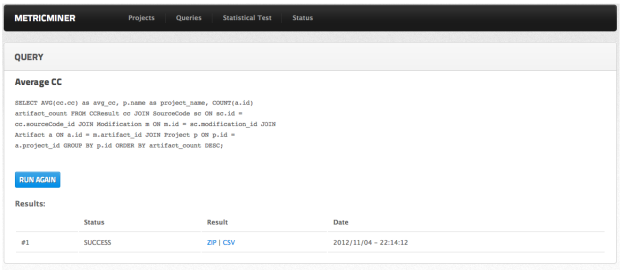


Fig. 1. A query in MetricMiner.

Besides downloading it as a CSV, researchers can also use the result of a query as an input to a statistical test. To do so, they choose two datasets and the statistical test (for example, T Student, Wilcoxon, and so on). MetricMiner then dynamically writes and executes an R script [4], and stores the result in an unique URL.

Researchers can also navigate into each project, as they have their own page. In Figure 2, we show the homepage for the Ant project, which contains more than 12,000 commits. The tool presents basic information, such as the total number of commits, committers, first and last commit's date/time. The tool also shows a few charts, showing the number of commits in the last twelve months, and the number of files changed in each commit for the last six months. Researchers can also add tags to the project to group similar projects in a future data extraction.

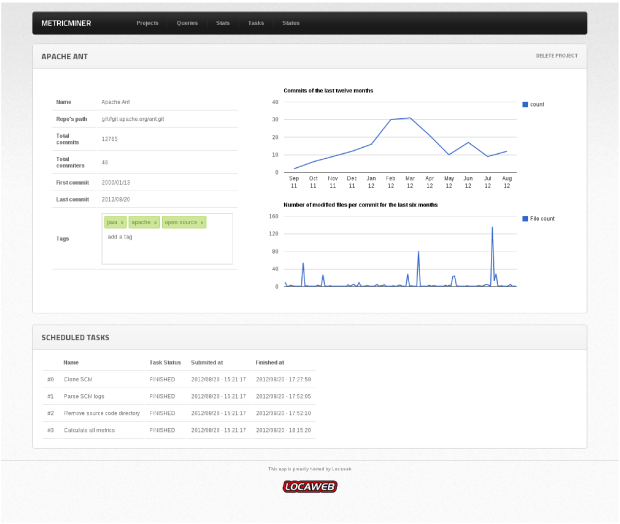


Fig. 2. Ant homepage, available on http://metricminer.org.br/project/12.

B. Architecture and Design Decisions

In Figure 3, we show the basic flow in MetricMiner. MetricMiner was developed in Java, and can be deployed in any Java web container. Currently, the application is running in *Apache Tomcat*. The tool also uses MySQL to store the data.

[4]The R Project for Statistical Computing. http://www.r-project.org/.

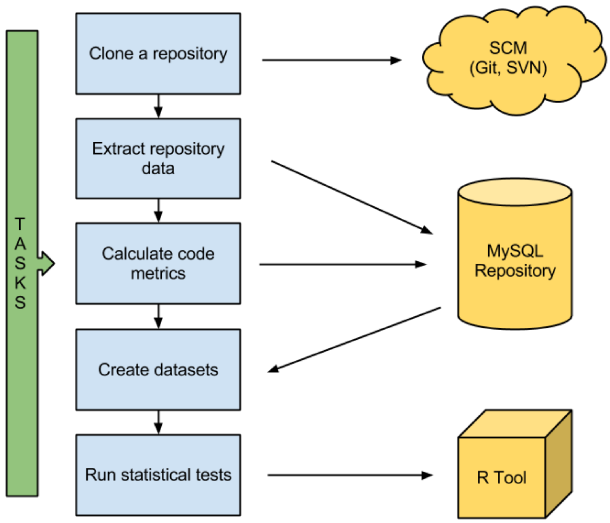


Fig. 3. Basic flow of working in MetricMiner

The domain model contains all information from the source control repository. It stores commits, authors, dates, and source codes. Each commit is related to a set of modifications. These modifications can be classified as a "change," "new file," or "deleted." For each file, MetricMiner stores the current source code and the difference (*diff*) to the last commit. The tool does not store binary artifacts, such as images, zip files, etc. By storing all the information, we enable researchers to create new metrics, and run over old repositories, without the need of restarting the whole process.

Internally, the tool uses a queue to organize the tasks it needs to execute. Tasks represent the steps that MetricMiner takes to extract information from a project: repository cloning, extract repository metadata, run code metrics, and run a statistical test. The task design is extensible. In Figure 4, we show an UML diagram that represents the tasks. If one needs to create a new task, s/he basically needs to create a concrete implementation of the interface. This design enables us to easily extend MetricMiner in the future to mine other sources, such as bug tracking systems and mailing lists.

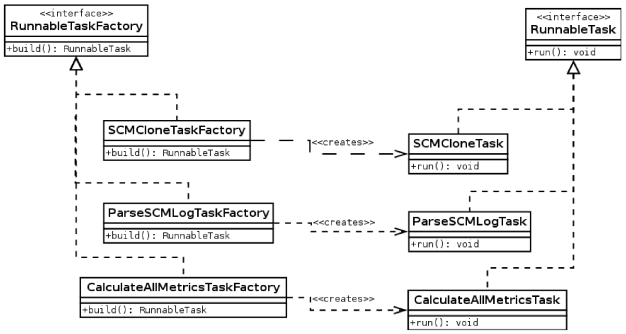


Fig. 4. UML that represents a Task in MetricMiner

Analogously, new code metrics can be inserted into MetricMiner. Researchers only need to implement a set of interfaces, and the metric will be ready to be executed in all source code. The interface is simple. There are three methods that need to be implemented: one that calculates the metric based on the source code, one that returns one (or many) results based on the calculation, and one that returns if that metric should be executed for that file (Java metrics should run only in *.java files, for example).

Not all code metrics require compiled code: they use statically analysis. The process of compiling a project can be costly and tricky, as each project contains its own way to be built. The existent metrics of MetricMiner use the *javaparser*[5] library. With it, the abstract syntactic tree is built and each metric is implemented as a visitor of this tree.

Many examples of it can be seen in the MetricMiner source code. This link[6] points to the implementation of the McCabe metric, using the mentioned visitor.

It is important to highlight the fact that, as soon as a new metric is implemented, MetricMiner automatically detects it, and schedules the execution of it to all repositories that exist in the tool.

The tool also abstracts the persistence problem: researchers are able to create the domain object the way they want, and MetricMiner, by using Hibernate, persists it, without the need of writing any SQL Insert statement. It means that researchers should only worry about creating an object-oriented model of their metrics. In Figure 5, we show the UML diagram that represents the internals of two code metrics in MetricMiner.

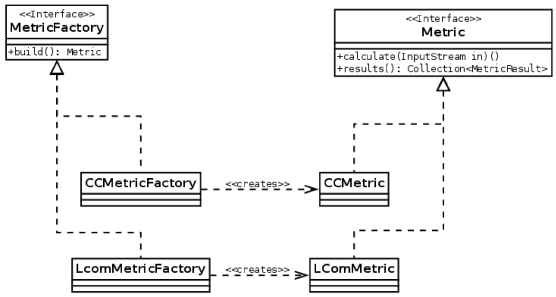


Fig. 5. UML that represents a Metric in MetricMiner

Currently, the metrics implemented are applicable only for Java sources. Nevertheless, implementing code metrics for other programming languages in the future will be easy, as the design of the classes that calculate the metrics are decoupled from the remainder of the system.

Performance is also something important for researchers. Therefore, during development we focused in optimizing the execution of tasks such as those described in the Figure 4. We tried different strategies to make the execution of all tasks faster. At the end, the current implementation makes use of first and second level caching, stateless Hibernate sessions, and data pagination. In numbers, we improved the average time of task from 5.3 minutes to 3 minutes.

C. Evaluation of the Tool

We evaluated the capability of the tool by mining all source code repositories available in the Apache Software Foundation.

[5]https://code.google.com/p/javaparser/. Last access on June, the 5th, 2013.
[6]https://github.com/metricminer-msr/metricminer/blob/master/src/main/java/org/metricminer/tasks/metric/cc/CCVisitor.java.

	Decreased CC	Did not change CC	Incremented CC
Documented refactoring	14 / 1,504	7 / 1,603	12 / 3,230
No refactoring documented	27 / 30,145	580 / 99,580	136 / 121,239

TABLE II
FINDINGS IN SOETENS AND DEMEYER STUDY / OUR REPLICATION

All repositories were accessed via Git. The complete list of projects can be found at http://git.apache.org/. There were 307 different projects in the website. At the end of the process, MetricMiner was able to process and store more than 800,000 commits from more than 2,000 different committers. There were more than 1.5 million different artifacts and 5 million different versions of source code. All these data are stored in a database with more than 180 gigabytes and available to researchers by means of the web application. All the process, from the beginning of the first project up to the metrics of the last one, took 90 hours.

In addition, to exemplify the value of the tool to researchers in the field, we reproduced a published study. Soetens and Demeyer [8] studied the effects of refactoring over the complexity of the system. They analyzed 776 versions that were extracted from PMD[7]. For each commit in the source control, the authors identified if a refactoring had occured (by reading the commit message), and whether the cyclomatic complexity of the project decreased or not.

To perform the analysis, the authors made use of two Eclipse plugins: the *SVNKit*, to extract data from the SVN repository, and the *Eclipse Metrics*, to calculate the cyclomatic complexity of the code. With these two plugins, they developed their own plugin to do all the work of loading the version of code, calculate the metrics, and save them into an XML file. After that, the XML files were processed and the associated metrics were related to a commit message. Then they classified the commits in two sets: those containing refactorings (denominated "documented refactoring") and the normal commits. Finally, the effects over the Cyclomatic Complexity was calculated.

We then used MetricMiner to reproduce their work. To do so, all we had to do was a query to extract the cyclomatic complexity of all classes that were updated in all projects[8]. As MetricMiner contains many projects, we were able to analyse more than 200,000 commits; 250 times more commits than the original study.

In Table II, we show the results found in their study versus our replication. One can notice that the amount of data analyzed was much higher than in the original paper. The full replication can be found in [9].

The extension of the original paper was facilitated by having all the data already available and common metrics, like cyclomatic complexity, already calculated for all source code artifacts. In addition, it is important to notice that we executed our study in more than 300 projects, while the original authors did it in only one project. The use of MetricMiner enabled us to run the study over many projects at once.

[7]http://pmd.sourceforge.net/

[8]The query and the resulting dataset can be found online at http://metricminer.org.br/query/1

III. RELATED WORK

A similar tool to MetricMiner is Sonar[9], a web application that analyzes the source code and extracts a huge variety of code metrics and structural dependencies among classes. The focus of this tool is to support the development team to keep track of the code quality. Sonar does not store metadata from the code control system and does not provide a way to extract its data. However, the number of different data visualizations is noticeable. Because of that, Sonar is frequently used in industry, but not for academic purposes.

Eclipse Metrics[10] is an Eclipse plugin that calculates code metrics. Developers can keep track of their code evolution while they are programming. However, the plugin does not execute metrics on the whole repository, but only in the current codebase. The tool requires compiled code, which may be a problem depending on the repository being analyzed.

Kalibro[11] and Analizo[12] are tools that calculate different code metrics. While Analizo calculates metrics in many different languages, Kalibro focuses on giving support to developers, giving them reference values to the metrics and pointing out potential problems. The tool is very flexible: developers can configure the reference values, and compose new metrics using JavaScript. However, Kalibro does not analyze the whole repository history. The user needs to select versions and recalculate the metrics manually.

Mezuro[13] is a web application built over Kalibro and Analizo. Through its web interface, the user adds software projects and the metrics are calculated over the code (using the Analizo tool). Reference values are presented to the users, suggesting good and bad points in the source code.

EvolTrack is a software evolution visualization tool. Developed as an Eclipse plugin, EvolTrack processes the history in a source code control, and enables users to see the evolution of the classes over time. The tool shows the class diagram of the project and allows the user to go forward or backwards, seeing the classes that were added or removed. Currently, the tool supports only SVN repositories. EvolTrack also has plugins for different data visualizations, such as EvolTrack-SocialNetwork [10], which shows the relationship among the developers during the evolution of the software.

ArchView [7] is a visualization tool to analyze the software evolution. The tool extracts information from the source control (CVS) and from the bug tracking (BugZilla). After processing the information, ArchView enables users to visualize different metrics from an artifact in any version of the software, using Kiviat diagrams.

CodeCity [11] makes use of the metaphor "software as a city," and generates a city based on the source code. Each

[9]http://www.sonarsource.org/

[10]http://metrics.sourceforge.net/

[11]http://www.kalibro.org/

[12]http://www.analizo.org/

[13]http://mezuro.org/

	MetricMiner	Sonar	Eclipse Metrics	Kalibro/Analizo	Mezuro	Evoltrack	ArchView	CodeCity	Boa
Web application	X	X	-	-	X	-	-	-	X
Interface to query data	X	-	-	-	-	-	-	-	X
Code metrics	X	X	X	X	X	-	X	-	-
Code metrics in non-compiled source-code	X	-	-	X	X	-	-	-	X
Graphic interface to visualize data	-	-	-	-	-	X	X	X	-
Statistical tests	X	-	-	-	-	-	-	-	-
Git repositories	X	-	-	-	X	-	-	-	-
SVN repositories	X	-	-	-	X	X	-	-	-
CVS repositories	-	-	-	-	X	-	X	-	-

TABLE III
COMPARISON BETWEEN METRICMINER AND OTHER TOOLS

class is a building. Classes are grouped in neighborhoods. The height and weight of the building depends on the number of methods and attributes of the class.

Boa [2] is a domain-specific programming language for analyzing ultra-large-scale software repositories. Boa makes use of distributed computing techniques to execute queries against software repositories in an efficient way. As soon as the researcher learns the DSL, they can extract interesting information in a simple way. By February of 2013, Boa had almost 700,000 projects in its repository.

In Table III, we compare the related tools. The main difference of MetricMiner is that it stores the calculated value of the metrics so that the queries may be executed very fast and researchers may adopt an exploratory approach in the large amount of data, rapidly prototyping their study, without needing to install anything in their workstations.

IV. CONCLUSION AND FUTURE WORK

In this paper, we described MetricMiner, the web application that facilitates the work of researchers involved in mining software repositories. The tool aids researchers in all steps during a MSR study, such as cloning the repository, extracting data, creating specific datasets, and running statistical tests. Using the tool, we were able to extract a huge quantity of data from many repositories. We were also able to replicate a study of the literature and extend it to more than 300 projects. The tool's extension points support the creation of specific metrics or tasks, enabling researchers to deal with the data in a personalized fashion.

More improvements will be done in MetricMiner in the future. Parallelization of the tasks execution, for instance, may improve more the performance of the application. In terms of metrics, it is possible to implement more metrics. Collecting data from other repositories, such as bug tracking and mailing lists would provide researchers the possibility of triangulating their findings. We also plan to gather more studies from the MSR literature and replicate them in MetricMiner as a way of evaluating and extending the tool support. These replication may be useful to identify data analysis patterns which may be made available in MetricMiner.

ACKNOWLEDGMENT

We thank Locaweb for sponsoring the project, providing us their cloud infrastructure. Marco Gerosa receives individual grant from CNPq.

REFERENCES

[1] C. Chidamber, S.; Kemerer. A metrics suite for object oriented design. pages 476–493. IEEE TSE, Vol. 20 (6), 1994.

[2] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. International Conference on Software Engineering (ICSE2013), 2013.

[3] B. Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, 1996.

[4] W. Li; S. Henry. Object-oriented metrics that predict maintainability. J. Systems and Software, vol. 23, no. 2, 1994.

[5] J. Lorenz, M.; Kidd. *Object-oriented metrics: A Practical Guide*. Prentice-Hall, 1994.

[6] T. McCabe. A complexity measure. pages 308–320. IEEE TSE, SE-2, Vol. 4, 1976.

[7] Martin Pinzger. *ArchView - Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, 2005.

[8] Quinten David Soetens and Serge Demeyer. Studying the effect of refactorings: a complexity metrics perspective. In *QUATIC 2010: The 7th International Conference on Quality in Information and Communications Technology*. IEEE Computer Society Press, IEEE Computer Society Press, 2010.

[9] F. Sokol, M. F. Aniche, and M.A. Gerosa. Does the act of refactoring really make code simpler? a preliminary study. In *4th Brazilian Workshop on Agile Methods*, 2013.

[10] C. M. Vahia, A. M. Magdaleno, and C. M. L Werner. Evoltrack-socialnetwork: Uma ferramenta de apoio à visualização de redes sociais. In *Congresso Brasileiro de Software (CBSoft) – Sessão de Ferramentas*, 2011.

[11] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In Jonathan I. Maletic, Alexandru Telea, and Andrian Marcus, editors, *VISSOFT*, pages 92–99. IEEE Computer Society, 2007.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

Assembler Restructuring in FermaT

Martin Ward
Software Technology Research Lab
De Montfort University
Bede Island Building,
Leicester LE1 9BH, UK
martin@gkc.org.uk

***Abstract*—The FermaT transformation system has proved to be a very successful tool for migrating from assembler to high level languages, including C and COBOL. One of the more challenging aspects facing automated migration, specifically when the aim is to produce maintainable code from unstructured "spaghetti" code, is to restructure assembler subroutines into semantically equivalent high level language procedures. In this paper we describe some of the many varieties of assembler subroutine structures and the techniques used by the migration engine to transform these into structured code. These transformations require a deep analysis of both control flow and data flow in order to guarantee the correctness of the result.**

Two separate case studies, involving over 10,000 assembler modules from commercial systems, demonstrate that these techniques are able to restructure over 99% of hand-written assembler, with no human intervention required.

I. Introduction

According to IDC research, much of the world's information still resides on mainframe systems [1] with some estimates claiming that over 70% of all business critical software runs on mainframes [5]. In industries such as banking, transportation, finance, insurance, government, and utilities, mainframe systems continue to run critical business processes. The most commonly used language in these systems is COBOL, but a significant proportion of systems are implemented in Assembler: amounting to a total of around 140 – 220 billion lines of assembler code [3]. The percentage of assembler varies in different countries, for example, in Germany it is estimated that about half of all data processing organizations uses information systems written in Assembler [6]. A typical large organisation will have several million lines of assembler code in operation: for example, the US Inland Revenue Service has over ten million lines of assembler currently in use.

In a recent survey of 520 CIOs internationally [2], more than half (56%) said that mainframe developers were struggling to meet the needs of the business and 78% stated that the mainframe will remain a key business asset over the next decade. 71% of CIOs are concerned that the looming mainframe skills shortage will hurt their business.

Analysing assembler code is significantly more difficult than analysing high level language code. With a typical well-written high level language program it is fairly easy to see the top-level structure of a section of code at a glance: conditional statements and loops are clearly indicated, and the conditions are visible. A programmer can glance at a line of code and see at once that it is, say, within a double-nested loop in the **else** clause of a conditional statement.

978-1-4673-5739-5/13/$31.00

Assembler code, on the other hand, is simply a list of instructions with labels and conditional or unconditional branches. A branch to a label does not indicate whether it is a forwards or backwards branch: and a backwards branch does not necessarily imply a loop. Simply finding all the branch instructions which lead to a particular label involves scanning the whole program: and this does not take into account the possibility of "relative branch" instructions where the target of a branch is an offset from a label, or from the current location. An unlabelled instruction can therefore still be the target of a branch: so simply determining the "basic blocks" of assembler code is far from trivial.

As well as being more difficult to analyse, for a given functionality there is more code to analyse. A single function point, which requires 220 lines of C or COBOL to implement, will need about 400 lines of macro assembler or 575 lines of basic assembler to implement. On the other hand, a higher level language such as perl will require only 50 lines on average to implement one function point [4].

Assembler systems are also more expensive to maintain than equivalent systems written in high level languages. Capers Jones Research computed the annual cost per function point as follows:

Assembler	£48.00
PL/1	£39.00
C	£21.00
COBOL	£17.00

Many of these systems were originally implemented in Assembler due to the need to maximise the limited memory, CPU and disk capacity of the systems available at that time. Today, the greatest need is for flexibility: the ability to update systems to meet new business challenges. There is therefore a great need to migrate from assembler to more modern high-level languages, and to move from the mainframe to more cost-effective hardware platforms.

In previous papers [7,8,9,10,11] we have described the application of program transformation technology to automated migration from assembler to a high level language. The basic approach follows three stages:

1) Translate the assembler into our internal Wide Spectrum Language (called WSL);
2) Apply correctness-preserving WSL to WSL transformations to the code to restructure, simplify, raise the

abstraction level, etc. These may include syntactic and/or semantic code slicing;
3) Translate the high-level WSL directly into the target language (currently either C or COBOL).

Since these papers were published there have been many improvements made to FermaT including:

- Improved detection and translation of self-modifying code;
- Extensive jump table detection;
- Improved dataflow analysis;
- Array detection and analysis (including detection of arrays of structures);
- Implementation of program slicing for WSL (our internal Wide Spectrum Language) and assembler;
- Static Single Assignment computation;
- Improvements in subroutine restructuring.

In this paper we focus on the last of these improvements, which tackles one of the more challenging aspects of assembler restructuring: extracting self-contained procedures from a mass of spaghetti code containing subroutine calls and returns.

First, a note on the terminology we use:

Module A module is assembled from a single source file plus associated copybooks and macros and generates a single listing and object file. FermaT processes each module in an assembler program separately and generates a target language file for each assembler module;

Assembler Program An assembler program consists of one or more modules which are assembled separately into object files which in turn are linked together to form an executable file;

Subroutine A subroutine is a group of assembler instructions which can be called from within the same module via an instruction which saves the return address (the address of the instruction following the call) in a register and branches to the start of the subroutine. A subroutine returns by branching to the saved return address;

Return Register The register in the subroutine call instruction in which the return address is stored;

Procedure A procedure is a WSL programming construct which can be called and which always returns to the statement after the call. The aim of assembler restructuring is to convert subroutines to procedures, and thereby eliminate the use of code addresses in the program.

II. Assembler to WSL Translation

A WSL *action system* consist of a collection of parameterless procedures with a starting action name and takes the form:

actions A_1 :
$A_1 \equiv$
  $\mathbf{S}_1$ **end**
...
$A_n \equiv$
  $\mathbf{S}_n$ **end endactions**

Within each action body $\mathbf{S}_i$, an action call of the form **call** A_j acts as a procedure call which results in the execution of the body $\mathbf{S}_j$. If execution reaches the end of $\mathbf{S}_j$ then it continues with the statement after the **call**. There may also be a special action Z such that **call** Z causes termination of the whole action system: in this case control flow is passed directly to the statement following the action system. If every execution of an action leads to an action call (i.e. control can never directly reach the end of the action body), then the action is called *regular*. If every action in the action system is regular then no action call will return and the action system can only be terminated by a **call** Z. In this case, actions are similar to labels and action calls are similar to **goto** statements.

The FermaT assembler to WSL translator translates each assembler module to a regular action system: so an unconditional branch translates directly to an action call, and a conditional branch translates to an **if** statement containing an action call. Any instruction or macro which causes the module to return to the caller, or to terminate abnormally, is translated using a **call** Z.

The translator works from the listing produced by the IBM assembler, rather than the collection of source files. This has the disadvantage that the translator has to be able to recognise and parse the many different listing formats produced by different versions of the assembler and different option settings. To do this, the translator is table driven: a text file `listings.tab` contains information about all the recognised listing formats (including the Tachyon assembler, Siemens assembler, Amdahl Personal Assembler, CA-Realia 370 Macro Assembler, MicroFocus Assembler, and z390 assembler). In all, there are a total of 38 different listing formats currently recognised.

Despite this disadvantage, there are several advantages to working from the assembler listing:

1) With a fully expanded listing, all information including copybooks and macro expansions is known to be present;
2) Object code data, instruction addresses and branch target addresses are available;
3) The cross reference listing gives the length, type, location and value of each assembler symbol.

The above information is, of course, implicitly present in the source code, but deriving it requires duplicating much of the functionality of the assembler. In addition, the translator would need to know exactly which options were used to assemble the production module: and the most accurate source of *this* piece of information is the listing itself.

The 16 general purpose registers are translated to the special variables r_0 to r_{15} and the condition code is translated to the special variable cc which can take one of the three values 0, 1, 2 or 3. Each instruction or macro is translated to a single action whose body consists of WSL code which implements all the behaviour of the instruction (including side-effects such as setting the condition code). The action name is either the label on the instruction or macro or a name of the form A_hexloc where hexloc is the six or eight digit hex representation of the current location (as provided in the listing).

As well as IBM mainframe assembler, WSL translators for other languages including Intel x86 assembler and a proprietary 16 bit embedded systems processor have been developed.

A. Subroutines in IBM Assembler

The IBM 370 and z/OS mainframes do not have a hardware stack. Inter-module calls are handled via a linked list of save areas. Each save area stores copies of all register values together with pointers to the next and previous save areas, and it is standard for every module to save and restore all registers in its savearea. So a caller can assume that register values are preserved over calls to an external module. Note that some non-standard code will pass parameters in non-standard registers, and even return results in registers: the migration engine can detect and process this non-standard code. Within a single module (which may consist of many thousands of lines of code) a different mechanism is used for subroutine call and return:

- A subroutine call is implemented as a BAL (Branch And Link) or BAS (Branch And Save) instruction. This stores the return address (the address of the next instruction in the sequence) in the indicated register and then branches to the indicated label. The subroutine body is responsible for storing the return address elsewhere, if the register is needed within the subroutine (or any of the subroutines it calls) for some other purpose.
- A subroutine return is implemented by reloading a register with the return address (if necessary) and then executing a BR (Branch to Register) instruction which branches to the address in the register.

Return addresses may be saved and restored in various places, loaded into a different register, overwritten, or simply ignored. Also, a return address may be incremented, or the return instruction may branch to an offset of the given return address. This feature might be used to branch over parameter data which appears after the BAL instruction, or to branch to one of several different return points. Merely determining which instructions form the body of the subroutine can be a major analysis task: there is nothing to stop the programmer from branching from the middle of one subroutine to the middle of another routine, or sharing code between subroutines, or branching directly out of a subroutine instead of doing a normal return to the stored address.

B. WSL Translation of BAL and BR

The WSL language does not have the concept of a "code address": so the assembler to WSL translator has to use some mechanism to emulate the BAL and BR instructions. A return address is represented as the integer value of the location of the instruction (this is the offset of the instruction from the start of the module).

A branch to register (BR) instruction is translated into WSL code to copy the register value to the special variable destination, and then call the dispatch action. The dispatch action tests destination against all the possible return addresses (these are all the possible return addresses which are encountered during the translation of the module). If the value matches, then we call the corresponding action. These return addresses are called "dispatch codes".

A branch and link instruction (BAL, BALR, BAS, BASR etc.) is translated to code which stores the dispatch code of the return point in the register indicated in the instruction, and then calls the action specified by the label in the instruction. The translator ensures that this dispatch code will be included in the dispatch action.

As an example, the following fragment of an assembler listing starts at location `0x0002F6`

```
CONVNX3  DS    0H
* LINK TO CONVERSION RTN
         BAL   R14,MM2MTH
* MOVE MMM TO OUTPUT
         MVC   DDO2M,WRKMTH
```

It translates to the following three WSL actions:

```
CONVNX3 ≡
  call A_0002F6 end
A_0002F6 ≡
  C : LINK TO CONVERSION RTN;
  r14 := 762;
  call MM2MTH end
A_0002FA ≡
  C : MOVE MMM TO OUTPUT;
  DDO2M := WRKMTH;
  call A_000300 end
```

Note that each action ends with a **call** to the next action in the sequence. The value 762 is the decimal equivalent of the hex location `0x0002FA` for the return address. Code to test detination against the value 762 is also added to the dispatch action:

```
dispatch ≡
  if destination = 0 then call Z
  ...
  elsif destination = 762 then call A_0002FA
  ...
  fi end
```

The body of subroutine `MM2MTH` ends with a BR instruction to return from the subroutine:

```
MM2MTH   EQU   *
* CONVERT MM TO MMM (ALPHA)
...
MM2MTHX  EQU   *
* RETURN FROM SUB ROUTINE
         BR    R14
```

This translates to the WSL code:

```
MM2MTH ≡
  C : CONVERT MM TO MMM (ALPHA);
  call A_000374 end
...
MM2MTHX ≡
  call A_0003D0 end
A_0003D0 ≡
```

C : RETURN FROM SUB ROUTINE;
destination := r_{14};
call dispatch **end**

So, the code at A_0002F6 sets r_{14} to the value 762 and branches to MM2MTH. When action MM2MTHX is called, the value in r_{14} is copied to destination and dispatch is called. Since destination has the value 762, the dispatch action will call A_0002FA and the code following the subroutine call is executed.

III. WSL to WSL Transformation

Stage two in the migration process is the automated application of WSL to WSL program transformations. Space precludes a full discussion of all the transformations applied in the migration process: since are 156 different transformations currently implemented in the engine, of which around 50 are used in a typical assembler to COBOL migration. The process is controlled by the transformation Fix_Assembler which analyses the module at each stage and determines the next transformation to apply. Many transformations are applied repeatedly: typically a module has several thousand transformations applied before translation to the target language, although large and complex modules can require more than one million transformations!

In this paper we will focus on the transformations which convert assembler subroutine code to inline code or WSL procedures. These transformations come under the general heading of "dispatch removal" since the aim is to eliminate calls to the dispatch action, and ultimately eliminate the action itself.

A. Single Call

The simplest case is where a subroutine is only called from one place in the program. In this situation, the value of the return address at the start of the subroutine is a known constant: provided the start of the subroutine can only be reached from the single call. (In other words, there are no direct branches to the subroutine entry point or to labels inside the subroutine body, and control flow cannot "fall through" into the body of the subroutine). If this is the case, then the Constant Propagation transformation will replace references to the return register, which appear in the body of the subroutine, by the actual dispatch code. For example, if the call to MM2MTHX at A_0002F6 in our example were the only call to that subroutine (and there was no other way to reach the subroutine body), then Constant Propagation would replace references to r_{14} by the dispatch code 762. When Constant Propagation encounters a call to dispatch it checks if destination contains a known dispatch code: which is true in this case. If so, then the call to dispatch is expanded (the call is replaced by the body of the action) and simplified. The result is that the **call** dispatch statement is replaced by **call** A_0002FA.

A dispatch call has been eliminated and further restructuring of the action system will have the effect of "inlining" the procedure body. In the case of a simple subroutine, there is an option in FermaT which will allow the subroutine body to be recovered and converted to a WSL procedure just before translation to the target language. It is still important to inline subroutines wherever possible since this can allow FermaT to unscramble some unstructured code around the module. For example, a branch out of the middle of the subroutine does not cause problems if the subroutine has been inlined.

Constant Propagation can also determine when a branch to register is actually a return from the module itself to the calling module. Registers are initialised with a special dispatch code which indicates a return from the module when branched to (i.e. the WSL program should terminate). If dataflow analysis shows that this dispatch code reaches a branch to register, then the **call** dispatch is replaced by a **call** Z. Similarly, a branch to register which is applied to the address of an external module is converted to a call to the module. If there is a valid dispatch code in another register, then the called module is assumed to return to this address.

B. Multiple Calls to a Simple Subroutine

A simple subroutine is one with the following characteristics:

1) It has a single entry point;
2) It only returns directly to the caller: e.g. it does not branch to the middle of another subroutine;
3) It contains no calls to other subroutines.

This last restriction might appear to be rather severe: but note that once a subroutine has been inlined or transformed into a WSL procedure, there are no longer any *subroutine* calls to that code. Once all the subroutines called by a subroutine have been processed, the transformed subroutine body will no longer contain *subroutine* calls, and requirement (3) is satisfied.

The FermaT transformation engine therefore processes subroutines in a "bottom up" order as they appear in the subroutine call graph: "leaf" nodes which contain no subroutine calls are processed first, followed by the next higher "layer" in the call graph, and so on.

Therefore, provided all subroutines satisfy requirements (1) and (2), and there are no recursive calls, then there will be subroutines which satisfy requirement (3), and by processing the call graph in a "bottom up" order, *all* subroutines can be handled by this method.

Transforming a simple subroutine into a WSL procedure takes these stages:

1) **Control Flow Analysis**: Starting with the subroutine entry point, FermaT traces forwards through the action system call graph to find the actions which compose the body of the subroutine. The analysis stops at any call to dispatch. When the analysis is complete, each of the actions in the proposed procedure body is checked to see if it is reachable from outside the subroutine without going through the entry action. This may be due to a branch into the middle of the subroutine body, or a branch out of the subroutine body (for example, into a generic error handler). Since there is no way of determining, in advance of the analysis, which instructions comprise the subroutine body, a branch out of the body will initially cause the external code to be included in the

proposed body. See below for how these situations are dealt with.

2) **Data Flow Analysis**: Once a suitable procedure body has been determined, FermaT carries out a data flow analysis on the proposed procedure body. This analysis checks that the value assigned to the return register on entry to the subroutine will be propagated to the destination variable for every call to dispatch. The analysis needs to track the return address through any assignments which save and restore the return register. Note that some subroutines may increment the return address before returning (see below);
3) **Create a New Procedure**: If the above tests are successful then the set of actions composing the body of the subroutine are extracted from the main action system and composed into a new (sub) action system with the subroutine entry point as the entry action. Within this new action system calls to dispatch are replaced by **call** Z. This action system forms the body of a new WSL procedure. Calls to the original subroutine in the main action system are replaced by a call to the WSL procedure followed by **call** dispatch;
4) **Constant Propagation**: The dispatch calls introduced in step (3) can now be eliminated via constant propagation. Since the subroutine call has been converted to a procedure call, the return address can be propagated over the procedure body and used to eliminate the call to dispatch. Note that if control flow falls through into the subroutine body from the body of another subroutine, or if there is a branch to the subroutine entry point from another subroutine, then there may be dispatch calls which cannot be removed at this stage.

Each time a simple subroutine is successfully converted to a procedure, one or more calls to dispatch (the return points of the original subroutine) are eliminated from the program.

In our example, it turns out that the subroutine MM2MTHX is a simple subroutine which does not call any other subroutines, so it can be converted to a WSL procedure. The result is:

```
CONVNX3 ≡
  call A_0002F6 end
A_0002F6 ≡
  C : LINK TO CONVERSION RTN;
  r14 := 762;
  MM2MTH();
  call dispatch end
A_0002FA ≡
  C : MOVE MMM TO OUTPUT;
  DDO2M := WRKMTH;
  call A_000300 end
```

where:

```
proc MM2MTH() ≡
  actions MM2MTH :
  MM2MTH ≡
    C : CONVERT MM TO MMM (ALPHA);
    call A_000374 end
  ...
  MM2MTHX ≡
    call A_0003D0 end
A_0003D0 ≡
  C : RETURN FROM SUB ROUTINE;
  destination := r14;
  call Z end
```

We have removed the single **call** dispatch at the end of the subroutine and inserted calls to dispatch at each of the original subroutine calls (which are now procedure calls). However, constant propagation can remove these calls. For example, the **call** dispatch above transforms to **call** A_0002FA. The assignment $r_{14} := 762$ can also be deleted, since we know that this dispatch code has been accounted for. If there are no other references to this dispatch code, then we know that a **call** dispatch can no longer lead to **call** A_0002FA, so the call to A_0002FA can be removed from dispatch. In turn, this allows the call to be restructured: for example, if the call in A_0002F6 is the only call to A_0002FA, then it can be expanded and the action deleted from the action system.

This algorithm will handle any simple subroutine. Once all the subroutine calls in a subroutine body have been converted to procedures, then the subroutine itself can be converted. So, if an assembler module consist entirely of simple subroutines, it can be fully restructured using these techniques: regardless of the degree of subroutine call nesting present. However, in practice, there are many modules which do not keep to the constraints of a simple subroutines. The exceptions include:

- Subroutines which exit abnormally from the middle (for example, branching directly to common error handling code). This is extremely common in assembler programs;
- Falling through from one subroutine into the start of another;
- Branching from the middle of one subroutine into the middle of another subroutine;
- Returning directly to the caller's caller (instead of via the immediate caller);
- Multiple entry points to a subroutine, or equivalently, having a section of common code shared by several subroutines;
- Multiple return points: either returning to the given return address or to an offset on the return address;
- Passing parameters as inline data after the subroutine call. Here the subroutine uses the return register to address data, then increments it to get the actual return address;
- Sometimes saving the return address and sometimes not;

All the above exceptional cases appear so regularly that any automated assembler migration solution needs to be able to handle them. Each of these will be discussed in more detail in subsequent subsections.

In addition to the above, there are also frequently "bugs" in the assembler code which can prevent the module from restructuring. The code might be a genuine bug in the sense

that the module would crash or give incorrect results under certain circumstances, or it might be a highly convoluted way of coding something which happens to give the correct results but is very difficult to analyse and understand. Such highly convoluted code may only work "by accident" in the sense that a small and apparently innocuous change to the program may cause it to stop working correctly.

The most common bugs which prevent a module from restructuring are:

- Recursive subroutines, or mutually recursive sets of subroutines. Since assembler calls do not use a stack (unless explicitly programmed to do so), but store the return address in a fixed memory location, a direct or indirect recursive call will cause the original return address to be overwritten. A common example of this is when an error handler needs to write to a file or write a message to the operator, and the file operation or message handler itself checks for errors and calls the error handling routine. This may not be discovered in testing if it is unlikely for an error to occur while displaying a message during error handling. However, the fact that this control flow path appears may well prevent the module from restructuring. It is also regarded as a bad programming practice.

- Returning from a subroutine before it has been called. This can occur when there is a control flow path from a module entry point to the code which loads a saved return address and then executes a BR instruction to return from a subroutine, and where there is no call to the subroutine along the path. If this path is taken, then the program will either crash (if nothing has been saved in the return address) or branch to the return point taken in the last call to the subroutine (which may have occurred in a previous call to this module);

- Restoring the return address for a different subroutine: for example, subroutine A stores the return address in `ASAVE`, subroutine B stores the return address in `BSAVE` but then when B returns it loads the address in `ASAVE` and branches to it.

- Not restoring the return register on every path through the subroutine: there may be a path on which the return register's value is corrupted (e.g. by being used as the return register for another subroutine call), but is not restored. If this path can be taken, then it is clearly a bug, but if the path is not taken in normal processing its presence will still impede the restructuring process;

Less common bugs include calling a subroutine and passing the return address in the wrong register, eg calling via `BAL R15,SUBR` when `SUBR` expects a return address in `R14`! Another example is a branch back to the instruction which saves the return address: if this branch is taken after the return address register has been modified, then the corrupted value will overwrite the correct value in the save area.

All the above bugs (and many others!) have been found in production code.

C. Subroutine With Exit

The most common way in which a subroutine fails to be a simple subroutine is for there to be a branch out of the middle of the subroutine: typically this branch will be to error handling code. The subroutine has detected an error: so it is no longer interested in returning but branches directly to an appropriate error handling routine.

If all simple subroutines in a module have been processed, but there are still subroutines remaining, then another level of analysis is triggered:

- Any call to Z or to a label which is also reachable from outside the subroutine body is treated as an "abnormal exit" from the subroutine. The labelled action is not included in the procedure body, instead code is generated to store a value in the special variable exit_flag and the subroutine then returns to the caller. This flag is set to 0 for a normal return, a value of 1 means that the subroutine terminated by a **call** Z, each higher value (if any) indicates that the subroutine terminated by calling a distinct label outside the subroutine body;

- A dataflow analysis is then carried out on this modified subroutine body. If the analysis succeeds, then the subroutine is converted to a procedure.

- Subroutine calls are replaced by the following WSL code:
  SUBR();
  if exit_flag $= 0$ **then call** dispatch
  elsif exit_flag $= 1$ **then call** Z
  elsif exit_flag $= 2$ **then call** A
  ... **fi**
  where SUBR is the name of the new procedure.

- Constant propagation will now eliminate the dispatch call, as in Section III-B

If the subroutine body executes code which causes an abnormal exit or a return from the whole module, then this code will be translated to WSL statements ending in a **call** Z. This can be handled as an exit from the subroutine, as above. In our case studies, over 21% of modules needed exit_flag before they could be restructured (see Section V).

D. Returning to the Caller's Caller

Suppose subroutine `SUB1`, which has a return address in `R1` calls subroutine `SUB2` which has a return address in `R2`. The WSL code for a return from `SUB2` will therefore be:

destination $:= r_2$; **call** dispatch

However, `SUB2` might also decide to return to the caller's caller: i.e. to the address in `R1` which is the return address for `SUB1`:

destination $:= r_1$; **call** dispatch

If this is the case, then the dataflow analysis will fail: since the value of destination on this call to dispatch is not the value in r_2 on entry to `SUB2`.

To handle this situation, if the dataflow analysis fails then FermaT will check that:

- There are calls to dispatch where destination is loaded from the return register; and
- There are other calls to dispatch where destination is loaded from a *different* register.

If this is the case, then the second set of calls are treated as subroutine exits (as in Section III-C) and the dataflow analysis is re-computed. Note that this rule only applies when the original dataflow analysis failed: since it is possible for a subroutine to save the return register and reload it into a different register.

E. Fall Through into a Subroutine

Another common case is where a subroutine consists of some initial code followed by the execution of another subroutine, which uses the same return register. Instead of implementing a call to the second subroutine, a parsimonious programmer might just branch directly to the start of the second subroutine, or even arrange the code so that execution "falls through" into the top of the second subroutine. For example:

```
SUB1 ...
     body of SUB1
SUB2 ...
     body of SUB2
     BR    R14
```

where both SUB1 and SUB2 take a return address in R14. If SUB2 can be converted to a WSL procedure, then the resulting WSL code is:

SUB1 $\equiv$
... body of SUB1...; SUB2(); **call** dispatch **end**

Now it should be possible to process SUB1.

Note that if the initialisation code in SUB1 includes a subroutine return, then the branch (or fall through) to SUB2 can be mistakenly identified as an exit from SUB1. This does not necessarily prevent restructuring, but may cause problems later.

F. Multiple Entry Points

Section III-E is an example of a more general problem: a subroutine which has multiple entry points, or, equivalently, multiple subroutines which share a section of code. (These issues illustrate our comment in Section II-A that it can be difficult to determine which instructions form the body of a subroutine).

If the common code can be restructured into a single action which calls dispatch (or dispatch and Z only), then FermaT can create a WSL procedure out of the common code. In this case, the two subroutines can be "disentangled" since each includes a call to the common code.

G. Branch to the Middle of a Subroutine

A subroutine SUBX may include in its body a direct branch (conditional or unconditional) into the middle of another subroutine SUBY. This includes two common cases (among others):

1) SUBY may be the subroutine which called SUBX, or may be the caller's caller. In this case, instead of returning normally, we have an abnormal exit (Section III-C). If it is possible to call SUBX without having previously called SUBY, then there is a bug in the program: since when SUBY tries to return, there will be no valid return address.
2) SUBY may be using the same return register as SUBX. In this case, we have code which is common to both SUBX and SUBY. Instead of creating a new subroutine to share this common code, the programmer has made use of the fact that both subroutine use the same return register.

For correct restructuring, these cases may need to be handled differently. Case (1) is an exit from the middle of a subroutine (Section III-C), so the branch should be translated into code which sets a flag and returns. Case (2) is an example of shared code: the code we branch to needs to be converted to a procedure which can be called from both SUBX and SUBY.

FermaT can usually distinguish between these two situations due to a careful ordering of the application of restructuring heuristics.

H. Multiple Return Points

If a subroutine is carrying out a test, whose result needs to be returned to the caller, then the usual way to handle this is either:

1) Set a flag in the subroutine which is tested in the caller; or
2) Execute a test (eg a compare instruction) in the subroutine just before returning. The test will set the condition code, and the condition code is not modified by the Branch to Register instruction, so the condition code can be tested by the caller.

However, some programmers eschew these methods and instead make use of the fact that an unconditional branch instruction is exactly four bytes long. If an unconditional branch is inserted immediately after the call (BAL) instruction, then the called subroutine can choose to return to the instruction immediately *after* the branch, by incrementing the return address by four. The code to call the subroutine is:

```
          BAL   SUBR,R14
          B     SUBERR
NORM      ... normal return
```

In this case, the code labelled NORM may appear to be unreachable (typically, it is not even labelled): but it can be reached if SUBR increments the value in R14 by four before returning. Note that it is quite common for a BAL to be followed by an unconditional branch (which may in turn be followed by unreachable code) when the subroutine does *not* increment its

return address: so the two situations must be distinguished by the migration engine.

The body of SUBR may contain code like this:

```
SUBR     ...
         CLC   WRKMM,=CL2'12'
         BH    SUBRERR
         B     4(R14)
SUBRERR  BR    R14
```

If the month number (in WRKMM) is greater than 12, then we flag the error by returning to the return address passed by the caller (the caller will then execute the branch to SUBERR). Otherwise, we return to the address four bytes on from the given return address (which leads to the code labelled NORM).

The instruction B 4(R14) translates to WSL as:

destination $:= r_{14} + 4$; **call** dispatch

The same effect could be achieved via LR R14,4(R14) followed by BR R14, which will translate as:

$r_{14} := r_{14} + 4$; destination $:= r_{14}$; **call** dispatch

More generally, the subroutine call can be followed by two or more unconditional branches:

```
         BAL   SUBR,R14
         B     RET1
         B     RET2
         ... more branches
         B     RETn
FINAL    ... final return point is here
```

To select the return point, SUBR can increment R14 by the appropriate multiple of 4 (0, 4, 8, 12 etc.).

This will have the effect of preventing the dataflow analysis from succeeding: we cannot prove that the original value in the return register ends up in the variable destination, if the value has been incremented by 4 (or more) in between. To handle this case, the dataflow analysis needs to be more subtle:

- Any increment of the return address (by a multiple of 4) is noted, by setting the flag Incremented_Return;
- An increment of the return address by a multiple of 4 is treated as a copy, as far as the dataflow analysis is concerned.

With these modifications, the dataflow analysis will succeed but will note that the return address may be incremented. since the increment may be inside a loop, it may not be possible (via static analysis of the subroutine body) to determine the exact set of possible increments for the return address: and therefore the number of return points. Instead, the migration engine looks at the set of calls and checks for a sequence of one or more unconditional branch instructions after the call to determine the set of return points.

Note that there may also be abnormal exits from the subroutine: so we choose not to use the variable exit_flag to determine the required return point. Instead, we use the return register to indicate which return point the subroutine selected. The migration engine generates code which sets the return register to zero, then calls the procedure (generated from the subroutine body), then tests the return register to see which return point is required:

$r_{14} := 0$;
SUBR();
if $r_{14} = 0$ **then call** RET1
elsif $r_{14} = 4$ **then call** RET2
...
elsif $r_{14} = 4 * (n - 1)$ **then call** RETn
else call FINAL **fi**

I. Inline Parameters Passed to Subroutine

Usually, parameters are passed to a subroutine in named data areas, or in registers or via a pointer in a register. However, some programmers have noticed that the return register can serve two purposes: if we include inline data immediately after the subroutine call, then this data can be accessed via the return register.

This situation is particularly tricky because a single register is, in effect, being used to store two pieces of information: (a) a pointer to the parameters; and (b) an offset from the return address. In the assembler, the code is arranged so that these two values are the same, but in the WSL translation the values are distinct. Therefore, the assembler to WSL translator has to detect when parameters are being passed as inline data and generate specific code to handle this. Just because a subroutine call is followed by data does not necessarily mean that this data is being used as parameters: for example an error routine might be called via BAL, even though it does not return. In this case, the data could be the start of a data area used by the module. The assembler to WSL translator therefore checks that the following three conditions all hold:

1) The subroutine is called via a BAL or BALR which is followed by data declarations, which in turn are followed by more executable code;
2) The subroutine body uses the return register to address data, eg via a Load or MVC (Move Characters) instruction;
3) The subroutine return is to an offset on the return register.

If all these conditions hold, then the subroutine call is translated as follows:

1) Ensure that the first inline parameter (the first data declaration after the call) has a label;
2) Generate the following code for the call:
   $r_n :=$!XF inline_par(code, ADDRESS_OF(par));
   call SUBR
   where code is the dispatch code for the return address (the address of the code following the inline data), and par is the name of the first data area;
3) Change any branch to the return address plus an offset, where the offset equals the length of the inline data, to a direct return.

The transformations which scan WSL code looking for subroutine calls are modified to check for WSL code of the form

$r_n :=$!XF inline_par(code, ADDRESS_OF(par));
call SUBR

as well as $r_n :=$ code; **call** SUBR.

Once a subroutine has been detected and the control flow and dataflow analysis confirmed that it can be converted to a procedure, the statement

$r_n :=$!XF inline_par(code, ADDRESS_OF(par))

is converted to $r_n :=$ ADDRESS_OF(par). The converted subroutine body (which is now a WSL procedure) can now treat r_n as a simple data pointer from which the parameters can be addressed: in other words, the code in the subroutine which accesses the parameters requires no special handling.

This approach can also handle cases where a subroutine has both inline parameters *and* multiple return points.

J. Inline Code Converted to a Subroutine

One day a programmer wanted to re-use a section of inline code which appeared elsewhere in the program. Instead of going to all the trouble of extracting this section of code and turning it into a subroutine, he or she realised that by inserting a suitable Load Address instruction just before the block of code, and a Branch to Register instruction just after it, the block of code could be used as a subroutine without moving it out of place. The modified code looks like this:

```
        LA    R14,RETLAB
SUBR    . . .
        block of code is here
        . . .
        BR    R14
RETLAB  . . .
```

Elsewhere, the block of code can now be called as a subroutine via: `BAL SUBR,R14`

This form of subroutine causes no difficulty to FermaT because the Load Address instruction, followed by falling through to the `SUBR` label generates the following WSL:

$r14 := 1234$; **call** SUBR

where 1234 is the offset of the label RETLAB. The dispatch code 1234 is also added to the dispatch action with corresponding label RETLAB. This is identical to the code generated by a normal subroutine call. The code between SUBR and RETLAB is converted to a WSL procedure, called as follows:

SUBR(); **call** RETLAB

and the RETLAB action is then restructured in the usual way.

IV. WSL to Target Language Translation

Once the automated WSL to WSL transformation stage is complete, the final stage in the migration process is translation from WSL to the target language. However, this is preceded by a further transformation stage in which the WSL code is manipulated to bring it closer to the target language. This stage is particularly important for migration to COBOL since the COBOL programming language has many restrictions and limitations which must be accommodated in order to generate compilable and executable COBOL.

A major limitation with some COBOL compilers is a lack of bit manipulation functions. Although bit fields and bit operations were introduced to the language in the ISO/IEC 1989:2002 standard, which was published in 2002, with a CS (Committee Draft) available in 1997, current mainframe compilers do not have native support for these operations. The WSL to COBOL translator can generate code for several targets including the following:

1) If the target is for Microfocus COBOL, then calls are generated to the built-in bit operations in Microfocus;
2) If the target is for IBM mainframe COBOL, then calls are generated to assembler support functions which implement the bit operations. These can process the bit operation at full speed, albeit with the overhead of a call.

Pack and unpack instructions (which convert string data to packed decimal and vice versa) can usually be implemented as a COBOL MOVE. For example, moving from a decimal field to a packed field will convert the string of decimal digits to a packed decimal value. However, the assembler instructions do not check the validity of the data, so a pack or unpack from a one byte source to a one byte target simply reverses the nybbles in the source field. This operation is frequently applied to general hex data, so has to be translated as a call to a support function.

Pointers are available in COBOL via the `SET ADDRESS OF ...` and `SET ... TO ADDRESS OF` statements but many COBOL programmers are unfamiliar with pointers, so the transformations attempt to eliminate as many pointer operations as possible via dataflow analysis and converting pointers to array indices.

The actual translation step is then a simple line-by-line translation of the "COBOL-like" WSL into a COBOL source file. This is followed by a further conversion of the COBOL source which handles formatting details such as indentation levels, spacing, and coping with the COBOL file format. A COBOL source line has a fixed format, dating back to the punched card era. The on-line card readers for the IBM 704, 709, 7090 and 7094 computers (introduced between 1954 and 1964) operated only in 'row binary' format: reading cards row-by-row into 12 pairs of 36-bit words ($2 \times 36 = 72$). The reader was not capable of reading more than 72 of the 80 columns of a card, so early compilers and assemblers could only 'see' those 72 columns. All COBOL compilers, including the most recent versions, therefore ignore columns 73–80 in order to be compatible with existing source code.

V. Case Studies

To test the effectiveness of the FermaT migration engine at restructuring commercial assembler modules, we took a copy of an assembler system currently in production in a large American insurance company. The system consists of over 3,000 programs and 8,991 assembler modules (many of which are used in more than one program). The FermaT migration engine was able to restructure all but 84 out of the 8,991

modules for a success rate of 99.07%. A significant number of the failures turned out, on analysis, to be due to bugs in the code or to code which would only work "by accident" (in the sense that an apparently innocuous change to the code would cause an error elsewhere in the program). Of the modules which were able to be restructured, 1,953 (21.9%) had subroutines with unstructured exits: these needed the exit_flag variable to be added before they could be restructured (see Section III-C).

A second case study involved 1,822 modules from an employee management system. After fixing all bugs uncovered by the migration process, all but seven modules could be restructured automatically, for a success rate of 99.62%. Of the modules which were restructured, 368 (21.3%) needed the exit_flag variable.

A major requirement is for the migrated COBOL to be maintainable. All modules which restructure will be converted into a hierarchy of single-entry single-exit procedures consisting of structured code with no GO TO statements. In addition, the McCabe cyclomatic complexity is typically reduced by at least 25%.

An example of a bug uncovered by the failure to restructure is the following code:

```
          BAS   R04,S00100
          ...
S00100    do some processing
          ...
          L     R04,S00R04
          BR    R04
```

Elsewhere in the module, S00R04 is used to save and restore the return address in R4, but in this subroutine the register is "restored" without having been saved. The return address will be overwritten by the contents of S00R04, which might be zero, or the return address left over from a previous call to a different subroutine.

Another example:

```
          LA    R15,4           IT CAME FROM VIM
          BAL   R10,SUBR020     GO DECIDE WHICH ONE
          LTR   R5,R5           DID WE FIND ANYTHING
          BZ    ERROR040        NO, SO ERROR CONDITION
          ...
ERROR040  EQU   *       ERROR CONDITION IF WE GET HERE
          ST    R15,ERRPECD2    SAVE ERROR CODE
          LA    R15,252         MAJOR ERROR CODE
          ICM   R15,12,ERRMODID INDICATE THE MODULE
          BR    R10             AND RETURN
```

Here, this code was called with a return address in R10. The error handler at ERROR040 stores some information and then attempts to return. But when we branch to ERROR040 after calling SUBR020, R10 now contains the return address that was passed in the call to SUBR020 so BR R10 will branch back to the LTR instruction and loop endlessly.

A. Performance

Performance of the migrated code depends somewhat on the type of code and also depends on the target platform. Typically, there is little degradation in performance when the COBOL is running in the mainframe environment: one test reported a 4% decrease in performance. In some cases, the COBOL can perform better than the original assembler: for example, the COBOL compiler can make use of newer and faster instructions than were available to the original assembler programmers. Also, self-modifying code can cause a severe performance hit on modern machines since the whole instruction cache is flushed when any instruction is modified.

VI. Conclusion

Despite the enormous technical and theoretical challenges presented by the analysis of assembler code: totally automated migration of assembler to a high-level language such as C or COBOL is feasible with complete restructuring achieved for over 99% of assembler modules.

References

[1] Robert Amatruda, "The Critical Need to Protect Mainframe Business-Critical Applications," IDC, White Paper, Jan., 2012.

[2] Compuware Corporation, "Mainframe Succession: Long Live the Mainframe," Compuware, White Paper, 2012.

[3] Capers Jones, *The Year 2000 Software Problem — Quantifying the Costs and Assessing the Consequences.*, Addison Wesley, Reading, MA, 1998.

[4] Capers Jones, "Backfiring: Converting Lines of Code to Function Points," *IEEE Computer* 28 #11 (Nov., 1995), 87–88.

[5] J. Scott, "The e-Business Hat Trick — Adaptive Enterprises, Adaptable Software, Agile IT Professionals," *Cutter IT Journal* 13 #4 (Apr. 2000), 7–12.

[6] Harry Sneed & Chris Verhoef, "Reengineering the Corporation–A Manifesto for IT Evolution," ⟨http://www.cs.vu.nl/∼x/br/br.html⟩.

[7] M. Ward, "Assembler to C Migration using the FermaT Transformation System," *International Conference on Software Maintenance, 30th Aug–3rd Sept 1999, Oxford, England* (1999).

[8] Martin Ward, "Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations," *Science of Computer Programming, Special Issue on Program Transformation* 52 #1–3 (2004), 213–255, ⟨http://www. cse. dmu. ac. uk/ ∼mward/ martin/ papers/ migration-t.ps.gz⟩ doi:dx.doi.org/10.1016/j.scico.2004.03.007.

[9] Martin Ward & Hussein Zedan, "Combining Dynamic and Static Slicing for Analysing Assembler," *Science of Computer Programming* 75 #3 (Mar., 2010), 134–175, ⟨http://www. cse. dmu. ac. uk/ ∼mward/ martin/ papers/combined-slicing-t.pdf⟩ doi:10.1016/j.scico.2009.11.001.

[10] Martin Ward, Hussein Zedan & Tim Hardcastle, "Legacy Assembler Reengineering and Migration," *20th IEEE International Conference on Software Maintenance, 11th-17th Sept Chicago Illinois, USA.* (2004).

[11] Martin Ward, Hussein Zedan, Matthias Ladkau & Stefan Natelberg, "Conditioned Semantic Slicing for Abstraction; Industrial Experiment," *Software Practice and Experience* 38 #12 (Oct., 2008), 1273–1304, ⟨http://www. cse. dmu. ac. uk/ ∼mward/ martin/ papers/ slicing-paper-final.pdf⟩ doi:doi.wiley.com/10.1002/spe.869.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

A Hidden Markov Model to Detect Coded Information Islands in Free Text

Luigi Cerulo*‡, Michele Ceccarelli*‡, Massimiliano Di Penta†, Gerardo Canfora†
*Dep. of Science and Technology, University of Sannio, Benevento (Italy)
†Dep. of Engineering, University of Sannio, Benevento (Italy)
‡BioGeM s.c.a r.l., Institute of Genetic Research "Gaetano Salvatore", Ariano Irpino, AV (Italy)

***Abstract*—Emails and issue reports capture useful knowledge about development practices, bug fixing, and change activities. Extracting such a content is challenging, due to the mix-up of source code and natural language, unstructured text.**

In this paper we introduce an approach, based on Hidden Markov Models (HMMs), to extract coded information islands, such as source code, stack traces, and patches, from free text at a token level of granularity. We train a HMM for each category of information contained in the text, and adopt the Viterbi algorithm to recognize whether the sequence of tokens—e.g., words, language keywords, numbers, parentheses, punctuation marks, etc.—observed in a text switches among those HMMs. Although our implementation focuses on extracting source code from emails, the approach could be easily extended to include in principle any text-interleaved language.

We evaluated our approach with respect to the state of art on a set of development emails and bug reports drawn from the software repositories of well known open source systems. Results indicate an accuracy between 82% and 99%, which is in line with existing approaches which, differently from ours, require the manual definition of regular expressions or parsers.

Keywords: *HMM; Natural Language Parsing; Mailing list mining.*

I. Introduction

Data available in software repositories is useful for software analysis and program comprehension activities. In particular, mailing lists and issue tracking systems are widely adopted in open source projects to exchange information about implementation details, high-level design, bug reports, code practices, patch proposals, and malfunctioning details, such as stack traces. A common practice in open source projects is to adopt mailing lists or bug tracking systems as the sole repository of software technical documentation available.

Extracting useful, relevant, and unbiased information from such free text archives is not straightforward. In Information Retrieval (IR) free text data is usually treated as vector of words counts. This is a representation usually adopted in the treatment of free text in many software engineering approaches [1]. Although this simplification works well for pure natural language texts, in software engineering it may fail as free text, generated by developers, is often not well-formed and interleaved with different information contents written in different coding syntaxes.

In this paper we introduce an approach, based on Hidden Markov Models (HMM), to extract coded information islands, e.g. source code and natural language, from free text at a token level of granularity. We consider the sequence of tokens—e.g., English words, programming language keywords, digits, parentheses, punctuation symbols, etc.—of a development email as the emissions generated by the *hidden states* of a HMM. We use the hidden states to model a specific coded information content, e.g., source code and natural language text. The Viterbi algorithm allows us to search for the path that maximizes the probability of switching between text and source code hidden states given an emission sequence [2]. Such a path allows us to classify each observed token in the corresponding coded information category. The approach could be easily extended to include also other text interleaved languages, such as stack trace and patches. The primary novelty and point of strength of the proposed approach is that it *does not require the manual definition (case by case) of regular expressions or parsers*, generally required for alternative approaches [3], [4], [5].

The automatic detection of information contained in the free text of a development email is useful for various tasks. For example, recovering the traceability between source code and software documentation stored in email has been tackled with methods based on vector of terms [6], [7]. Such methods are not able to recognize whether a term refers to natural language or source code contexts. Instead, knowing which is the context provenance of an index term could be beneficial to improve the precision of the traceability relations by weighting more meaningful terms [3]. In particular, summarization techniques are usually designed for specific types of artifacts and cannot be applied to emails where a mixture of languages may coexist. Thus a method able to distinguish terms coming from different email contexts is crucial.

Generally speaking, removing irrelevant information from data may improve the quality of data extraction in approaches based on information retrieval techniques where a term's indexing procedure is adopted to build a language model [8]. The detection of different coding information in textual contents allows for excluding noisy data from the indexing process. Traceability recovery [6], impact analysis [9], bug report assignment [10], [11], code/text summarization [12], [13] are approaches that could benefit from methods that are able to discriminate noisy data and relevant information.

To evaluate our approach, we adopted a set of freely available HTML books, fully annotated with source code fragments

978-1-4673-5739-5/13/$31.00

(e.g., "Thinking in Java" by Bruce Eckel), random generated text files, and sets of emails and bug reports from two well-known open source software systems (Linux Kernel and Apache httpd). Results indicate an accuracy ranging between 82% and 99%, generally in line with alternative approaches (e.g., the one by Bacchelli *et al.* [3]) that require a manual customization and/or the definition of regular expressions or parsers.

The paper is organized as follows: Section II describes related work. Section III describes the proposed approach. Section IV provides details about the empirical evaluation procedure. Section V reports and discusses the obtained results. Section VI discusses threats to the validity of the evaluation, while Section VII concludes the paper and outlines directions for future work.

II. Related Work

The problem of extracting useful models from textual software artifacts has been approached mainly by combining three different techniques: regular expressions, island parsers, and machine learning. A first approach has been proposed by Murphy and Notkin [5]. They outlined a lightweight lexical approach based on regular expressions that a practitioner should follow to extract patterns of interests (e.g., source code, function calls or definitions). Bettenburg *et al.* developed *infoZilla*, a tool that allows to detect and extract patches, stack traces, source code, and enumerations from bug reports and their discussions [4]. They adopted a fuzzy parser and regular expressions to detect well defined formats of each coded information category obtaining, on Eclipse bug reports, an accuracy of 100% for patches, and 98.5% for stack traces and source code. Tang *et al.* proposed an approach to clean e-mail data for subsequent text mining [14]. They used an approach based on Support Vector Machines to detect source code fragments in emails obtaining a precision of 93% and a recall of 72%. Bacchelli *et al.* [15] introduced a supervised method that classify lines into five classes: natural language text, source code, stack traces, patches and junk text. The method combines term based classification and parsing technique, obtaining a total accuracy ranging between 89% and 94%.

Although such approaches are lightweight and exhibit promising level of performance they may be affected by the following drawbacks:

- *Granularity level*. Most of the methods, in particular those based on machine learning techniques, classify lines. Our method classifies tokens, thus reaching a finer level of granularity useful for high interspersed language constructs.
- *Training effort*. Methods based on island parsers and regular expressions require expertise for the parser or the regular expression construction. Furthermore, such approaches work well on the corpus adopted for the construction of the parser, however are not generalizable. Our method learns directly from data and does not require particular skills.
- *Parser limitations*. Context free parsers or regular expression parsers rely on deterministic finite state automata designed on pre-defined patterns. For example, in modeling the patch language syntax, Bacchelli *et al.* search for lines surrounding two @@s. This may be a limitation if such a pattern is not consistently used or exhibits some variations. Sometimes developers may report only the modified lines by copying the output of a differencing tool, and such output is slightly different from source code. Our method is based on Markov models, which rely on a nondeterministic finite state automaton making the detection of noisy languages, such as stack traces, more robust.
- *Extension*. Since using island parsers and/or regular expressions require a significant expertise, introducing a new language syntax can be problematic. We propose a method that learns directly from data, thus requiring an adequate number of training samples to model the language syntax of interest.

An application of a HMM in extracting structured information from unstructured free text has been proposed by Skounakis *et al.* [16]. They represent the grammatical structure of sentences with a hierarchical hidden Markov model and adopt a shallow parser to construct a multilevel representation of each sentence to capture text regularities. The approach has been validated in a biomedical domain for extracting relevant biological concepts.

III. Methods

In the following, we first report background notions about HMM. Then, we describe our approach for identifying source code fragments in natural language text by describing a basic HMM, an improved model able to detect islands of other languages than source code (e.g., logs, XML), and by providing details about model calibration.

A. Background notions on Hidden Markov Models

A Markov model is a stochastic process where the state of a system is modeled as a random variable that changes through time, and the distribution for this variable only depends on the distribution of the previous states (Markov property). A Hidden Markov Model (HMM) is a Markov model for which the state is only partially or none observable [17]. In other words, the state is not directly visible, while outputs, dependent on the state, are visible. Each state has a probability distribution over the possible output symbols, thus the sequence of symbols generated by an HMM gives some information about the sequence of hidden states.

Hidden Markov models are especially known for their application in temporal pattern recognition such as speech, handwriting, and gesture recognition [18], [19], part-of-speech tagging [20], musical score following [21], and bioinformatics analyses, such as CpG island detection and splice site recognition [22].

Formally, a HMM is a quadruple (Σ, Q, T, E), where:

- Σ is an alphabet of output symbols;

- Q is a finite set of states capable of emitting output symbols from alphabet Σ;
- T a set of transition probabilities denoted by t_{kl} for each $k, l \in Q$, such that for each $k \in Q$, $\sum_{i \in Q} t_{ki} = 1$.
- E a set of emission probabilities denoted by e_{kb} for each $k \in Q$ and $b \in \Sigma$, such that for each $k \in Q$, $\sum_{i \in \Sigma} e_{ki} = 1$.

Given a sequence of observable symbols, $X = \{x_1, x_2, \ldots, x_L\}$, emitted by following a path, $\Pi = \{\pi_1, \pi_2, \ldots, \pi_L\}$, among the states, the transition probability t_{kl} is defined as $t_{kl} = P(\pi_i = l | \pi_{i-1} = k)$, and the emission probability e_{kb} is defined as $e_{kb} = P(x_i = b | \pi_i = k)$. Therefore, assuming the Markov property, the probability that the sequence X was generated by the HMM, given the path Π, is determined by:

$$P(X|\Pi) = t_{\pi_0 \pi_1} \prod_{i=1}^{L} e_{\pi_i x_i} t_{\pi_i \pi_{i+1}}$$

where π_0 and π_{L+1} are dummy states assumed to be respectively the initial and final states. The objective of the decoding problem is to find an optimal generating path Π^* for a given sequence of symbols X, *i.e.*, a path such that $P(X|\Pi^*)$ is maximized:

$$\Pi^* = \arg\max_{\Pi} P(X|\Pi)$$

The Viterbi algorithm is able to find such a path in $O(L \cdot |Q|^2)$ time [2]. As a clarification example, let us consider the classical unfair-casino problem [22]. To improve the chance of winning, a dishonest casino uses loaded dice occasionally, while most of the time using fair dice. The loaded dice has a higher probability of landing on a 6, with respect to the fair dice where the probability of each outcome is equal to $1/6$. Suppose the loaded dice has the following probability distribution: $P(1) = P(2) = P(3) = P(4) = P(5) = 1/10$ and $P(6) = 1/2$. We are interested to unmask the casino. Thus, given a sequence of throw data, we would like to known when the casino uses a fair dice and when a loaded dice.

The HMM representing the rolling dice game is shown in Fig. 1, where the alphabet is $\Sigma = \{1, 2, 3, 4, 5, 6\}$, and the state space is $Q = \{FairDice, LoadedDice\}$. Suppose the dishonest casino switches between the two hidden states, fair dice and loaded dice, with the transition probabilities shown in the figure. When the casino uses the fair dice the emission probabilities are those of the fair dice, while when it uses the loaded dice the emission probabilities are those of the loaded dice. Let us assume that we observe the following sequence of rollings: 1, 2, 6, 4, 3, 6, 5, 2, 6, 6, 4, 1, 3, 6, 6, 6, 6, 6, 6, 6, 6, 5, 4, 6, 1, 6. We cannot tell which state each rolling is in. For example, the subsequence 6, 6, 6, 6, 6, 6 may happen using the loaded dice or it can happen using the fair dice even though the later case has less probability. The state is hidden from the sequence, e.g., we cannot determine the sequence of states from the given sequence. The Viterbi algorithm is able to determine (decode) the most probable sequence of states emitting the observed sequence of symbols [2].

TABLE I
THE TOKEN ALPHABET Σ.

Symbol	Token	Regexp
WORD	any alphanumeric character	`[a-zA-Z0-9]+`
KEY	a WORD token that is also a language keyword (eg. C/C++, Java, Perl, SQL, ...)	
UNDSC	the underscore character	`\_+`
NEWLN	the newline character	`[\n\r]`
NUM	a WORD token that is a pure sequence of digits	`\d+`
the char itself	any other character not matching the previous patterns	`[^\s\w]`

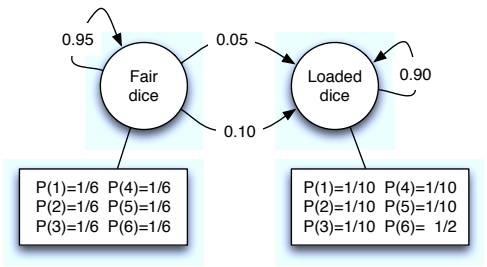


Fig. 1. The unfair-casino HMM.

B. A basic Hidden Markov Model

To model our problem we adopt an HMM defined as follows. We model a text as a sequence of tokens, *i.e.* sequences of characters separated by spaces (`\s`). Each token belongs to a symbol class that constitutes the alphabet of our HMM. Table I shows the alphabet and the regular expression adopted to detect those symbols in text.

Table II shows some examples of texts and their representation as a sequence of tokens. The goal is to detect whether a symbol encountered in a text sequence comes from the natural language text or it is part of a source code fragment. For example, encountering a source code KEY symbol does not guarantees that the portion of the analyzed text is a source code fragment as many keywords could be also part of the English natural language (e.g., `while`, `for`, `if`, `function`, `select`, ...). We model this behavior assuming that each symbol could be emitted by two states, one modeling natural text language, and one modeling source code text. The HMM is in fact composed by two sub-HMM, one modeling the transitions among symbols belonging to natural text language sequences, and another modeling the transitions among symbols belonging to source code text. Clearly, the transition

TABLE II
SOME TOKEN SEQUENCE EXAMPLES.

Text	Token sequence
`"My dear Frankenstein," exclaimed he, "how glad I am to see you!"`	`" WORD WORD WORD , " NEWLN WORD WORD , " WORD WORD WORD WORD NEWLN WORD WORD WORD ! "`
`for(int i=0;i<10;i++) s+=1;`	`KEY ( WORD WORD = NUM ; WORD < NUM ; WORD + + ) WORD + = NUM ;`

probabilities between two symbols could be different if they belong to different language syntaxes. For example, after a KEY symbol in natural language text, it is more likely to find a WORD symbol, while in the source code text it is more usual to find a punctuation mark symbol or special character, such as opening parenthesis. Formally, the HMM state space is defined as:

$$Q = \{\Sigma_{TXT}, \Sigma_{SRC}\}$$

where $\Sigma_{TXT} = \{\text{WORD}_{TXT}, \text{KEY}_{TXT}, \dots\}$, and $\Sigma_{SRC} = \{\text{WORD}_{SRC}, \text{KEY}_{SRC}, \dots\}$. Each state emits the corresponding alphabet symbol without subscript label $_{TXT}$ or $_{SRC}$. For example, the KEY symbol can be emitted by KEY_{TXT} or KEY_{SRC} with a probability equal to 1. If the probability for staying in a natural language text is p and the probability of staying in source code text is q, then the transition from a state in Σ_{TXT} to a state in Σ_{SRC} is $1-p$, instead the inverse transition is $1-q$. The above defined HMM emits the sequence of symbols observed in a text by evolving through a sequence of states $\{\pi_1, \pi_2, \dots, \pi_i, \pi_{i+1}, \dots\}$ with the transition probabilities t_{kl} defined as:

$$t_{kl} = P(\pi_i = l | \pi_{i-1} = k) \cdot p, \text{ if } k, l \in \Sigma_{TXT}$$
$$t_{kl} = P(\pi_i = l | \pi_{i-1} = k) \cdot q, \text{ if } k, l \in \Sigma_{SRC}$$
$$t_{kl} = \frac{1-p}{|\Sigma|}, \text{ if } k \in \Sigma_{TXT}, l \in \Sigma_{SRC}$$
$$t_{kl} = \frac{1-q}{|\Sigma|}, \text{ if } k \in \Sigma_{SRC}, l \in \Sigma_{TXT}$$

and the emission probabilities defined as:

$$e_{kb} = 1, \text{ if } k = b_{TXT} \text{ or } k = b_{SRC}, \text{ otherwise } 0.$$

Fig. 2 shows the global HMM composed by two sub-HMM, one modeling natural language text and another modeling source code. Fig. 3 and Fig. 4 show their transition probabilities, estimated on the Frankenstein novel and PostgreSQL source code respectively. We detail in Section III-D how these probabilities could be estimated.

It is interesting to observe how typical token sequences are modeled by each HMM. For example, in the source code HMM a number (NUM) is typically preceded by open braces or brackets modeling arguments in a function call or array indexing, the underscore character follows and is followed by a WORD modeling typical variable naming convention. Instead, in the natural language HMM, numbers (NUM) are preceded just by the dollar symbol ($) indicating currency, and likely followed by a dot, indicating text item enumerations. Instead, in the source code HMM it is noticeable that numbers are part of an arithmetic/logic expressions, array indexing, or function argument enumeration.

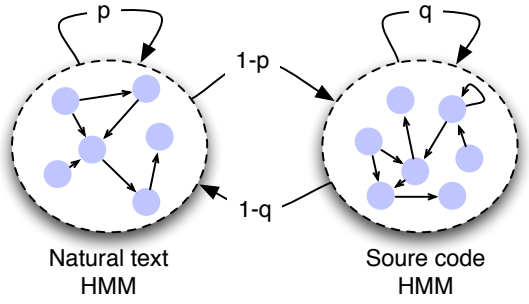


Fig. 2. The source code – natural text island HMM.

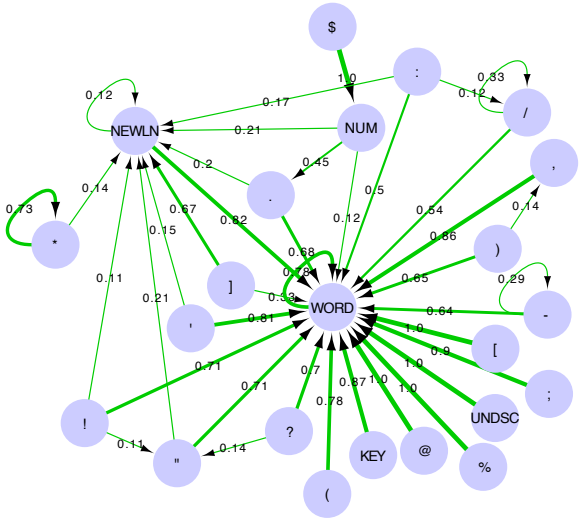


Fig. 3. A natural text HMM trained on the Frankenstein novel (transition probabilities less than 0.1 are not shown).

C. An extension of the basic model

The basic HMM can be extended to include other language syntaxes usually adopted in development emails, such as patches, log messages, configuration parameters, failure

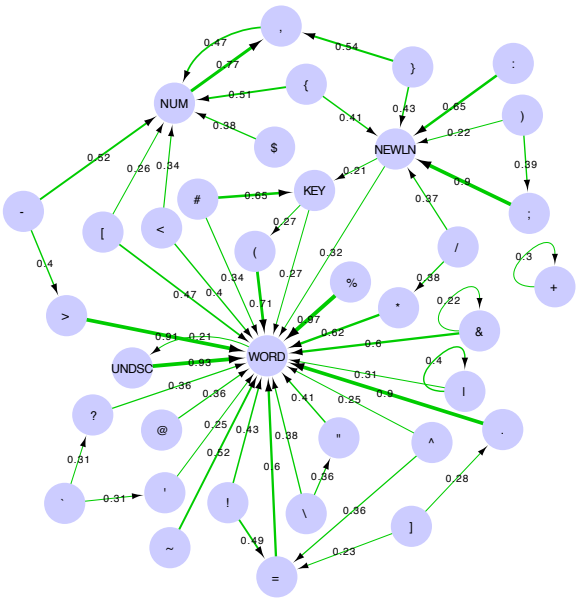


Fig. 4. A source code HMM trained on PosgreSQL source code (transition probabilities less than 0.2 are not shown).

reproducing steps, XML, and so on. In general to include n language syntaxes we introduce the language transition probability matrix $W = w_{ij}$, for $i, j \in \{1, 2, \ldots, n\}$ that defines the probabilities of staying into a particular language syntax and of switching from one syntax to another. Formally, if w_{ii} is the probability to stay into a language syntax i then the probability to switch from i to $j \neq i$ is given by: $w_{ij} = (1 - w_{ii})/(n - 1)$, supposing a uniform distribution among language syntaxes and assuring that $\sum_{j=1}^{n} w_{ij} = 1$. For two language syntaxes, such as natural text and source code, the language transition probability matrix becomes that of the above described basic HMM (where $w_{11} = p$ and $w_{22} = q$):

$$W = \begin{vmatrix} w_{11} & (1 - w_{11}) \\ 1 - w_{22} & w_{22} \end{vmatrix}$$

D. Model parameter estimation

The model parameters that need to be estimated are the transition probability matrix $T = \{t_{kl}\}$ and the language transition probability matrix $W = \{w_{ij}\}$. In a specific language context $t_{kl} = P(\pi_i = l | \pi_{i-1} = k)$ can be easily estimated by computing the number of transitions between two subsequent token symbols on training sample texts. For example, Fig. 3 shows the transition probabilities estimated on the Frankenstein novel, downloaded from the "Free ebooks – Project Gutenberg" (http://www.gutenberg.org). Fig. 4 shows the transition probabilities estimated inside a collection of C source code files extracted from the PosgreSQL source code repository (http://www.postgresql.org).

Language transition probabilities strongly depend on the nature of the text and the writing style adopted. This information may not be known in advance. The aim is to estimate how many transitions could happen between two language syntaxes. For example, in development mailing list messages usually there is no more than one source code fragment in a text message. If this is the case, we can assume, *a priori*, the transition probability from natural language text to source code approximated to $1/N$, where N is the number of tokens in the message. It could happens that the transition between two language syntaxes could never occur. This may be the case of stack traces and patches. By observing developers' emails, it is usual to notice that, after a stack trace is provided, the resolution patch is introduced with natural language phrases, such as *"Here is the change that we made to fix the problem"*. This leads to a null transition probability from stack trace to patch code. Other heuristics could be adopted to refine the estimation of transition probabilities that reflects specific properties or styles adopted [23].

IV. Empirical evaluation procedure

To evaluate the effectiveness of our approach we adopt the Precision and Recall metrics, known respectively also as Positive Predictive Value (PPV) and Sensitivity [24]. In particular, for each language syntax, i, we compute:

$$P_i = \frac{TP_i}{TP_i + FP_i}; \quad R_i = \frac{TP_i}{TP_i + FN_i}$$

where TP_i is the number of tokens correctly classified in the language syntax i; FP_i is the number of tokens wrongly classified as i; and FN_i is the number of tokens wrongly classified in a language syntax different from i. The total classification effectiveness is computed in term of classification accuracy, as:

$$ACC = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} (TP_i + FP_i)}$$

where n is the number of considered language syntaxes.

We set up three experiments: i) in the first (E1) we adopt public available html/latex textbooks with fully annotated source code fragments; ii) in the second (E2) we build a corpus of random text files pasting together natural language text, code fragments, and patches; and iii) in the third (E3) we evaluate the method in a real mailing list context.

A. **E1:** *Annotated textbooks*

In this experiment we use three textbooks related to computer programming topics containing Natural Language Text (NLT) and Source Code fragments (SRC). We exploit the fact that source code fragments are surrounded with html/latex tags, as shown in Table III, making the evaluation of classification accuracy feasible. The first is a programming book on Java with a lot of source code fragment examples; the second is a tutorial C programming with many coding style examples; the last is the documentation of an R package, Biostrings, containing usage examples on DNA string manipulation.

For each textbook, we use natural language transition probabilities estimated from the Frankenstein novel. Instead, for the source code HMM, we adopt a collection of source code files drawn from a software system that are representative of

TABLE III
ANNOTATED TEXTBOOKS ADOPTED IN EXPERIMENT **E1**.

	Text Book	Source code tagged with
1.	Thinking in Java, 3rd Edition (Bruce Eckel)	`<p class=code>...</p> <p class=codeInline>...</p>`
2.	Programming in C: A Tutorial (Brian W. Kernighan)	`<pre>...</pre>`
3.	R Biostrings package manuals	`\example{...} \code{...}`

TABLE IV
CROSS MAILING LIST VALIDATION - TRAINING CONDITIONS.

	PostgreSQL	Linux Kernel	Frankenstein novel	Patchwork comments	Patchwork Patches
E2.1	×			×	×
E2.2	×		×		×
E2.3		×	×		×
E2.4		×		×	×

the programming language encompassed in the textbook. In particular, we use JEdit (*version 3*) Java source code files for "Thinking in Java"; the Linux kernel (*version 3.9-rc6*) C source code files for "Programming in C tutorial"; and the *kernlab* R package source code files for "Biostrings package manuals".

B. **E2:** *Random generated text*

In this experiment we build an artificial corpus of textual files by combining three different kinds of language syntaxes: Natural Language text (NLT), Source Code (SRC), and Patches (PCH). A text file is generated by pasting together randomly selected pieces of information coming from the following repositories: i) source code C files from Linux kernel version, *3.9-rc6*, available at www.kernel.org; ii) patch proposals and natural language text from four Linux patchwork repositories available at https://patchwork.kernel.org. Natural language text is extracted from the patch textual comments after a manual purification of the selected sample that consists of eliminating automatic mailman directives (e.g., *from*, *reply*, *suggested by*) and inside code and stack trace fragments, if present. An example of random text adopted in this experiment is shown in Fig. 5. The Linux patchwork repository is organized in different sub-projects, and patches are attached as supplementary files separated from the body of the message. For the scope of this experiment we select 50 random patch messages from each of the following Linux patchwork projects:

- *linux-pci*, Linux PCI development list;
- *linux-pm*, Linux power management;
- *linux-nfs*, Linux NFS mailing list;
- *LKML*, Linux Kernel Mailing List.

We perform a cross-mailing list validation by leaving out 3 of the four considered mailing lists with the remaining one adopted for testing. Writing and programming styles could affect the transition probability estimation of the corresponding HMM. For example, a source code HMM estimated on C source files coming from two different software systems may not be the same because of different programming styles no matter whether the source code language is the same. The goal of this experiment is to evaluate to what extent different training condition affects the classification performance. In particular, we consider the following four training conditions, namely E2.1, E2.2, E2.3, and E2.4.

- *E2.1*: source code transition probabilities estimated on a collection of PostgreSQL (*version 9.2*) C source code files; natural language text and patches transition probabilities estimated on the leaved out 3 of the four considered mailing lists.
- *E2.2*: source code transition probabilities estimated on a collection of PostgreSQL (*version 9.2*) C source code files; natural Language transition probabilities estimated on the Frankenstein novel; and patches transition probabilities estimated on the leaved out 3 of the four considered mailing lists.
- *E2.3*: source code transition probabilities estimated on a collection of Linux kernel (*version 3.9-rc6*) C source code files not adopted for random text generation; natural Language transition probabilities estimated on the Frankenstein novel; and patches transition probabilities estimated on the leaved out 3 of the four considered

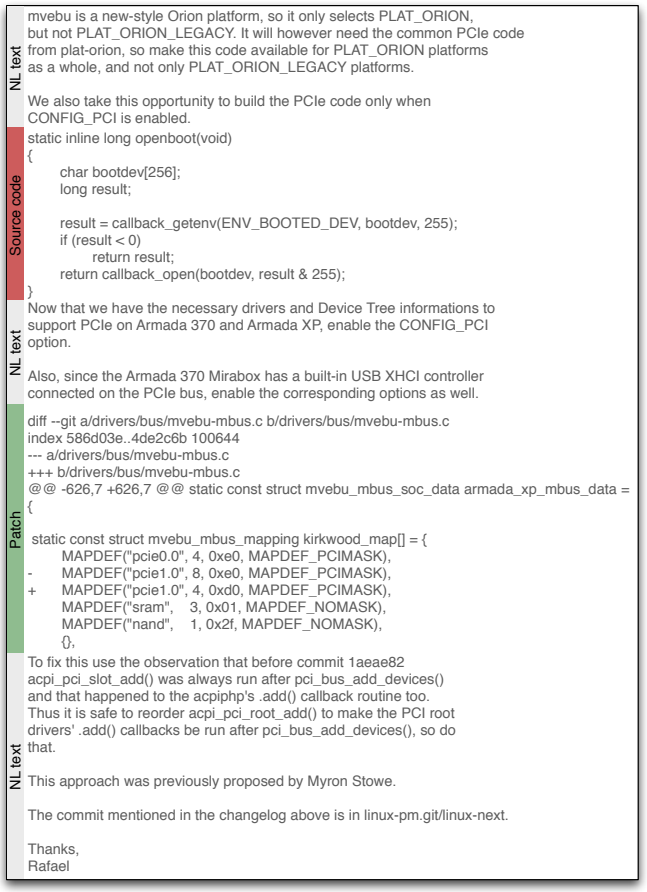


Fig. 5. An example of random generated text file.

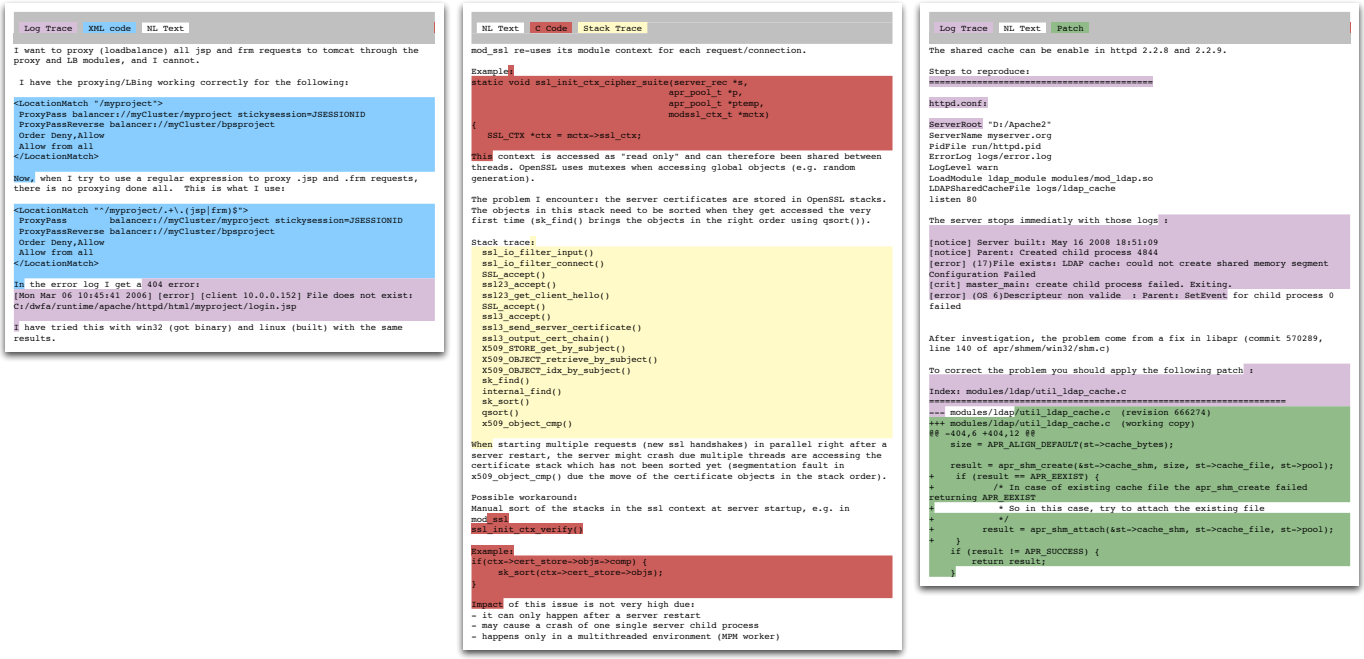


Fig. 6. Annotated message examples adopted for manual inspection.

mailing lists.

- *E2.4*: source code transition probabilities estimated on a collection of Linux kernel (*version 3.9-rc6*) C source code files not adopted for random text generation; natural language text and patches transition probabilities estimated on the leaved out 3 of the four considered mailing lists.

The training condition *E2.4* encloses the writing and programming styles adopted in the Linux Patchwork mailing lists in both natural language text and source code HMMs, as the samples are taken from the same domain adopted for testing. Instead, the training conditions *E2.1*, *E2.2*, and *E2.3* adopts source code and natural language text taken from a completely or partially different domain. Table IV summarizes how training sample are combined for each training condition.

C. E3: *Mailing list and bug report classification*

In this experiment we evaluate the approach on messages drawn from real mailing lists and bug tracking systems. For this purpose we use the Linux-CEPH filesystem mailing list and the Apache httpd bug tracking system. We consider a number of language syntaxes that is representative for a typical message exchanged in those systems. For Linux-CEPH filesystem mailing lists we consider three language syntax categories: Natural Language text (NLT), Source Code (SRC), and Patches (PCH). Apache httpd bug reports have a more complex structure that ensemble almost six language syntax categories: Natural Language text (NLT), Source Code (SRC), Patches (PCH), Stack Traces (STR), XML code (XML), and log messages (LOG). The source code category comprises: C language, shell scripts, and Javascript. Stack traces are C function call stacks with memory references and memory dumps. XML code usually refers to Apache httpd configuration files issues, and httpd server error response. Finally, LOG traces are typical error/log messages printed out by the Apache httpd server.

We try to reproduce a typical real usage condition by adopting the following evaluation protocol: i) we extract the last 200 messages from a mailing list/bug tracking; ii) one of the authors trains the model by using a half set of the extracted messages (i.e., 100 random messages leaved out from the corpus); iii) another author annotates the remaining messages with the trained model and counts the number of false positives, i.e., the sum of all wrong classified tokens, by manual inspection. This allows us to compute the total accuracy of classification. To this aim, we developed a tool that shows in a browser the annotated messages by using different background colors. Fig. 6 shows some annotated message examples adopted in the manual inspection.

V. Results

Table V reports results about experiment E1. The overall accuracy ranges from 0.86 to 0.97, attesting a promising performance on textbook encompassing different programming languages. The number of tokens in favor to natural language is higher in the first two textbooks and lower in the last one, making the dataset unbalanced in all cases. Precision and recall measures, and their harmonic mean (F-measure) are more appropriate for unbalanced datasets. The prediction accuracy of natural language outperforms source code in the first two textbook and is almost similar to source code for the last one. This leads us to conclude that the natural language model trained on the Frankenstein novel is almost accurate for modeling the natural language content of such textbooks. For source code, we can observe that the best performance

TABLE V
RESULTS ON ANNOTATED TEXTBOOKS (EXPERIMENT E1).

	Text Book	No. of tokens NLT	No. of tokens SRC	Correctly classified NLT	Correctly classified SRC	Accuracy	NLT Pr	NLT Rc	NLT Fm	SRC Pr	SRC Rc	SRC Fm
1.	Thinking in Java, 3rd Edition (Bruce Eckel)	65055	43044	62244	31440	0.867	0.843	0.957	0.896	0.918	0.730	0.813
2.	Programming in C: A Tutorial (Brian W. Kernighan)	9754	3429	9046	2667	0.888	0.922	0.927	0.924	0.790	0.778	0.784
3.	R Biostrings package manuals	2195	11577	1932	11439	0.971	0.933	0.880	0.905	0.978	0.988	0.982

has been obtained for the R language (Precision 0.978 and Recall 0.988), while the worst for the C language (Precision 0.790 and Recall 0.778). An average performance has been obtained for the Java language (Precision 0.918 and Recall 0.730). We believe that this is mainly due to the programming style adopted, as we show with the next experiment (E2). The source code HMM was trained with real software systems (PosgreSQL for the C language, Jedit for the Java language, and Biostrings for the R language). It is likely that the coding style adopted, usually in textbooks, to teach basic programming techniques differs from the coding practices adopted by senior developers.

Tables VI, VII, VIII, and IX show results of cross mailing list validation (E2) obtained with the training conditions E2.1, E2.2, E2.3, and E2.4 respectively. The first training condition (E2.1) uses, for source code, training samples coming from a different test set domain. The second training condition (E2.2) uses, for source code, training samples coming from a different test set domain (PostgreSQL development community). The third training condition (E2.3) uses, for natural language, training samples coming from a different domain (Frankenstein novel). The fourth training condition (E2.4) uses, for each language syntax (NLT, SRC, and PCH), training samples coming from the same domain of the test set (the Linux kernel development community).

As expected, the best performance is obtained under the training condition E2.4 (F-measure ranging between 0.87 and 0.99 in all mailing lists and for each language syntax), where natural language, source code, and patch syntaxes are modeled with examples coming from a domain that is closely related to the domain of examples we wish to classify. When the natural language HMM is trained on examples of a different domain (E2.2 and E2.3) the corresponding NLT prediction performance decreases (F–measure less than 0.8 in almost all cases). Instead, the NLT prediction performance persists on almost the same level in E2.1 and E2.4 (F-measure ranging from 0.86 to 0.96). A different behavior can be observed for source code HMM. When source code HMM is trained with examples of a different domain (E2.1 and E2.2) the decrement in prediction performance affects both source code and patches (F-measure drops to around 0.7 for SRC and 0.8 for PCH in almost all cases). This is because the source code snippets written by developers in the mailing list messages are usually confused with patches. Biasing in fact the detection of source code snippets by the source code HMM trained on PostgreSQL source code samples, as shows by the confusion matrices reported in Tables VII and VIII.

Table X reports results of experiment E3. The overall performance is terms of classification accuracy is 0.824 for

TABLE VI
CROSS MAILING LIST VALIDATION - TRAINING CONDITION E2.1.

	Classified as NLT	SRC	PCH	Precision	Recall	F-measure
	linux-LKML (Accuracy: 0.854)					
NLT	3356	117	162	0.953	0.923	0.938
SRC	34	2066	1251	0.883	0.617	0.726
PCH	132	156	5439	0.794	0.950	0.865
	linux-nfs (Accuracy: 0.859)					
NLT	2210	11	23	0.940	0.985	0.962
SRC	27	2453	1254	0.982	0.657	0.787
PCH	113	33	4259	0.769	0.967	0.857
	linux-pci (Accuracy: 0.841)					
NLT	4516	114	128	0.962	0.949	0.955
SRC	74	2649	1716	0.955	0.597	0.735
PCH	106	10	4206	0.695	0.973	0.811
	linux-pm (Accuracy: 0.756)					
NLT	1823	17	26	0.778	0.977	0.866
SRC	44	1353	1518	0.975	0.464	0.629
PCH	477	18	3338	0.684	0.871	0.766

TABLE VII
CROSS MAILING LIST VALIDATION - TRAINING CONDITION E2.2.

	Classified as NLT	SRC	PCH	Precision	Recall	F-measure
	linux-LKML (Accuracy: 0.789)					
NLT	2499	286	850	0.940	0.687	0.794
SRC	36	2092	1223	0.825	0.624	0.711
PCH	123	159	5445	0.724	0.951	0.822
	linux-nfs (Accuracy: 0.825)					
NLT	1924	35	285	0.964	0.857	0.907
SRC	21	2322	1391	0.971	0.622	0.758
PCH	50	34	4321	0.721	0.981	0.831
	linux-pci (Accuracy: 0.750)					
NLT	3224	327	1207	0.976	0.678	0.800
SRC	46	2636	1757	0.887	0.594	0.712
PCH	33	10	4279	0.591	0.990	0.740
	linux-pm (Accuracy: 0.746)					
NLT	1369	75	422	0.820	0.734	0.775
SRC	22	1499	1394	0.952	0.514	0.668
PCH	278	0	3555	0.662	0.927	0.772

TABLE VIII
CROSS MAILING LIST VALIDATION - TRAINING CONDITION E2.3.

	Classified as NLT	SRC	PCH	Precision	Recall	F-measure
	linux-LKML (Accuracy: 0.892)					
NLT	2493	339	803	0.946	0.686	0.795
SRC	19	3332	0	0.887	0.994	0.937
PCH	123	85	5519	0.873	0.964	0.916
	linux-nfs (Accuracy: 0.966)					
NLT	1946	53	245	0.975	0.867	0.918
SRC	7	3727	0	0.986	0.998	0.992
PCH	43	0	4362	0.947	0.990	0.968
	linux-pci (Accuracy: 0.887)					
NLT	3284	234	1240	0.986	0.690	0.812
SRC	13	4426	0	0.948	0.997	0.972
PCH	33	10	4279	0.775	0.990	0.869
	linux-pm (Accuracy: 0.908)					
NLT	1374	82	410	0.822	0.736	0.777
SRC	19	2896	0	0.972	0.993	0.982
PCH	278	0	3555	0.897	0.927	0.912

TABLE IX
CROSS MAILING LIST VALIDATION - TRAINING CONDITION E2.4.

	Classified as NLT	SRC	PCH	Precision	Recall	F-measure
	linux-LKML (Accuracy: 0.950)					
NLT	3280	183	172	0.955	0.902	0.928
SRC	21	3330	0	0.916	0.994	0.953
PCH	135	124	5468	0.970	0.955	0.962
	linux-nfs (Accuracy: 0.987)					
NLT	2229	7	8	0.947	0.993	0.969
SRC	9	3725	0	0.998	0.998	0.998
PCH	116	0	4289	0.998	0.974	0.986
	linux-pci (Accuracy: 0.972)					
NLT	4534	119	105	0.975	0.953	0.964
SRC	6	4433	0	0.964	0.999	0.981
PCH	108	47	4167	0.975	0.964	0.969
	linux-pm (Accuracy: 0.935)					
NLT	1831	10	25	0.785	0.981	0.872
SRC	19	2896	0	0.988	0.993	0.990
PCH	482	25	3326	0.993	0.868	0.926

Apache httpd bug reports and 0.974 for linux–CEPH mailing list. In Apache httpd bug reports, we detected up to six language syntaxes, while in the linux-CEPH mailing list we detected three. The complexity of bug reports makes the detection of languages more problematic. The manual inspection revealed that sometimes log messages are confounded with stack traces, while source code is almost detected correctly and also patch proposals. XML tags are correctly detected, however sometimes the approach fails because the free text contained between XML tags is recognized as natural language text.

VI. THREATS TO VALIDITY

This section describes threats that can affect the validity of the approach validation, namely *construct*, *conclusion*, *reliability*, and *external* validity.

A. Construct Validity

Threats to *construct* validity may be related to imprecisions in our measurements. In particular, metrics adopted to evaluate the approach performance may not measure the concepts we want to assess. The third experiment (E3) is particularly affected by this threat, as the difference in opinions on what qualifies as information encoded in the documents may affect the estimation of classification accuracy. For example, source code comments may or may not be considered as natural language text, or the presence of references to method calls in a sentence may or may not qualify as a code snippet. In computing the overall accuracy of experiment E3, we considered multi-line source code comments as natural language text, and we did not consider function/method calls mentioned in natural language sentences as real source code fragments.

TABLE X
RESULTS OF MANUAL VALIDATION (EXPERIMENT E3).

Mailing list	No. of tokens	False positives	Accuracy
Linux–CEPH filesystem discussion	66298	1687	0.974
Apache httpd bug reports	32991	5809	0.824

B. Conclusion Validity

Threats concerning the relationship between the treatment and the outcome may affect the statistical significance of the outcomes. We performed our experiments with representative samples letting us to obtain outcomes with an adequate level of confidence and error. In experiment E3, over 100,000 tokens were evaluated manually, while in experiment E2 we classified over 400 mailing list messages, and in experiment E1 we classified over 130,000 tokens from three textbooks.

C. Reliability validity

Threats to reliability validity concern the capability of replicating this study and obtaining the same results. Scripts and datasets adopted to run the experiments are available on http://www.rcost.unisannio.it/cerulo/dataset-scam2013.tgz.

D. External Validity

Threats concerning the generalization of results may induce the approach to exhibit different performance when applied to other contexts and/or different language syntaxes. In our experiments. we chose contexts with unrelated characteristics, software systems developed by separate communities, and in E3 we considered up to six language syntaxes. Moreover, in experiment E2 we performed a cross mailing list validation which is more powerful than a regular internal k-fold cross-validation. We are aware that a further empirical validation on a larger set of free text repositories would be beneficial to better support our findings.

VII. CONCLUSIONS AND FUTURE WORK

We introduced a method based on Hidden Markov Models to identify code information typically included in development mailing list and bug report free text. We performed a cross mailing list evaluation on text assembled with random pieces of natural language text, source code, and patches obtaining

promising results when the training samples are from the same mailing list information domain (accuracy ranging from 0.87 to 0.99). Furthermore, we performed a manual evaluation on a sample of real mailing list messages and bug reports obtaining, also in such a case, an overall classification accuracy ranging from 0.82 to 0.97, which is similar to the performance obtained by Bacchelli *et al.* in a comparable context [3].

The approach has, in fact, a performance level that in similar conditions is nearly equivalent to the most of literature methods generally based of regular expression and/or island parser. In addition, the method does not need the expertise required for the construction of the parser/regular expressions, as it learns directly from data. This last property allows the approach to be robust, especially with noisy data where regular expressions may fail, because of unexpected pattern variations from the original scheme on which the regular expression was designed.

The method opens for new opportunities in the context of software engineering free text mining. To this aim, we plan to investigate towards the following research directions:

- *HMM alphabet*. The token alphabet has been designed for general purposes. It can be improved by exploiting the language syntax to be detected. To this aim island parsers could be adopted to identify token patterns that may be meaningful and effective for a particular language syntax category. This will increase the HMM alphabet but could improve also the language detection capability.
- *High order HMM*. A HMM is also known as first-order Markov model because of a memory of size 1, i.e., the current state may depends only on a history of previous states of length 1. The order of a Markov model is the length of the history or context upon which the probabilities of the possible values of the next state depend, making high order HMMs strictly related to n-gram models. We believe that such a capability may be useful to model more precisely language syntaxes and capture, for example, specific programming styles, and the "naturalness" of software which is likely to be repetitive and predictable [23].
- *Evaluation dataset*. Last, but not least, we plan to evaluate the proposed approach on further datasets.

References

[1] D. Binkley and D. Lawrie, "Development: Information retrieval applications," in *Encyclopedia of Software Engineering*, 2010, pp. 231–242.

[2] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *Information Theory, IEEE Transactions on*, vol. 13, no. 2, pp. 260–269, 1967.

[3] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 375–384.

[4] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *Proceedings of the 2008 international working conference on Mining software repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 27–30.

[5] G. C. Murphy and D. Notkin, "Lightweight lexical source model extraction," *ACM Transactions on Software Engineering and Methodology*, vol. 5, pp. 262–292, 1996.

[6] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, Oct. 2002.

[7] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*. IEEE Computer Society, 2003, pp. 125–137.

[8] B. Ribeiro-neto and Baeza-yates, *Modern Information Retrieval*. Addison Wesley, 1999.

[9] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *Proceedings of the 11th IEEE International Software Metrics Symposium*, ser. METRICS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 29–.

[10] ——, "Supporting change request assignment in open source development," in *Proceedings of the 2006 ACM symposium on Applied computing*, ser. SAC '06. New York, NY, USA: ACM, 2006, pp. 1767–1772.

[11] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370.

[12] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 223–226.

[13] K. Spärck Jones, "Automatic summarising: The state of the art," *Inf. Process. Manage.*, vol. 43, no. 6, pp. 1449–1481, Nov. 2007.

[14] J. Tang, H. Li, Y. Cao, and Z. Tang, "Email data cleaning," in *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. New York, NY, USA: ACM, 2005, pp. 489–498.

[15] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza, "Content classification of development emails," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 375–385.

[16] M. Skounakis, M. Craven, and S. Ray, "Hierarchical hidden markov models for information extraction," in *In Proceedings of the 18th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 2003, pp. 427–433.

[17] L. E. Baum, T. Petrie, G. Soules, and N. Weiss, "A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains," *The Annals of Mathematical Statistics*, vol. 41, no. 1, pp. 164–171, 1970.

[18] X. Huang, Y. Ariki, and M. Jack, *Hidden Markov Models for Speech Recognition*. New York, NY, USA: Columbia University Press, 1990.

[19] T. Starner and A. Pentl, "Visual recognition of american sign language using hidden markov models," in *In International Workshop on Automatic Face and Gesture Recognition*, 1995, pp. 189–194.

[20] S. M. Thede and M. P. Harper, "A second-order hidden markov model for part-of-speech tagging," in *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, ser. ACL '99. Stroudsburg, PA, USA: Association for Computational Linguistics, 1999, pp. 175–182.

[21] B. Pardo and W. Birmingham, "Modeling form for on-line following of musical performances," in *Proceedings of the 20th national conference on Artificial intelligence - Volume 2*, ser. AAAI'05. AAAI Press, 2005, pp. 1018–1023.

[22] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Jul. 1998.

[23] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847.

[24] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

Fix-it: An Extensible Code Auto-Fix Component in Review Bot

Vipin Balachandran
VMware
Bangalore, India
vbala@vmware.com

***Abstract*—Coding standard violations, defect patterns and non-conformance to best practices are abundant in checked-in source code. This often leads to unmaintainable code and potential bugs in later stages of software life cycle. It is important to detect and correct these issues early in the development cycle, when it is less expensive to fix. Even though static analysis techniques such as tool-assisted code review are effective in addressing this problem, there is significant amount of human effort involved in identifying the source code issues and fixing it. Review Bot is a tool designed to reduce the human effort and improve the quality in code reviews by generating automatic reviews using static analysis output. In this paper, we propose an extension to Review Bot– addition of a component called Fix-it for the auto-correction of various source code issues using Abstract Syntax Tree (AST) transformations. Fix-it uses built-in fixes to automatically fix various issues reported by the auto-reviewer component in Review Bot, thereby reducing the human effort to greater extent. Fix-it is designed to be highly extensible– users can add support for the detection of new defect patterns using XPath or XQuery and provide fixes for it based on AST transformations written in a high-level programming language. It allows the user to treat the AST as a DOM tree and run XQuery UPDATE expressions to perform AST transformations as part of a fix. Fix-it also includes a designer application which enables Review Bot administrators to design new defect patterns and fixes. The developer feedback on a stand-alone prototype indicates the possibility of significant human effort reduction in code reviews using Fix-it.**

I. Introduction

Coding standard violations and defect patterns are abundant in checked-in code [1]. The presence of these source code issues leads to unmaintainable code and bugs in later stages of software life cycle, when it is more expensive to fix. Even though static analysis techniques such as peer code review are considered to be effective for solving this problem, as observed in [1], they are not highly effective in practice due to: *a*) significant human effort, *b*) difficulty in validating code against a lengthy coding standard and *c*) deprioritization of coding standard checks in favor of logic verification.

In [1], the author proposed a tool called Review Bot, which is an extension to the open-source code review tool Review Board [2]. Review Bot uses output of multiple static analysis tools to automatically post a code review, which covers majority of coding standard violations and common defect patterns. It also recommends appropriate code reviewers based on change history of source code lines. Even though automatic reviews reduce human effort and improve code review quality, developers still have to analyze the issues reported in automatic reviews, find out potential fixes and re-submit the code for review. Considering the fact that the fixes for a subset of coding standard violations and defect patterns can be automated, we propose a pre-processing step before creating an automatic review from static analysis output. This pre-processing step aims to fix automatically a large subset of source code issues. Post this step, there will be less number of source code issues for the developer to fix, which would improve the productivity. As indicated by the user study in [1], majority of source code issues in automatic reviews are due to the lack of understanding of coding standard rules and best practices and the developers are willing to address those issues. This implies the possibility of significant human effort reduction by the integration of automatic source code correction with code review.

Towards this goal of automatic correction of source code issues, we propose a new component called Fix-it in Review Bot. Fix-it internally maintains the source code as an Abstract Syntax Tree (AST) and uses AST transformations for auto-correction. It uses built-in fixes performing AST transformations to automatically fix the various issues reported by the auto-reviewer component in Review Bot. Fix-it allows the users to treat the AST as an XML DOM tree and supports extensibility in the form of custom defect patterns (AST structure pattern) written in XPath or XQuery [3]. It also supports extensibility in the form of custom fixes (performing AST transformations) written in Java or XQuery UPDATE expressions. Even though the current implementation supports only Java, the architecture is generic to support code written in any programming language which has an AST parser. Fix-it also includes a designer application to facilitate Review Bot administrators to create new defect patterns and fixes.

Refactoring is another task where significant human effort is involved. Even though there are refactoring tools such as TXL [4], Stratego/XT [5] etc., which support extensibility in terms of rules written in a tool-specific language, they are often under-utilized. Emerson et al. [6] reported that 90% of the refactorings are manual without using any refactoring tool. One of the reasons for this under-utilization is the learning curve involved in understanding these tools [7]. Fix-it supports custom fixes which can apply a particular automatic refactoring. Since Fix-it is used in a centralized way, majority

978-1-4673-5739-5/13/$31.00

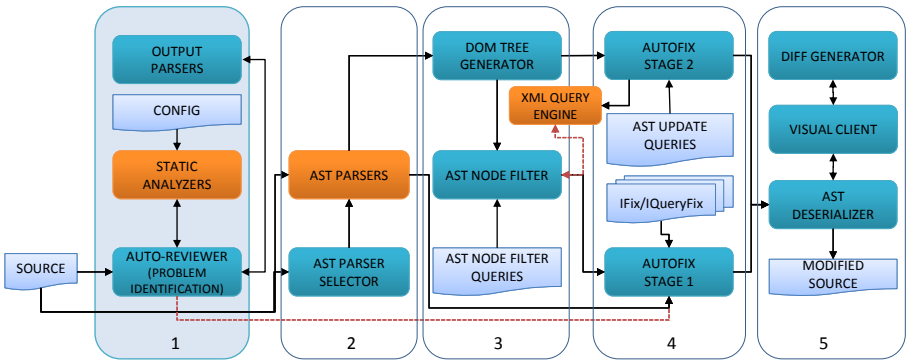


Fig. 1. Fix-it architecture

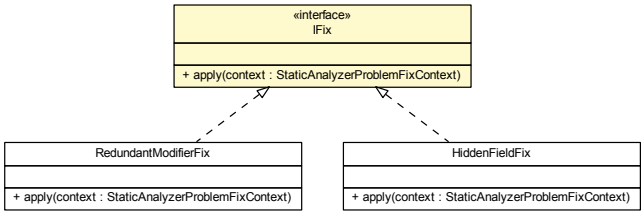


Fig. 2. Part of `IFix` type hierarchy

of the developers need not learn how to write the refactoring code. Further, we think it would be easier to write refactoring code in a widely adopted language such as Java or XQuery instead of a tool-specific language.

It could be argued that the developers themselves can run static analysis tools within their development environment and manually fix the issues before code review. As mentioned in [1], it might be difficult to enforce such a practice. The same argument applies for the use of refactoring tools before code review submission. Also, it would be easier to apply a common configuration for these tools when used in a centralized manner.

The design of Fix-it is motivated by the following observations.

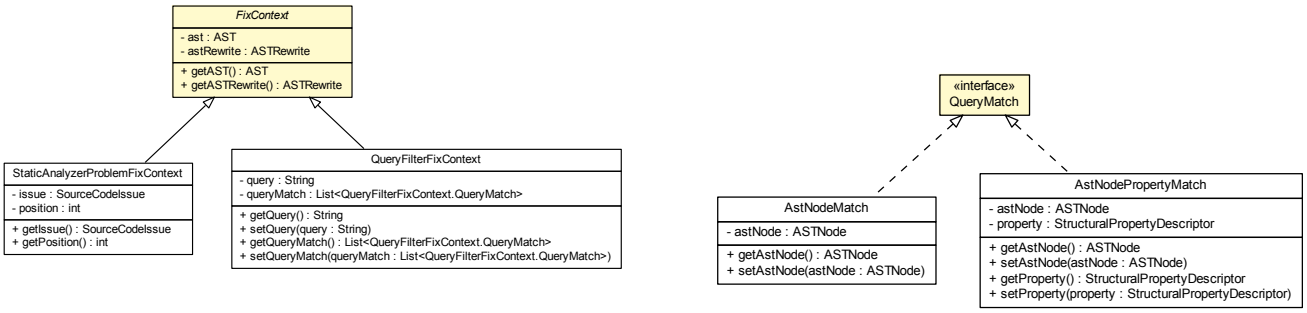
- AST is a convenient representation of source code to apply transformations.
- AST can be modeled as a DOM tree.
- XML and its query languages are widely adopted.

The rest of the paper is organized as follows. In Section II, we discuss Fix-it architecture followed by its extensibility in Section III. Section IV discusses the Fix-it designer application. Section V contains developer feedback on fixes possible for a subset of static analysis rules enabled in Review Bot. The related work is in Section VI and we conclude in Section VII.

II. Fix-it Architecture

The architecture of Fix-it is shown in Fig. 1. The modules within Fix-it and the modules in Review Bot which it interacts with are grouped into the following functional stages:

1) **Problem Identification**
The modules in this stage are already part of Review Bot. These modules are responsible for identifying the issues in the source code using automatic static analysis tools such as FindBugs [8]. The *Auto-Reviewer* invokes various *Static Analyzers*, which in-turn analyzes the source code for coding standard violations and defect patterns. The *Config* module supplies the necessary configuration (rules/checks to run, customized error/warning messages etc.) for the static analyzers. The output format varies from one static analyzer to another; hence the auto reviewer uses the *Output Parsers* module to convert various static analyzer output to a common format. It should be noted that a stand-alone version of Fix-it can be made by implementing these modules separately. In such a case, the auto-reviewer could be relabeled as *Problem Identification* module.
2) **AST Generation**
In this stage, the *AST Parser Selector* selects an AST parser based on the source language and constructs an AST. The current implementation uses `ASTParser` class in Eclipse JDT core [9]. In an alternate implementation, one could use a collection of ANTLR parsers [10] to support multiple programming languages.
3) **DOM Tree Generation and Node Filtering**
The *DOM Tree Generator* creates a DOM tree adapter which wraps the input AST in such a way that the updates on the DOM tree are translated to updates on the underlying AST. The *AST Node Filter* uses the *XML Query Engine* to evaluate XPath and XQuery expressions (*AST Node Filter Queries*) to filter out AST nodes which match user-defined defect patterns. The current implementation uses Saxon [11] as the query evaluation engine.
4) **Auto-correction**
This stage uses a collection of automatic fixes to fix some of the source code issues identified in Stage 1 and defect patterns identified in Stage 3. The *Autofix Stage*$_1$ module selects and applies a fix (written in Java) based on the source code issue type or AST node filter query. The *Autofix Stage*$_2$ module applies the fixes written as XQuery UPDATE expressions (*AST Update Queries*). It transforms the AST by evaluating the queries on the DOM tree wrapping the AST.



(a) `FixContext` type hierarchy (b) `QueryMatch` type hierarchy

Fig. 3. Fix context types

```
context.getUnit().accept(new ASTVisitor() {
   @Override
   public boolean visit(MethodDeclaration node) {
      for (Object obj : node.modifiers()) {
         Modifier m = (Modifier) obj;
         if (m.getStartPosition()==context.getPosition()) {
            context.getASTRewrite().remove(m, null);
            break;
         }
      }
      return false;
   }
});
```

Listing 1. Fix for Checkstyle's RedundantModifier check

```
@Override
public void apply(QueryFilterFixContext context) {
   for (QueryMatch m : context.getQueryMatch()) {
      assert m instanceof AstNodeMatch;
      ASTNode node = ((AstNodeMatch) m).getAstNode();
      assert node instanceof NumberLiteral;
      NumberLiteral literal = (NumberLiteral) node;
      context.getASTRewrite().set(literal, NumberLiteral.
           TOKEN_PROPERTY,
                "0" + literal.getToken(), null);
   }
}
```

Listing 2. Fix for "missing digit before decimal point"

5) **User Interaction**
This is the final stage– it generates the modified source code from the transformed AST (*AST Deserializer*) and presents the identified problems and the modified source code to the user (*Visual Client*). The visual client uses a *Diff Generator* to generate the side-by-side line diff and provides options to inspect the problems, corresponding fixes and to selectively apply a subset of fixes to the original source file.

III. Extending Fix-it

A. Support for New Static Analyzers

New static analyzers can be added in Stage$_1$ if it is possible to write an output parser for it. Most of the static analyzers produce output with some structure and an output parser is feasible in most cases. The current implementation of Review Bot supports only Java and uses FindBugs, Checkstyle [12] and PMD [13] for static analysis. It is a future work to add support for other languages; the main challenge in Fix-it will be to modify the fix stages to work with different types of ASTs. For example, supporting C++ implies a new DOM tree adapter for wrapping the C++ AST and a separate set of fixes.

B. Adding a New Fix

If there is no existing fix for a problem identified in Stage$_1$, users can write a fix in Java and add to the collection of fixes. This is done by implementing the `IFix` interface (Fig. 2) and associating the `.class` file with the problem ID. The details of the identified problem will be passed in as an instance of `StaticAnalyzerProblemFixContext` (Fig. 3(a)). As an example, Listing 1 is a snippet from the built-in `RedundantModifierFix` for the RedundantModifier check (to detect redundant modifiers; for example, `public`, `abstract` are redundant for a method declaration in an `interface`) in Checkstyle. Users can follow the same pattern of visiting interested node type (`MethodDeclaration` in `RedundantModifierFix`), checking if the node covers the problem identified and finally applying changes to the node. The Fix-it designer (Section IV) can be used to identify the AST node type associated with a specific problem.

C. Support for Custom Defects

Users can write XPath or XQuery expressions to detect custom defect patterns. For example, the following XPath expression when evaluated on a DOM tree adapter of an AST detects violations of the coding standard rule: "Floating point constants should always be written with a digit before the decimal point.".

```
//VariableDeclarationFragment[../..//@PRIMITIVETYPECODE="
     float" and fn:starts-with(.//NumberLiteral/@TOKEN, ".")
     ]//NumberLiteral
```

Filter queries like the above can be designed using the Fix-it designer application (Section IV-A). The result of the filter query is used to construct a fix context (instance of `QueryFilterFixContext`, see Fig. 3), which is then passed

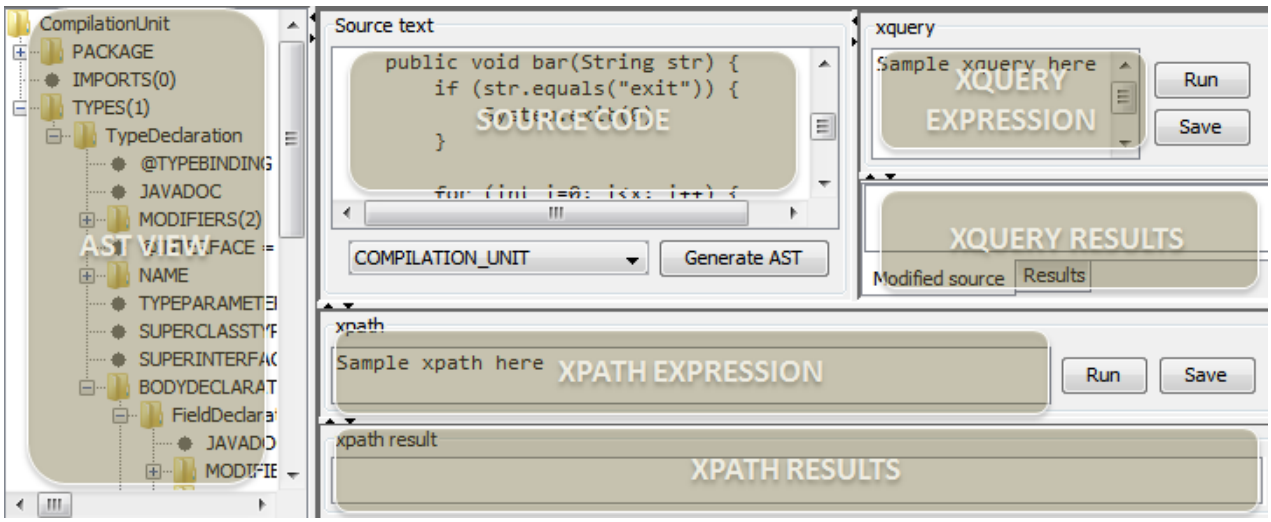


Fig. 4. Fix-it designer

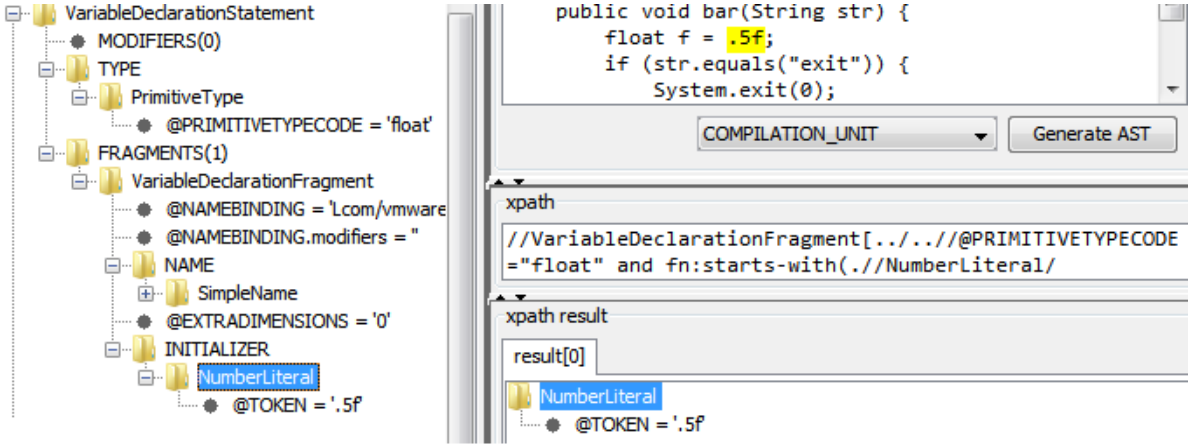


Fig. 5. AST view and filter based on XPath expression

to the AutoFix Stage$_1$ module where a fix (implementation of `IQueryFilterFix` which has a method `accept` with a single argument of type `QueryFilterFixContext`) is resolved based on the query string and applied. The fix in the case of the above filter query involves modifying the `TOKEN` property of the `NumberLiteral` AST node (Listing 2). If the defect pattern is complicated to be expressed in XPath or XQuery, users can input the query as '`/`' which matches the root of the AST and the corresponding `IQueryFilterFix` can traverse the AST to find the desired defect pattern and fix it.

D. Writing a Fix in XQuery

In cases where a fix is possible by simply deleting a node in the AST or by replacing the value of an AST node property, users can provide XQuery UPDATE expressions which modify the AST DOM tree. For example, the expression in Listing 3 provides a fix for the violation of coding standard rule which mandates that a constant identifier should not contain lower-case letters.

```
let $pattern := '^[A-Z_\$][A-Z0-9_\$]*$'
for $i in //FieldDeclaration
            [MODIFIERS/Modifier[@KEYWORD="final"]]
               //SimpleName[not(fn:matches(@IDENTIFIER, $
                    pattern))]
let $b := $i/@NAMEBINDING
let $id := fn:upper-case($i/@IDENTIFIER)
return (
  replace value of node $i/@IDENTIFIER with $id,
  for $j in //STATEMENTS//SimpleName[@NAMEBINDING=$b]
       return replace value of node $j/@IDENTIFIER with $id
)
```

Listing 3. Fix for "lower-case letters in constant identifier"

IV. Fix-it Designer

The designer application provides a GUI for designing AST node filter queries and AST DOM tree transformation queries. Fig. 4 shows the main window. The *AST view* pane displays the AST of the code snippet entered in the *Source Code* pane. For example, the AST sub-tree corresponding to the variable declaration `float f = .5f` is shown in Fig. 5.

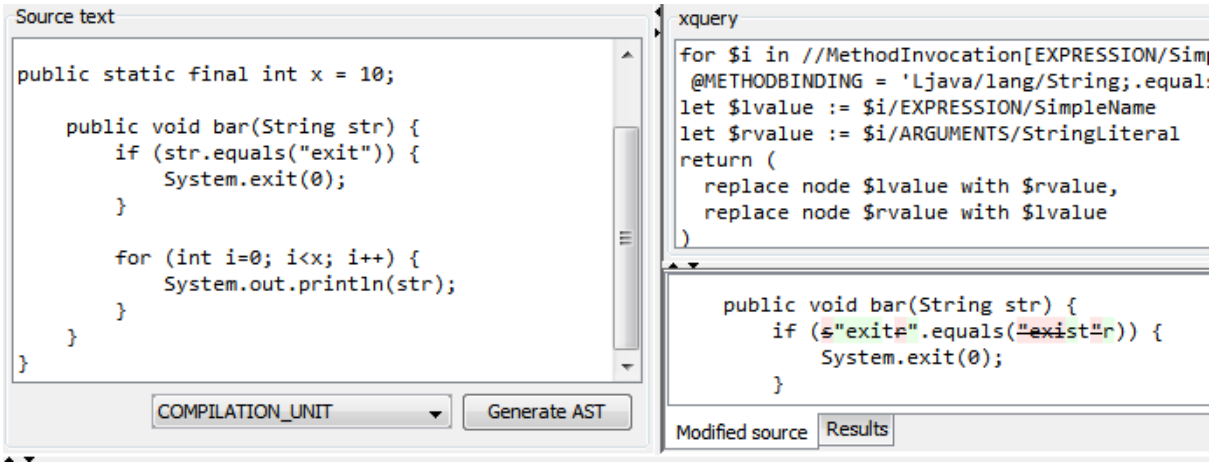


Fig. 6. AST transformation using XQuery

The highlighted AST node corresponds to the code fragment "`f = .5f`". Each of the AST nodes can have zero or more attributes (names prefixed with @) and zero or more child elements (nodes with name in upper-case letters). Each of the child elements can have zero or more AST nodes as its children; the child count is in parentheses after the node name. The `*BINDING` attributes are special attributes which do not have any structural significance and are used for cross-referencing between AST nodes. For example, any `SimpleName` node corresponding to the `float` variable `f` will have the same `NAMEBINDING` attribute value as that of its `VariableDeclarationFragment`. Listing 3 uses this idea to find the references of the constant identifier.

A. Designing AST Node Filter

A node filter can be written in either XPath or XQuery considering the AST as an XML DOM tree. Fig. 5 shows the result of evaluating the following XPath expression on the AST shown in the AST view pane.

```
//VariableDeclarationFragment[../..//@PRIMITIVETYPECODE="
    float" and fn:starts-with(.//NumberLiteral/@TOKEN, ".")
    ]//NumberLiteral
```

In this case, the user is interested in all floating point literals starting with a decimal point. The expression returns all variable declaration nodes with float type if the corresponding literal starts with a decimal point. Listing 2 is a fix corresponding to this filter.

If the filter query is complicated, XQuery FLWOR expression can be used instead of XPath. A FLWOR (acronym for `for, let, where, order by, return`) expression has 5 clauses. The `for` and `let` clauses generate a sequence of tuples. Some of these tuples are discarded by the optional `where` clause. The resultant tuples are then reordered by the optional `order by` clause. Finally, the `return` clause generates the output of the FLWOR expression. The following example shows the filter for unused method parameter. The FLWOR expression returns all variable declaration nodes which are method parameters if they are not referenced at least once in the method body.

```
for $i in //MethodDeclaration/PARAMETERS/
    SingleVariableDeclaration
let $b := $i/@NAMEBINDING
where (
   fn:empty($i/../../BODY/Block//SimpleName
      [@NAMEBINDING = $b])
)
return $i
```

B. Designing AST Update Queries

Fig. 6 shows the output of evaluating the following XQuery UPDATE expression.

```
for $i in //MethodInvocation[EXPRESSION/SimpleName and
    ARGUMENTS/StringLiteral and @METHODBINDING = "Ljava/
    lang/String;.equals(Ljava/lang/Object;)Z"]
let $lvalue := $i/EXPRESSION/SimpleName
let $rvalue := $i/ARGUMENTS/StringLiteral
return (
  replace node $lvalue with $rvalue,
  replace node $rvalue with $lvalue
)
```

This UPDATE expression fixes the violation of Checkstyle's EqualsAvoidNull check which ensures that the string literal is on the left side of `equals()` comparison with a `String` instance variable. On evaluating this expression, it finds all `String.equals` method invocation nodes with variable name on left side and string literal on right side and swaps them.

The next example provides a fix for the case where a floating point literal starts with a decimal point.

```
for $i in //VariableDeclarationFragment[../..//
    @PRIMITIVETYPECODE="float" and fn:starts-with(.//
    NumberLiteral/@TOKEN, ".")]//NumberLiteral
let $v := fn:concat("0", $i/@TOKEN)
return replace value of node $i/@TOKEN with $v
```

V. Results

A. Feedback on Checkstyle Fixes

As per the results in [1], on average, 86% of comments in an automatic review was generated by Checkstyle. In an attempt to estimate the effort reduction due to Fix-it, we requested a 10-year experience developer to provide feedback on possible Checkstyle fixes. The developer is proficient in Java and XML technologies and familiar with AST transformation

TABLE I
DEVELOPER FEEDBACK ON CHECKSTYLE FIXES

Checkstyle Module	#Rules enabled	#Rules with fixes	#Rules with fixes (advanced features)
Annotations	2	2	2
Block Checks	5	2	5
Class Design	6	4	5
Coding	30	16	19
Duplicate Code	1	0	0
Headers	1	1	1
Imports	6	4	4
Javadoc Comments	4	0	0
Metrics	5	1	1
Miscellaneous	6	3	6
Modifiers	2	2	2
Naming Conventions	11	5	10
Regexp	4	3	3
Size Violations	6	0	1
Whitespace	12	0	12
Sum	101	43	71

using Eclipse JDT core API. All the necessary details about Fix-it including ways to extend it was provided before the evaluation. The developer was requested to provide feedback on the following:

- For each of the Checkstyle rules enabled in Review Bot, is it possible to provide a fix with the current features in Fix-it?
- For rules where a fix is not possible, is it possible to provide a fix if ASTs corresponding to dependent files (with resolved bindings) are available? Can a fix be provided if the source text can be manipulated directly?

The feedback grouped by Checkstyle modules is given in Table I. The ability to fix 43% of issues with the current features and 70% with advanced features (planned for future) implies significant effort reduction during code review.

VI. RELATED WORK

The Eclipse IDE [14] has support for automatic code formatting and cleanup. It also supports automatic code correction in the form of quickfixes. Even though the formatting and cleanup features can be user configured, it is not possible to extend any of these features. Static analysis tools like FindBugs and Checkstyle support extensibility by writing new detectors in Java. The static analysis tool PMD supports extensibility via Java or by writing defect patterns in XPath, which is quite similar to the node filter use case in Fix-it. However, there is no support for automatic correction in any of these tools. AppPerfect [15] is a static analyzer which uses around 750 rules to detect various source code issues and automatically fixes 180 issues, but lacks extensibility. Also, as mentioned in [16], static analysis tools will receive limited use depending on how they are integrated with the development process. There are several refactoring tools which support user extensibility [4], [5], [17], [18]; however these type of tools are often under-utilized [6] and it might be difficult to enforce its usage as in the case of static analysis tools. The utilization of refactoring tools could be improved by a tight integration with the development process, like the way we proposed in this paper. In [19], the authors proposed a tool called `spdiff` to infer transformation specifications (semantic patches) from a collection of manually performed modifications in the source code. The inferred semantic patches can then be applied to other source files to make similar changes. It would be an interesting future work to extend Fix-it designer with the capability of inferring semantic patches which can be used as fixes.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a code auto-correction component called Fix-it in Review Bot. We discussed various ways to extend Fix-it. Through a user study, we have shown that Fix-it can reduce the human effort in code review to a significant extent. Support for writing complex refactorings involving multiple files and hooking into external frameworks during autofix stages are some of the future works.

VIII. DOWNLOADS

A stand-alone prototype version of Fix-it and the demonstration video can be downloaded at http://bit.ly/12fIiu6.

REFERENCES

[1] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 931–940.

[2] "Review Board," http://www.reviewboard.org/.

[3] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler, "Xquery 1.0 and xpath 2.0 formal semantics," *W3C recommendation*, vol. 23, 2007.

[4] J. R. Cordy, "Source transformation, analysis and generation in txl," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, ser. PEPM '06. New York, NY, USA: ACM, 2006, pp. 1–11.

[5] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/xt 0.17. a language and toolset for program transformation," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 52–70, Jun. 2008.

[6] E. Murphy-Hill, C. Parnin, and A. Black, "How we refactor, and how we know it," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 5 –18, jan.-feb. 2012.

[7] D. Campbell and M. Miller, "Designing refactoring tools for developers," in *Proceedings of the 2nd Workshop on Refactoring Tools*, ser. WRT '08. New York, NY, USA: ACM, 2008, pp. 9:1–9:2.

[8] "FindBugs," http://findbugs.sourceforge.net.

[9] "JDT Core Component," http://www.eclipse.org/jdt/core/index.php.

[10] "ANTLR," http://www.antlr.org/.

[11] "SAXON - The XSLT and XQuery Processor," http://saxon.sourceforge.net/.

[12] "Checkstyle," http://checkstyle.sourceforge.net/.

[13] "PMD," http://pmd.sourceforge.net/.

[14] "Eclipse," http://www.eclipse.org/.

[15] "AppPerfect," http://www.appperfect.com/.

[16] N. Ayewah and W. Pugh, "The google findbugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 241–252.

[17] H. Li and S. Thompson, "Let's make refactoring tools user-extensible!" in *Proceedings of the Fifth Workshop on Refactoring Tools*, ser. WRT '12. New York, NY, USA: ACM, 2012, pp. 32–39.

[18] K. Maruyama and S. Yamamoto, "Design and implementation of an extensible and modifiable refactoring tool," in *Proceedings of the 13th International Workshop on Program Comprehension*, ser. IWPC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 195–204.

[19] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo, "Semantic patch inference," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 382–385.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

sql-schema-comparer: Support of Multi-Language Refactoring with Relational Databases

Hagen Schink
Institute of Technical and Business Information Systems
Otto-von-Guericke-University
Magdeburg, Germany
hagen.schink@gmail.com

***Abstract*—Refactoring is a method to change a source-code's structure without modifying its semantics and was first introduced for object-oriented code. Since then refactorings were defined for relational databases too. But database refactorings must be treated differently because a database schema's structure defines semantics used by other applications to access the data in the schema. Thus, many database refactorings may break interaction with other applications if not treated appropriately. We discuss problems of database refactoring in regard to Java code and present *sql-schema-comparer*, a library to detect refactorings of database schemes. The sql-schema-comparer library is our first step to more advanced tools supporting developers in their database refactoring efforts.**

I. Introduction

Relational databases provide sophisticated functions for data persistence and retrieval [1]. In relational databases, data is stored according to a relational schema [2]. A relational schema predefines the logical structure of data.

Software applications are subject to change because of changing requirements and business conditions. However, adapting software applications may also require the adaption of the relational schema and vice versa.

Refactoring is a methodology to transform source code in such a way that the source-code's functionality is preserved [3]. Refactorings are specific code transformations and were first defined for object-oriented source code. But refactorings exist for relational databases too [4], [5]. Relational refactorings enable developers and database administrators to improve the structure of a relational schema without changing the informational semantics of the relational schema. Thus, a relational schema provides the same information before and after a relational refactoring.

Different APIs for many programming language exist to access relational databases. However, how APIs access data in the relational database depends on the structure defined by the relational schema. Thus, if the relational schema changes, e.g. because of refactoring, database access may be broken. Broken database access may only be noticed by integration tests or at runtime.

State-of-the-art IDEs provide information about syntactical changes or errors before integration or runtime tests take effect. Hence, the information can enable software developers to recognize and correct syntactic and some semantic errors already during development. We argue that information about syntactical changes in database schemes can ease the problems of database refactoring.

In the following we first present two techniques for accessing a relational database with the programming language Java. Then, we describe how database refactoring affects database access from Java. In Sec. III we give a more general explanation of problems resulting from database refactorings. Furthermore, we propose an approach to automatically detect database schema modifications affecting database interaction and present the *sql-schema-comparer* library that implements tool support for our approach. In Sec. IV we present related work before we conclude our work and present our future plans for the sql-schema-comparer library in Sec. V and Sec. VI.

II. Database Interaction and Refactoring

In this section we explain how data is retrieved from a relational database using the programming language Java. We also explain how changes of the relational schema affect the interaction between Java code and databases.

A. Examples of Database Interaction in Java

We consider two approaches to database interaction with Java: (1) a direct approach utilizing SQL statements and (2) an abstracted approach utilizing an object-relational mapper. The following examples are based upon two representatives of these approaches: (1) the Java Database Connectivity (JDBC) and (2) the Java Persistence API (JPA).

1) Direct Database Interaction with JDBC: With JDBC developers define database requests with SQL. In JDBC a database query returns an object of type `ResultSet` representing the SQL query result. Listing 1 presents a possible definition of a query that retrieves all first names of employees starting with the letter *M*.

Listing 1: JDBC: Parameterized query for first names

```
1 String stmt = "SELECT firstname FROM employees "
2               + "WHERE firstname LIKE ?";
3 PreparedStatement query = con.prepareStatement(stmt);
4
5 query.setString(1, "M%");
6 ResultSet result = query.executeQuery();
```

Listing 2: Annotated class `Department`

```
1  @Entity
2  @Table(name="departments")
3  public class Department implements Serializable {
4
5    /* Snip */
6
7    private int id;
8    private String name;
9
10   public void setId(int id) { this.id = id; }
11
12   @Id
13   public int getId() { return id; }
14
15   public void setName(String name) { this.name = name; }
16
17   public String getName() { return name; }
18 }
```

Because of missing abstraction, JDBC requires detailed knowledge about the relational schema. For instance, in Listing 1 Line 5, the passed parameter must be an instance of class `String` because column `firstname` stores characters. Thus, a software developer must enforce type correctness.

2) Abstracted Database Interaction with JPA: JPA provides means to define a mapping between relational tables and object-oriented classes. This mapping is called *object-relational mapping* (ORM). Thus, in contrast to JDBC, developers receive objects instead of plain data from a relational database.

Listing 2 shows an implementation of a class representing a department. In JPA, each class annotated with `@Entity` becomes part of the ORM, thus, class properties are mapped upon columns in the mapped relational table. Hence, regarding Listing 2, we expect a table `departments` (Line 2) with the two columns `id` (Line 10 and 13) and `name` (Line 15 and 17)[1].

B. Database Refactoring and Interaction

The Database Refactoring website lists 69 database refactorings.[2] In the following we give examples on how database refactorings affect database interaction. To keep the following discussion concise, we focus only on two database refactorings, namely the Split Column Refactoring and the Introduce Column Constraint Refactoring.

1) Split Column Refactoring: The purpose of the Split Column Refactoring is to separate information that is combined in one table column. Let's assume we have a database column `amount` storing an amount of money and its currency. We want to split the column into two columns `amount` and `currency` to make queries for the currency easier. Existing JDBC queries like in Listing 3 need to be adapted because the column `amount` does not hold any currency information after refactoring. A possible adaption is given in Listing 4.

With JPA, we need to extend the ORM to map to the new column `currency`. Therefor, we need a new getter/setter pair, e.g. `getCurrency`/`setCurrency`.

Listing 3: Query accounts by currency

```
1 SELECT amount FROM accounts
2 WHERE amount LIKE '%EUR';
```

Listing 4: Modified query for accounts by currency

```
1 SELECT currency FROM accounts
2 WHERE currency = 'EUR';
```

2) Introduce Column Constraint Refactoring: The purpose of the Introduce Column Constraint Refactoring is to define a new column constraint to increase data quality. Let's assume we have a database table `accounts` with a column named `type`. The column `type` distinguishes different kinds of accounts by an uppercase letter. To ensure that only valid types are stored in table `account`, we can apply a column constraint like in Listing 5.

Listing 5: *CHECK* constraint on column `type`

```
1 'type' text CHECK(type = 'A' OR type = 'B' OR type = 'C')
```

A column constraint helps users with direct database access to preserve data quality. But applications accessing a table with constraints may not know about the restrictions enforced by the constraints. Such applications and their users are not able to handle the column constraints adequately. Developers of such applications may have to adapt the user interface and the applications database access to allow only such account types introduced by the Introduce Column Constraint Refactoring. Thus, the introduction of a column constraint by a refactoring may result in considerable effort to adapt related applications independent of the kind of database interaction.

III. Semantics-Preservation in Databases

In Sec. II-B, we showed that a database refactoring may result in considerable effort to adapt interacting software applications. We argue that this is the result of different meanings of semantics-preservation for software applications and database

[1] Because method `getId()` (Line 13) is annotated with `@Id`, JPA uses property access. Thus, columns are defined by getter/setter pairs.

[2] http://databaserefactoring.com/, 03/23/2013

schemes. This difference of semantics-preservation originates from a different semantics idea that is expressed in the term *informational semantics* [4].

The informational semantics of a database is preserved if the same information is available before and after a database transformation. We call a database transformation preserving the informational semantics a database refactoring. For instance, the Split Column Refactoring (c.f. Sec. II-B1) preserves the informational semantics because after the refactoring the information from the original column is just distributed over two columns. The Introduce Column Constraint Refactoring (c.f. Sec. II-B2) preserves the informational semantics because the refactoring just states explicitly an implicit assumption about a column's content.

However, although database refactorings preserve a database's informational semantics, application developers still may need to adapt the application code to the schema modifications. The question how the application code needs to be adapted may only be answered by the application developers themselves. Thus, application developers need a tool that makes schema modifications visible in such a way that application developers can decide whether and how to adapt the application code to the schema modification. In the following, we discuss which schema modifications affect the interaction with databases and, thus, need to be considered by application developers. Then, we present `sql-schema-comparer`, a library to compare databases schemes, that provides functions to detect database modifications.

A. Schema Modifications Affecting Database Interaction

In Sec. II-B we discussed the Split Column and Introduce Column Constraint Refactoring. In the following, we discuss general database refactoring categories in contrast to specific database refactorings.

Database refactorings can be distinguished by five categories: *Architectural*, *Data Quality*, *Performance*, *Referential Integrity*, and *Structural* [4]. To keep the following discussion concise, we consider the categories Data Quality and Structural.

In simple terms, with Data Quality Refactorings we set constraints on a column's content and with Structural Refactorings we modify a tables structure. Thus, Data Quality and Structural Refactorings may affect interacting applications differently.

1) Data Quality Refactorings: Data Quality Refactorings modify column data and column constraints. Except for the Consolidate Key Strategy Refactoring no columns are created or removed. Thus, if one of the following conditions is fulfilled database queries still work in an application:

1) a column constraint is removed,
2) application provides data conforming with constraints,
3) modified data formats can be processed by the application.

The first and the second condition are related: An application needs to comply with column constraints or it will not be able to insert/update data after refactoring. The first case is trivial because any data of the column's type will comply to non-existing constraints. The third condition points to issues after data format alignment: If the data format is changed, an application may not be able to process a new format, although the application may be able to insert/update data. For instance, assume we normalize telephone numbers stored in a database. After normalization applications may not be able to process the new format, although unnormalized data can still be entered into the database.

Data Quality Refactorings may not affect database interaction on first sight because refactorings may only affect certain corner cases. Therefore, after a Data Quality Refactoring, developers have to check the validity of all interactions.

2) Structural Refactorings: Structural Refactorings modify the structure of single tables as well as the whole database schema. Apart from the Introduce Calculated Column and Introduce Surrogate Key Refactoring, all Structural Refactorings rename, move, or drop schema elements, i.e. columns or tables. Therefore, after a structural refactoring an application will not be able to reference the modified schema element. Thus, Structural Refactorings affect database interaction if applied on a schema element that is accessed by an application.

B. SQL Schema Comparison

In Sec. III-A we discussed the effects of Data Quality and Structural Refactoring in regard to database interaction. In the following, we discuss important features a software tool has to provide for minimal support of developers adapting the interacting application code.

Effects of Data Quality Refactoring are not visible on first sight, thus, developers have to make qualified decisions about the impact of a Data Quality Refactoring. Therefor, tool support needs to find and present modified column constraints. Tool support for Structural Refactoring should be able to detect missing schema elements and to reveal the new position of moved schema elements. Therefor, a tool needs to be able to reveal the connection to other tables (foreign key relations) to show information about the new schema element's position.

Changes due to refactoring may be revealed by database schema comparison, thus, comparing the initial and the refactored database schema. In general, techniques for schema comparison are summarized under the term *schema matching* [6].

We state that adaption of database interaction after database refactoring is a manual task that is not likely to be completely automated in all cases. Because the adaption is a manual task, gathering information for adapting database interaction should be automated to provide a benefit. Different methods exist to support the comparison of two database schemes [6]. But these methods may depend on user interaction to complete a schema match, because these methods assume arbitrary schema modifications. But a database refactoring is a well defined schema modification that can consist of the following steps:

1) Create, Drop, Rename Table
2) Create, Drop, Rename, Move Column
3) Add, Drop, Change Column Constraint

A software tool can recognize single steps automatically if it has access to each single schema state after a step has been applied.

C. The sql-schema-comparer Library

We present *sql-schema-comparer*, a software library to compare and check SQL schemes.[3] The library allows to compare:

1) two SQL schemes,
2) an SQL statement and an SQL schema

with each other. The library provides support for SQLite statements and schemes and elementary support for JPA annotations in Java files.

The schema comparison and statement validation is performed on a graph representation. The schema comparison algorithm is able to automatically detect small schema modifications as described in Sec. III-B.

We argue that the sql-schema-comparer library provides the necessary functionality to build the basis for more sophisticated tools that, additionally, provide visual hints to inform developers about SQL schema refactorings.

1) Implementation Details: The library translates schema and statement information into a graph representation using the *jgrapht* graph library.[4] The graph contains nodes of type `ISqlElement`[5] implemented in the library and edges of type `DefaultEdge` (jgrapht). Tab. I lists all graph elements and their implementation in the sql-schema-comparer library.

Database	Graph	sql-schema-comparer
Table	Node	`SqlTableVertex`
Column	Node	`SqlColumnVertex`
Table-column relation	Edge	`TableHasColumnEdge`
Foreign-key relation	Edge	`ForeignKeyRelationEdge`

Table I: Graph elements and their respective Java types.

Nodes can represent tables or columns. Column nodes contain a list of column constraint information. Edges can represent a table-column relationship or a foreign-key relationship. Foreign-key relationships contain a relation between the referencing column to the referenced table and a reference to the foreign-key column in the referenced table. Fig. 1 shows an example graph that contains two table nodes *t1* and *t2* and four column nodes *c1*, *c12*, *c21*, and *c2*. The columns *c1* and *c12* are part of table *t1* and the columns *c21* and *c2* are part of table *t2*. Additionally, the graph contains an edge representing a foreign-key relationship between column *c12* and table *t2* by foreign-key column *c21*.

The sql-schema-comparer library is able to create schema graphs of SQLite databases, SQL SELECT statements, and JPA entity source files (Java classes annotated with `@Entity` from JPAs namespace `javax.persistence`). Tab. II lists all front-end implementations available in the sql-schema-comparer library. Fig. 2 shows the schema graph creation process. Note that, optionally, we are able to pass a database schema graph to the SQL Statement front-end. The SQL Statement front-end uses the database schema graph to augment the SQL statement's schema graph with type information that are not present in an SQL SELECT statement.

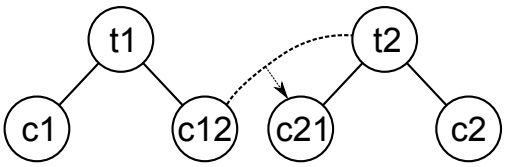


Figure 1: Database schema with a Foreign Key Relationship.

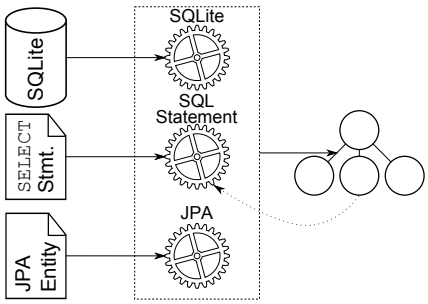


Figure 2: Schema graph creation process.

Source Type	sql-schema-comparern Front-end
SQLite	`SqliteSchemaFrontend`
`SELECT` statement	`SqlStatementFrontend`
JPA entity	`JPASchemaFrontend`

Table II: Available front-ends for schema graph creation.

The SQL schema comparison algorithm compares two SQL schemes by comparing each node of one schema graph with the nodes in the other schema graph respecting the table-column and foreign-key relationship. Taking into account our assumption, after refactoring we find a mismatch for at most one node. For each matching column node the algorithm compares the list of column constraints to detect constraint modifications.

The SQL statement validation compares an SQL statement with an SQL schema. The validation tries to match each node of the statement graph with a node of the schema graph. Additionally, if the algorithm cannot find a matching column node (a node with the same identifier and table node), the algorithm tries to find the column in referenced tables. Therefor, the algorithm checks for each column that matches the missing columns identifier if it is reachable via a foreign-key relationship. Fig. 3 shows an example schema graph with column node *c12* moved from table node *t1* to table node *t2*. The algorithm first looks for all column nodes matching the identifier *c12*. In respect to Fig. 3, the algorithm finds two possible match candidates in table *t2* and *t3*. Next, the algorithm checks if a foreign-key relationship exists. Because only table node *t2* is related to table node *t1* by a foreign-key relationship, column node *c12* of table node *t2* is identified as moved column.

[3] https://github.com/hschink/sql-schema-comparer

[4] http://jgrapht.org

[5] We skip the full namespace of sql-schema- comparer's internal types and mark classes of the jgrapht graph library accordingly.

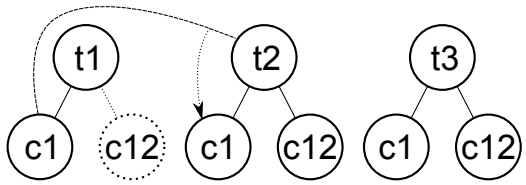


Figure 3: Resolving moved nodes by foreign-key relationships.

According to the classification by Rahm et al. [6], the SQL schema comparison algorithm and SQL statement validation possess the following properties. Matching granularity is different for tables and columns: Table nodes of on schema are only compared with table nodes of the other schema, thus, table comparison does not involve column comparison. In contrast, the comparison of column nodes also involves table node comparison to detect moved columns. Thus, the algorithms combine *element-level matching* (matching only table nodes with each other) as well as *structure-level matching*. In addition to the node's type, matching involves the node's identifier, i.e., table and column name. Because the matching algorithm combines element-level and structure-level matching and name matching, the algorithm can be called a *hybrid matcher*. Finally, because we assume that only a single node changed between two schemes, we restrict matching cardinality to 1:1.

2) Use Cases and Examples: The sql-schema-comparer library allows to compare two SQL schemes or an SQL statement with an SQL schema. In the following we describe the libraries application within a third-party application. For first tests you may use the library from the command-line.[6] Additionally, unit tests describe possible uses cases and expected results.[7]

a) Comparing SQL Schemes: For schema comparison we need to create a graph representation for each schema. Schema graphs are instances of class `SimpleGraph` (jgrapht) with the node type `ISqlElement` and edge type `DefaultEdge` (jgrapht). We may create instances manually or by parsing SQLite files. The latter is described in the following.

First of all we need to create an instance of the SQLite front-end as described in Listing 6, Line 1. The method `createSqlSchema()` of the front-end returns the schema instance for the SQLite file (Listing 6, Line 2).

The schema comparison is implemented in class `SqlSchemaComparer` which takes two schema instances on construction (Listing 7, Line 1). The field `comparisonResult` of an `SqlSchemaComparer` instance contains the match result. The match result contains the affected tables, columns, and column constraints. Because a comparison can only detect one refactoring step, the result can only contain one table or column.

[6]For details on the command-line usage refer to https://github.com/hschink/sql-schema-comparer#commandline-usage.

[7]Unit tests can be found in the project's subdirectory `src/test`.

Listing 6: Create a schema instance of an SQLite file

```
1 ISqlSchemaFrontend frontend = new SqliteSchemaFrontend(
      DATABASE_FILE_PATH);
2 Graph<ISqlElement, DefaultEdge> schema = frontend.
      createSqlSchema();
```

Listing 7: Compare two schema instances

```
1 SqlSchemaComparer comparer = new SqlSchemaComparer(
      schema1, schema2);
2 SqlSchemaComparisonResult result = comparer.
      comparisonResult;
```

b) Comparing an SQL Statement with an SQL Schema: To compare an SQL Schema with an SQL statement we need to extract a schema instance for the database (see Listing 6) as well as for the SQL statement. Latter is possible with the `SqlStatementFrontend` front-end as shown in Listing 8, Line 1. The front-end takes at least an SQL statement. Additionally, it is possible to pass a database schema, so type and constraint information can be extracted for the columns mentioned in the SQL statement.

The database and statement comparison is implemented in class `SqlStatementExpectationValidator` (Listing 9, Line 1). The class takes a database schema instance on construction. Calling the method `computeGraphMatching()` with the SQL statement schema returns a matching result (Listing 9, Line 2). The result contains missing tables or columns and columns that appear to be moved to different tables.

IV. Related Work

Two fields of research are addressed by this paper: multi-language refactoring and schema matching. In the following, we discuss both fields separately.

A. Multi-Language Refactoring

The *Cross-Language Link* (XLL) framework allows multi-language refactoring if language or technology-specific refactorings are available [7]. It is not discussed how the XLL framework can support developers if a refactoring exists for a database schema but not for object-oriented code or vice versa. Furthermore, the XLL framework has no support for semi-refactored states, e.g., the database is refactored but the object-oriented code does not match with the refactored schema.

Many approaches exist that describe multi-language refactoring for programming languages of a common paradigm or with a common technological base. In [8]–[11] approaches are presented that use source-code meta-models, namely FAMIX, MOOSE, UML, and the Common Meta-Model to describe and modify code of object-oriented programming languages. The common idea of these approaches is to describe a refactoring once for the common meta-model and to re-use it on the

Listing 8: Create a schema instance of an SQL statement

```
1 ISqlSchemaFrontend frontend = new SqlStatementFrontend(
      STATEMENT, null);
2 Graph<ISqlElement, DefaultEdge> schema = frontend.
      createSqlSchema();
```

Listing 9: Compare a database and a statement schema

```
1 SqlStatementExpectationValidator validator = new
      SqlStatementExpectationValidator(dbSchema);
2 SqlStatementExpectationValidationResult result = validator.
      computeGraphMatching(statementSchema);
```

different concrete languages. Because of this design syntactical elements of languages of different paradigms cannot be described and, thus, not be refactored.

B. Schema Matching

An overview of schema matching approaches are given in [6], [12]. The presented schema matchers implement a general approach to schema matching. Thus, in contrast to the sql-schema-comparer library, they allow to compare arbitrary schemes with each other. Furthermore, because the presented schema matchers only determine match candidates, user interaction is still necessary to select valid match candidates. To the best of our knowledge no schema matching approach exists dedicated to the specifics of database refactoring.

V. Conclusion

Relational databases are an integral part of many modern software applications. Database interfaces exist for the majority of programming languages. However, modern IDEs provide no means for developers to recognize database refactorings or even simple schema modifications. Thus, developers are on their own to detect SQL schema modifications and to adapt their applications to a modified schema.

In this paper, we discussed the influence of database refactorings to interacting software applications. For this purpose we discussed two database refactorings in detail: (1) Split Column and (2) Introduce Column Constraint Refactoring. Furthermore, we discussed a minimal feature set that a tool should provide to support developers adapting their software application to a refactored database schema. Then, we concluded that we may only be able to automate schema modification detection under certain assumptions. Finally, we presented the *sql-schema-comparer* library that allows to detect schema modifications and to validate SQL statements against SQL schemes.

VI. Future Work

Using the *sql-schema-comparer* library requires a reasonable amount of user interaction and, thus, is not practically usable. In our ongoing work we focus on integrating the sql-schema-comparer library into the Eclipse IDE. The idea for Eclipse integration is to provide database refactoring support for Java for two use cases: (1) direct database access with the *Java Database Connectivity* (JDBC) and (2) abstract database access with the *Java Persistence API* (JPA).

With JDBC we use SQL statements (c.f. Sec. II-A1) and with JPA we use JPA entities (c.f. Sec. II-A2) to interact with databases. The sql-schema-comparer library is able to compare SQL statements and JPA entities with a given SQLite database. Thus, in a future Eclipse plug-in, we are able to not only highlight those SQL statements and JPA entities that do not match with a given database schema but to give more detailed information about the mismatch. In a first step we may let the developer select the SQL statement or JPA entity as well as the SQLite database file that need to be compared.

Assume that a developer has access to each single change of an SQLite file's schema.[8] With the sql-schema-comparer library we are able to create a change history of the database's schema. The change history may provide useful information for adapting mismatching SQL statements or JPA entities to the new database schema.

References

[1] E. F. Codd, "The 1981 ACM Turing Award Lecture Relational Database : A Practical Foundation for Productivity," vol. 25, no. 2, 1982.

[2] ——, "A relational model of data for large shared data banks. 1970." *M.D. computing : computers in medical practice*, vol. 15, no. 3, pp. 162–6, 1970. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/9617087

[3] M. Fowler, *Refactoring: Improving the Design of existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[4] S. Ambler, *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[5] S. Ambler and P. Sadalage, *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[6] E. Rahm and P. a. Bernstein, "A survey of approaches to automatic schema matching," *The VLDB Journal*, vol. 10, no. 4, pp. 334–350, Dec. 2001.

[7] P. Mayer and A. Schroeder, "Cross-Language Code Analysis and Refactoring," *SCAM*, pp. 94–103, 2012.

[8] S. Ducasse, M. Lanza, and S. Tichelaar, "MOOSE: An Extensible Language-Independent Environment for Reengineering Object-Oriented Systems," *CoSET*, 2000.

[9] S. Tichelaar, "Modeling Object-Oriented Software for Reverse Engineering and Refactoring," Ph.D. dissertation, University of Berne, Switzerland, 2001.

[10] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer, "Towards Automating Source-Consistent UML Refactorings," *UML*, 2003.

[11] D. Strein, H. Kratz, and W. Lowe, "Cross-Language Program Analysis and Refactoring," *IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 207–216, 2006.

[12] P. Shvaiko and J. Euzenat, "A survey of schema-based matching approaches," *Journal on Data Semantics IV*, no. October 2004, 2005.

[8] With SQLite the current schema can be preserved by copying the file itself.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

A Relational Symbolic Execution Algorithm for Constraint-Based Testing of Database Programs

Michaël Marcozzi*
Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
Namur, Belgium
Email: michael.marcozzi@unamur.be

Wim Vanhoof
Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
Namur, Belgium
Email: wim.vanhoof@unamur.be

Jean-Luc Hainaut
Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
Namur, Belgium
Email: jean-luc.hainaut@unamur.be

***Abstract*—In constraint-based program testing, symbolic execution is a technique which allows to generate test data exercising a given execution path, selected within the program to be tested. Applied to a set of paths covering a sufficient part of the code under test, this technique permits to generate automatically adequate test sets for units of code. As databases are ubiquitous in software, generalizing such a technique for efficient testing of programs manipulating databases is an interesting approach to enhance the reliability of software. In this work, we propose a relational symbolic execution algorithm to be used for testing of simple Java methods, reading and writing with transactional SQL in a relational database, subject to integrity constraints. This algorithm considers the Java method under test as a sequence of operations over a set of constrained relational variables, modeling both the database tables and the method variables. By integrating this relational model of the method and database with the classical symbolic execution process, the algorithm can generate a set of Alloy constraints for any finite path to test in the control-flow graph of the method. Solutions of these constraints are data which constitute a test case, including valid content for the database, which exercises the selected path in the method. A tool implementing the proposed algorithm is demonstrated over a number of examples.**

I. Introduction

Testing [1], [2] constitutes the primary approach to improve the reliability of software. This motivates [3] much research to develop relevant automated techniques to test efficiently all aspects of software. In particular, two successful white-box approaches have emerged [4] to generate automatically adequate test data for testing units of code, with regard to their expected functions. In search-based testing (e.g. [5]), optimization heuristics are used to search the input space of the program for adequate test data. In constraint-based testing (e.g. [6], [7], [8], [9], [10], [11]), the program is transformed into constraints whose solutions are adequate test data.

*F.R.S.-FNRS Research Fellow

978-1-4673-5739-5/13/$31.00

Among constraint-based test data generation strategies, classical symbolic execution [12] works in three successive steps. First, a finite set of finite paths in the control flow graph [13] of the program are selected for testing. Several selection algorithms have been proposed (see e.g. [14]) to select a set of paths in the program which satisfies a given code coverage criterion [13]. Secondly, each selected path is symbolically executed [15], [16], [17], [12] to build a set of path constraints over the program inputs. These constraints are such that when the program is executed with respect to input values satisfying them, the execution is guaranteed to follow the path from which they were generated. In order to build these constraints, the symbolic execution algorithm considers the static single assignment form of the program and expresses the control dependencies imposed by the path to be tested. Thirdly, the constraints generated for each of the selected paths are solved, producing each time a new test case, which will exercise a different path among those initially selected in the program.

Databases are nowadays ubiquitous in software and many units of code interact intensively with a large, persistent and highly-structured relational database [18]. But barely few works (see [19], [20], [21]) have studied how to automate the generation of test inputs for such database programs, to test the correct *interaction* between the code and the database.

In this work, we propose to generalize the classical symbolic execution technique to the case of programs interacting with a relational database through SQL statements. We intend to generate a set of test cases for the program, including, for each test case, an input database content and values for the program inputs, to satisfy a given code coverage criterion. The path selection step of classical symbolic execution does not require any particular modification to handle database programs, as the presence of SQL statements in a program typically does not modify the way its control flow graph can be explored. The existing path selection algorithms can thus be used as well for database programs. However, the symbolic execution and constraints solving steps need to be adapted. First, because the generated constraints must now describe the content

of the relational tables of the database, subject to complex integrity constraints, as well as the content of the variables of the program. Secondly, because the symbolic execution process must be able to generate constraints modeling the complex behavior of the SQL statements used in the program, able to fail when violating the integrity constraints of the database.

The core strategy of our technique is, as we suggested in [21], to model every variable of the program and every table (which is, mathematically, a relation) in the database as a relational variable containing a mathematical relation over simple domains, like integers. Each statement in the program, including both imperative and SQL statements, can then be modeled as a simple relational operation over these relational variables. By applying the classical symbolic execution mechanism over this relational version of the program, we can derive a set of path constraints over the program input variables. The generated constraints are here relational constraints and the input variables refer both to the classical inputs of the program and to the content of each of the database tables at the program start. As some of the relational variables manipulated by the relational version of the program model tables in a database, they must obey the integrity constraints described by the relational schema of this database, such as, for example, the primary key or foreign key constraints. These schema constraints are modeled as relational constraints as well, and the path and schema constraints can then be combined into an unique input constraints system. Each solution to this relational constraints system represents a test input, including an initial state for each table in the database, with respect to which the program can be executed and is guaranteed to follow the execution path to be tested.

The main contribution of this work is a relational symbolic execution algorithm for database programs. This algorithm generates test data for a language composed of simple static Java methods, interacting with a relational database using SQL statements, through JDBC. Given the SQL DDL code describing the database schema, the Java code of the method and a finite path in the control flow graph of this method, the algorithm generates the corresponding relational input constraints in the Alloy language [22]. These constraints can then be solved using the Alloy analyzer [22]. The algorithm has been coded in Java and used to generate sets of test cases for a number of sample Java methods. The generated test sets satisfy classical code coverage criteria [13].

The remainder of this paper is organized as follows. Section II details the part of the Java/SQL syntax which is supported by our algorithm. In section III, we describe and illustrate our symbolic execution algorithm. We provide three examples of the whole test data generation process in section IV. Finally, some conclusions, related and future work are provided in section V.

II. Syntax of the tested Java/SQL programs

In this section, we define precisely which subset of the Java/SQL syntax our algorithm can execute symbolically. This subset has been chosen to offer a good compromise between simplicity and expressiveness. It includes common imperative statements, expressions and conditions, typical methods for List objects manipulation, SQL base primitives used in online transaction processing (OLTP) programs, base transaction management statements and typical database schema constraints constructs.

A sample database program in the language handled by our symbolic execution algorithm is provided in figure 1. This sample describes a database with two tables: one for shop departments and one for the articles stored in each of these departments. The total number of articles stored in a department is saved for each department. The sample also describes the method manipulating this database: it adds a set of new articles to the database and updates the departments' article counts. If the department of an added article was not in the database, it is added to the database as well. The articles are inserted one by one in isolated transactions. If a transaction was successful, the index of the added article is saved in a list.

In the next paragraphs, the chosen notation for the BNF grammar of the syntax includes some additional meta-symbols: {...} (grouping), $^?$ (zero or one times), * (zero or more times) and $^+$ (one or more times). When a single nonterminal appears several times in a single production, subscript notation allows to distinguish between the occurrences.

A. Database program

A database program is composed of the SQL DDL code of the database schema and of the code of the Java method under test.

⟨*database-program*⟩ ::= ⟨*sql-ddl*⟩ ⟨*java-method*⟩

B. Database schema

The relational database schema is a list of table definitions. This list can be empty. In such a case, the program works independently of any database. This allows our symbolic execution algorithm to be used for test input generation in the case of classical Java methods, with no interaction with a database. Each table is identified by its name, contains at least one attribute and endorses exactly one primary key. Foreign keys and additional check constraints can be declared for a table. A row in a table cannot be deleted or see its primary key value updated as long as there exists at least another row in the database that references it (ON DELETE/UPDATE NO ACTION). Semantics of all the schema creation primitives conforms to the classical SQL DDL specification provided by ISO.

⟨*sql-ddl*⟩ ::= ⟨*table*⟩*
⟨*table*⟩ ::= `CREATE TABLE`⟨*id*⟩`(`⟨*att*⟩$^+$⟨*p-key*⟩⟨*f-key*⟩*⟨*chk*⟩*`);`
⟨*att*⟩ ::= ⟨*id*⟩ `INTEGER NOT NULL ,`
⟨*p-key*⟩ ::= `CONSTRAINT` ⟨*id*⟩$_{cst}$ `PRIMARY KEY (` ⟨*id*⟩$_{att}$ `)`
⟨*f-key*⟩ ::= `,CONSTRAINT` ⟨*id*⟩$_{cst}$ `FOREIGN KEY (` ⟨*id*⟩$_{att}$ `)` `REFERENCES` ⟨*id*⟩$_{tab}$ `(` ⟨*id*⟩$_{refid}$ `)`
⟨*chk*⟩ ::= `,CHECK(`⟨*id*⟩ {`<` | `=` | `>`} ⟨*integer*⟩`)`
⟨*id*⟩ ::= {a |...| z | A |...| Z}{a |...| z | A |...| Z | 0 |...| 9}*
⟨*integer*⟩ ::= -$^?$ {1 | ... | 9}{0 | ... | 9}* | 0}

Fig. 1. SQL DDL and Java code of a database program inserting articles and updating departments' articles count in a shop database.

```
CREATE TABLE department (                    CREATE TABLE article (
 id INTEGER NOT NULL,                         barcode INTEGER NOT NULL,
 numberOfArticles INTEGER NOT NULL,           theDep INTEGER NOT NULL,
 CONSTRAINT dPK PRIMARY KEY (id),             CONSTRAINT aPK PRIMARY KEY (barcode),
 CHECK(numberOfArticles > 0))                 CONSTRAINT aFK FOREIGN KEY(theDep)REFERENCES department(id))
```

```
1  static void sample (Connection con,Scanner in,List<Integer> newArticles) throws SQLException {
2   int i = 0;
3   List<Integer> addedArticles = new ArrayList<Integer>();
4   while (!(newArticles==null) & i<newArticles.size()) {
5     int error = 0;
6     int departmentId = in.nextInt();
7     ResultSet departments = con.createStatement().executeQuery("SELECT id FROM department WHERE id="+departmentId);
8     if (!departments.next())
9      con.createStatement().execute("INSERT INTO department VALUES ("+departmentId+",1)");
10    else
11     con.createStatement().execute("UPDATE department SET numberOfArticles=numberOfArticles+1 WHERE id ="+departmentId);
12    try {
13     con.createStatement().execute("INSERT INTO article VALUES ("+newArticles.get(i)+","+departmentId+")");
14    } catch (SQLException e) {
15     error = 1;
16    };
17    i = i + 1;
18    if (error==0) {
19     con.commit();
20     addedArticles.add(i−1);
21    } else
22     con.rollback ();
23  }; }
```

C. Method parameters and body

We consider simple static Java methods manipulating only internal variables and parameters. Variables can only be typed as '`int`', '`java.util.List<java.lang.Integer>`' or '`java.sql.ResultSet`'. The method receives a connexion to the database (typed as '`java.sql.Connection`'), an input scanner (typed as '`java.util.Scanner`') and some lists of integers (typed as '`java.util.List<java.lang.Integer>`') as input parameters. The connection with the database is supposed to stay reliable and every SQL statement to be processed without any technical problem during the whole method execution. Semantics of all the Java constructs conforms to the classical Java specification and documentation. Semantics of all SQL statements conforms to the classical SQL specification provided by ISO.

⟨*java-method*⟩ ::= `static void` ⟨*id*⟩ `(`⟨*db-con*⟩`,`⟨*inp*⟩ ⟨*parameters*⟩`) throws SQLException {` ⟨*stmt*⟩* `}`
⟨*db-con*⟩ ::= `Connection con`
⟨*inp*⟩ ::= `Scanner in`
⟨*parameters*⟩ ::= `{, List<Integer>` ⟨*id*⟩ `}*`

1) Common statements and lists management: common condition, loop and assignment statements, as well as common integer expressions and boolean conditions can be used. Lists can be manipulated using the '`add(Integer)`', '`remove(int)`', '`get(int)`' and '`size(int)`' methods. The '`java.util.ArrayList<Integer>`' implementation of these methods is supposed to be used. A list variable can be '`null`'.

⟨*stmt*⟩ ::= `if (`⟨*cond*⟩`) {`⟨*stmt*⟩*$_{then}$`} {else {`⟨*stmt*⟩*$_{else}$`}}`?`;`
| `while (`⟨*cond*⟩`) {` ⟨*stmt*⟩* `};`
| `{int | List<Integer>}`? ⟨*id*⟩ `=` ⟨*expr*⟩`;`
| ⟨*id*⟩`.add(` ⟨*int-expr*⟩ `);`
| ⟨*id*⟩`.remove(` ⟨*int-expr*⟩ `);`
⟨*cond*⟩ ::= `true`
| `false`
| `(`⟨*cond*⟩$_1$ `{& | |}` ⟨*cond*⟩$_2$`)`
| `(!` ⟨*cond*⟩`)`
| `(`⟨*int-expr*⟩$_1$ `{< | == | >}` ⟨*int-expr*⟩$_2$`)`
| `(`⟨*id*⟩ `== null)`
⟨*expr*⟩ ::= ⟨*int-expr*⟩ | ⟨*list-expr*⟩
⟨*int-expr*⟩ ::= ⟨*id*⟩
| ⟨*integer*⟩
| `(`⟨*int-expr*⟩$_1$ `{+ | - | * | /}` ⟨*int-expr*⟩$_2$`)`
| `(-` ⟨*int-expr*⟩ `)`
| `(`⟨*id*⟩`.get(`⟨*int-expr*⟩`))`
| `(`⟨*id*⟩`.size())`
⟨*list-expr*⟩ ::= ⟨*id*⟩
| `null`
| `new ArrayList<Integer>()`

2) Interacting with the outside world: the input scanner parameter of the method can be used to get integer data from the "outside world" (user prompt, network access, reading from a file, etc.). This interaction is supposed to always succeed, without any technical problem.

⟨*stmt*⟩ ::= `{int}`? ⟨*id*⟩ = in.nextInt();

3) Reading data from the database: Data can be read from the database using simple SQL queries. The obtained ResultSet can be accessed using the '`next()`' and '`getInt(String)`' methods.

⟨*stmt*⟩ ::= ⟨*select*⟩
| ⟨*id*⟩`.next();`
⟨*select*⟩ ::= `{ResultSet}`?⟨*id*⟩`=`
`con.createStatement().executeQuery`
`("SELECT{`⟨*id*⟩$_i$`,}*`⟨*id*⟩$_n$`FROM`⟨*id*⟩$_{tab}$`{WHERE`⟨*db-cond*⟩`}`?`");`

⟨*db-cond*⟩ ::= (⟨*db-cond*⟩$_1$ {AND | OR} ⟨*db-cond*⟩$_2$)
| (NOT ⟨*db-cond*⟩)
| (⟨*id*⟩ {< | = | >} ⟨*db-int-expr*⟩)
⟨*db-int-expr*⟩ ::= ⟨*id*⟩
| ⟨*integer*⟩
| (⟨*db-int-expr*⟩$_1$ {+ | - | * | /} ⟨*db-int-expr*⟩$_2$)
| (- ⟨*db-int-expr*⟩)
| "+(⟨*int-expr*⟩)+"
⟨*int-expr*⟩ ::= ⟨*id*⟩$_{tab}$.getInt("⟨*id*⟩$_{att}$")
⟨*cond*⟩ ::= (⟨*id*⟩.next(⟨*int-expr*⟩))

4) Writing data into the database: data can be written into the database using simple SQL INSERT, UPDATE or DELETE statements. If the execution of a such a statement provokes a violation of one of the database schema integrity constraints, the database remains unmodified by the statement, an exception is thrown within the program and the method execution is stopped. Such exceptions can be caught using a try/catch structure.

⟨*stmt*⟩ ::= ⟨*db-write*⟩
| try { ⟨*db-write*⟩ } catch (SQLException e) { ⟨*stmt*⟩* };
⟨*db-write*⟩ ::=con.createStatement().execute("INSERT INTO ⟨*id*⟩ VALUES ({⟨*db-int-expr*⟩$_i$,}*⟨*db-int-expr*⟩$_n$)");
| con.createStatement().execute("UPDATE ⟨*id*⟩$_{tab}$ SET ⟨*id*⟩$_{att}$=⟨*db-int-expr*⟩ {WHERE ⟨*db-cond*⟩}$^?$");
| con.createStatement().execute("DELETE FROM ⟨*id*⟩ {WHERE ⟨*db-cond*⟩}$^?$");

5) Transactions management: SQL transactions are managed through the classical commit and rollback statements. A commit statement makes permanent all the changes made to the database by the program during the current transaction, closes this transaction and opens a new one. A rollback statement restores the database to its state at the start of the current transaction, closes this transaction and opens a new one. All pending transactions are supposed committed at the beginning of the tested method.

⟨*stmt*⟩ ::= con.commit() ; | con.rollback() ;

III. A relational symbolic execution algorithm for database programs

A. Inputs and outputs

The symbolic execution algorithm proposed here receives as inputs the SQL DLL code describing the schema of the tested database, the Java code of the tested method and an execution path in this method. It produces as output a relational constraints system, whose solutions are such that when the method is executed with respect to any of these solutions, its execution will follow the given path.

The execution path received as input by our algorithm is supposed to be a finite path in the method's control flow graph [13]. It defines which branch of each encountered If statement was taken, how many times the body of each encountered While loop was executed (this number must be finite), if the Catch clause of each of the encountered Try/Catch statements was executed, and if the method execution was brutally stopped by an uncaught exception thrown by a ⟨*db-write*⟩ statement violating database integrity.

The relational constraints generated as output by our algorithm are expressed using a widely used and well-documented language, offering good analysis tools, called Alloy [22]. Solving this constraints system allows to find values for each of the inputs of the analyzed Java method. These inputs include the content of each database table at method start, the content of every list received as a parameter by the method and the value returned by each '`in.nextInt()`' call made by the method. If the constraint system produced for a given path has no solution, this means that the path is infeasible.

B. Algorithm principle

The principle of the algorithm is to perform a relational symbolic execution of the program path received as input. Each of the different values taken by the method variables and by the database tables during the execution of the path is represented by a corresponding symbolic relational variable. First, symbolic execution generates constraints stating that the variables corresponding to the initial content of each table in the database conform to the database schema. Then, symbolic execution analyzes one by one the method statements executed by the path, in the order in which the path specifies that they are executed. Every time a statement sets or changes the value of a method variable or database table, symbolic execution generates a new constraint stating how the symbolic variable representing this new value can be computed from the other symbolic variables. Every time an If, While, Try/Catch or DB-Write statement is encountered, symbolic execution generates a constraint over the symbolic variables such that when the program is executed with respect to values satisfying this constraint, the execution is guaranteed to take the considered path.

C. Relational constraints generation rules

In this section, we illustrate the execution of the algorithm over the sample database program detailed in figure 1. We detail each step of the symbolic execution process over the path where the While loop is executed once, the Then branch of both the If statements are taken, the Catch clause of the Try/Catch is not executed and no uncaught exception is thrown (lines 1-9, 12-13, 17-20, 4 and 23). At each step, we present the rules used by our algorithm to generate Alloy symbolic variables and constraints. This step by step rules description process allows us to introduce the whole symbolic execution mechanism.

The algorithm always starts by generating symbolic variables and relational constraints for the database tables defined in the database schema. For each table, an Alloy type is first (1) defined so that every symbolic variable representing a set of rows of the table will be a subset of this type. Then, a symbolic variable is created to represent the initial content of the table (2). Finally, constraints are generated to enforce on this content the primary key (3) as well as all the foreign keys (4) and check constraints (5) defined in the table. For the database described in figure 1, the generated Alloy code is as follows:

```
module testCase // Name of the Alloy constraints model
//  Type for TABLE department
(1) sig department{id : Int,numberOfArticles : Int}
pred equaldepartment[a:department,b: department] {
a·id = b·id && a·numberOfArticles = b·numberOfArticles}
fact{all disj a, b: department | !equaldepartment[a,b]}
(2) sig departmentINPDB2 in department {}
(3) fact{all disj a, b: departmentINPDB2 | !((a·id = b·id))}
(5) fact{all a: department | a·numberOfArticles > 0}
// Type for TABLE article
(1) sig article {barcode : Int,theDep : Int}
pred equalarticle[ a: article , b: article ]
{a·barcode = b·barcode && a·theDep = b·theDep}
fact{all disj a, b: article | !equalarticle [a, b]}
(2) sig articleINPDB1 in article {}
(3) fact{all disj a, b: articleINPDB1 |
!(( a·barcode = b·barcode))}
(4) fact{all a: articleINPDB1 |
one b:departmentINPDB2 |a·theDep = b·id}
```

The second step executed by our algorithm is to generate a relational Alloy type for Java '`List<Integer>`' objects (1), and generic Alloy predicates and functions for their add/remove/get/size methods (2). Lists are modeled in Alloy as an interval of indexes from 0 to a natural number, where each index i is mapped to an integer value representing the i^{th} element of the list. The '`null`' value is represented by the Null singleton type (3).

```
(1) // List type definition
(3) one sig Null {}
sig List { index: set Int,elems: index →one Int }
fun head[l: set Int] : Int
{ { e:l | (all p: l | (!( e=p)) ⇔ (e< p)) } }
fun tail [l: set Int] : set Int
{ l − { e:l | (all p: l | (!( e=p)) ⇔ (e< p)) } }
pred isSuccessive[l: set Int]
{ all i:tail [l] | (one j:l | i = 1·add[j]) }
fact { all l:List |
(#l·index=0 | head[l·index]=0 && isSuccessive[l·index])}
pred isNull[l: List + Null] { l=Null }
pred isEmpty[l: List] { #l·index=0 }
(2) // Methods add − remove − get − size
pred add[l: List, e: Int,newList: List]
{ newList·index = l·index + #l·index &&
(all i:l·index | newList·elems[i] = l·elems[i])
&& newList·elems[#l·index] = e }
pred remove[l: List, e: Int,newList: List]
{ size [l] ≥ e &&
newList·index = l·index − (#l·index)·sub[1]
&& (all i:newList·index | (i< e ⇔
newList·elems[i] = l·elems[i])
&& (i≥ e ⇔ newList·elems[i] = l·elems[i · add[1]])) }
fun get[l: List, i: Int] : Int { l·elems[i] }
fun size[l: List] : Int { #l·index }
```

The third step executed by our algorithm is to define relational variables for the initial content of each list parameter of the method. For the method considered in this section, the following code is generated:

```
one sig newarticlesIN1 in List + Null {}
```

The algorithm can then proceed with symbolic execution of the tested method. It follows the path received as input and considers all statements one by one. In the case of the method and path considered in this section, the two first statements to be executed are Assignment statements. Symbolic execution for Assignment creates a new symbolic variable of the correct type to represent the new value of the assigned variable (1) and generates a constraint to specify that this new symbolic variable contains the value computed by evaluating the expression on the right of the '=' symbol (2).

```
(1) one sig iREL1 in Int {}
(2) fact { iREL1 = 0 }
```

```
(1) one sig addedarticlesREL2 in List {}
(2) fact { isEmpty[addedarticlesREL2] }
```

The second statement in the path is a While statement. As the path specifies that the loop body must be executed this time, a relational constraint is generated to specify that the loop condition should be true:

```
fact{(!isNull [newarticlesIN1]&&(iREL1< size[newarticlesIN1]))}
```

Then the algorithm proceeds with symbolic execution of the statements in the loop body, as specified within the input path. The first statement is an Assignment statement:

```
one sig errorREL2 in Int {}        fact { errorREL2 = 0 }
```

Symbolic execution for use of the input scanner simply creates a new symbolic variable to represent the scanned value:

```
one sig departmentidIN2 in Int {}
```

Symbolic execution for Select statements creates a new symbolic variable of the type of the table on which the Select query is executed (1), to represent the content of the ResultSet variable. A relational constraint is then generated (2) to specify that a row is part of the ResultSet if and only if it is part of the current content of the table on which the Select query is executed and that it enforces the WHERE condition of the Select query if it exists:

```
(1) sig departmentsREL3 in department {}
(2) fact { all e: department | (e in departmentINPDB2 &&
(e·id = departmentidIN2)) ⇔ e in departmentsREL3}
```

Symbolic execution for Select also creates a symbolic variable to represent the specific order in which the rows were returned by the Select query. This relational variable (1) represents an interval of indexes from 0 to the number of rows returned by the select statement (2), where each index is mapped to one of the returned rows (3):

```
(1) sig departmentsREL3indexes in Int {
mapdepartmentsREL3: one departmentsREL3 }
(2) fact { (#departmentsREL3indexes=0 ||
(head[departmentsREL3indexes] = 0 &&
isSuccessive [departmentsREL3indexes]))
 && #departmentsREL3indexes = #departmentsREL3 }
(3) fact { all disj a, b: departmentsREL3indexes |
!( a·mapdepartmentsREL3=b·mapdepartmentsREL3) }
```

As the path specifies that the Then branch of the If statement must be executed this time, a relational constraint is generated to specify that the If condition should be

true. For each ResultSet object, the algorithm records the number of times the '`next()`' method has been called on this object. This value represents the index of the row pointed by the cursor of the ResultSet at the current execution state of the path. When the boolean value returned by the '`next()`' method is used in an If or While condition, this value states if the number of rows in the ResultSet is greater or equal to the number of times the '`next()`' method has been called so far on this ResultSet:

fact{!(#departmentsINTERNALPROG3indexes≥ 1)}

Symbolic execution for Insert creates a new symbolic variable (1) for the new content of the table on which the Insert statement is executed. Then a relational constraint is generated stating that this new variable can be obtained by adding one row with the correct attribute values to the old one (2). Finally, relational constraints are added to specify that, in the considered path, no constraint was violated during insert. In this case, a relational constraint is added to state that there should not be any row in the previous content of the table whose primary key value is the same as in the row inserted by the statement (3):

(1) **sig** departmentRELDB1 **in** department {}
(2) **fact** { **one** e:department | departmentRELDB1 = departmentINPDB2 + e && e·numberOfArticles = 1 && e·id = departmentidIN2 }
(3) **fact** { **no** e:departmentINPDB2 | e·id=departmentidIN2 }

The same process is applied for the second insert statement. A relational constraint (1) is added to state that the inserted article should reference an existing row in the department table:

sig articleRELDB2 **in** article {}
fact { **one** e:article | articleRELDB2 = articleINPDB1 + e && e·theDep = departmentidIN2 && e· barcode = get[newarticlesIN1,iREL1] }
fact{**no** e:articleINPDB1|e·barcode=get[newarticlesIN1,iREL1]}
(1) **fact**{**one** e:departmentRELDB1|e·id=departmentidIN2}

The assignment statement is then symbolically executed:

one sig iREL4 **in Int** {}
fact { iREL4 = (iREL1)·add[1] }

As the path specifies that the Then branch of the If statement must be executed this time, a relational constraint is generated to specify that the If condition should be true:

fact { (errorREL2 = 0) }

Symbolic execution for Commit statements simply does nothing. Symbolic execution for Rollback statements tells the algorithm to use the symbolic variable representing the content of each database table before the last executed Commit statement (saved by the algorithm) to represent the current content of the table.

Symbolic execution for Add and Remove statements creates a new relational variable to represent the new content of each variable which is currently referencing the List object modified by the statement. A constraint is added to relate the old and new object rerferenced by these variables using the add/remove Alloy predicates defined earlier:

one sig addedarticlesREL6 **in** List {}
fact { add[addedarticlesREL2, (iREL4)·sub[1],addedarticlesREL6] }

As the path specifies that the loop body must not be executed any more, a relational constraint is generated to specify that the loop condition should be false:

fact{!((! isNull [newarticlesIN1]&&(iREL4< size[newarticlesIN1])))}

As all the statements in the path have been symbolically executed, the algorithm stops and returns the generated Alloy constraints model. The Alloy analyzer [22] can be asked to find a valuation for the defined relational variables which satisfies the constraints, using the commands detailed below. It should be observed that this valuation provides an assignment for the method inputs that will exercise the symbolically executed path, as well as the value of each method variable and database table at the end of the method execution over these inputs. This is particularly useful, as to produce unit tests, one needs both adequate inputs and the corresponding outputs produced by the method.

assert inputsExist{!(newarticlesIN1 **in** List + Null && departmentidIN2 **in Int** && articleINPDB1 **in** article && departmentINPDB2 **in** department) }
check inputsExist

Table I describes the full set of rules used by the algorithm to translate between Java/SQL and Alloy expressions and conditions. Table II details the relational constraints generated by the algorithm for every possible behavior of an Insert, Update or Delete statement. Table III explains the abbreviations used in tables I and II.

IV. Examples of test set generation

The algorithm proposed in this work has been coded in Java and exercised over a set of sample database programs. For each program, a set of execution paths have been selected to satisfy a classical coverage criterion [13]. Each path was symbolically executed and we asked the Alloy analyzer [22] to find solutions for the constraints generated by the algorithm. Each of these computed solutions was checked to be actually input data with respect to which the experimented program was guaranteed to follow the selected path.

The Alloy analyzer is a program which allows to solve Alloy constraints in order to find structures that satisfy them. Basically, it transforms the set of relational constraints into a set of boolean constraints, and solves them using a SAT solver. The process requires to define the maximal scope of the structures the Alloy analyzer should search in [22]. The set of possible structures within this scope is then explored, and the found solutions are returned one by one on demand. In case no solutions to the constraints can be found within a given maximal scope, the solving process is repeated with an increased value for the scope, until it eventually reaches a threshold value.

Three sample database programs were selected to offer a clear illustration of how our algorithm can symbolically

TABLE I. TRANSLATION OF JAVA/SQL EXPRESSIONS AND CONDITIONS INTO ALLOY

Parameters	**alloyOf(Paramaters)**
$\langle id \rangle$	alloyName
$z \in \mathbb{Z}$	z
($\langle int\text{-}expr \rangle_1$ {+|-|*|/} $\langle int\text{-}expr \rangle_2$)	(alloyOf($\langle int\text{-}expr \rangle_1$)).{add | sub | mul | div}[alloyOf($\langle int\text{-}expr \rangle_2$)]
(- $\langle int\text{-}expr \rangle$)	(- (alloyOf($\langle int\text{-}expr \rangle$)))
$\langle id \rangle$.get($\langle int\text{-}expr \rangle$)	get[alloyName,alloyOf($\langle int\text{-}expr \rangle$)]
$\langle id \rangle$.size()	size[alloyName]
$\langle id \rangle_{tab}$.getInt("$\langle id \rangle_{att}$")	$\mathsf{numberOfCallsToNext}_{tab}$.map$\mathsf{alloyName}_{tab}[\langle id \rangle_{att}]$ fact{#$\mathsf{alloyName}_{tab}$indexes >= $\mathsf{numberOfCallsToNext}_{tab}$}
$\langle id \rangle$=null;	fact{ isNull[alloyName] }
$\langle id \rangle$=new ArrayList<Integer>();	fact{ isEmpty[alloyName] }
true false	(0=0) (0=1)
($\langle cond \rangle_1$ {&||} $\langle cond \rangle_2$)	(alloyOf($\langle cond \rangle_1$) {&&|||} alloyOf($\langle cond \rangle_2$))
(! $\langle cond \rangle$)	(!(alloyOf($\langle cond \rangle$)))
($\langle int\text{-}expr \rangle_1$ {<|==|>} $\langle int\text{-}expr \rangle_2$)	((alloyOf($\langle int\text{-}expr \rangle_1$)) {<|=|>} (alloyOf($\langle int\text{-}expr \rangle_2$)))
($\langle id \rangle$==null)	isNull[alloyName]
$\langle id \rangle$.next()	(#alloyNameindexes >= numberOfCallsToNext)
($\langle db\text{-}cond \rangle_1$ {AND|OR} $\langle db\text{-}cond \rangle_2$), row	(alloyOf($\langle db\text{-}cond \rangle_1$,row) {&&|||} alloyOf($\langle db\text{-}cond \rangle_2$,row))
(NOT $\langle db\text{-}cond \rangle$), row	(!(alloyOf($\langle db\text{-}cond \rangle$,row)))
($\langle id \rangle${<|=|>} $\langle db\text{-}int\text{-}expr \rangle$), row	(row.$\langle id \rangle$ {<|=|>} (alloyOf($\langle db\text{-}int\text{-}expr \rangle$,row)))
$z \in \mathbb{Z}$, row	z
$\langle id \rangle$, row	row.$\langle id \rangle$
($\langle db\text{-}int\text{-}expr \rangle_1$ {+|-|*|/} $\langle db\text{-}int\text{-}expr \rangle_2$), row	(alloyOf($\langle db\text{-}int\text{-}expr \rangle_1$,row)).{add | sub | mul | div}[alloyOf($\langle db\text{-}int\text{-}expr \rangle_2$,row)]
(- $\langle db\text{-}int\text{-}expr \rangle$), row	(- (alloyOf($\langle db\text{-}int\text{-}expr \rangle$,row)))
"+$\langle int\text{-}expr \rangle$+", row	alloyOf($\langle int\text{-}expr \rangle$)

TABLE II. RELATIONAL CONSTRAINTS GENERATION FOR $\langle db\text{-}write \rangle$ STATEMENTS

INSERT INTO $\langle id \rangle$ VALUES ($\langle db\text{-}int\text{-}expr \rangle_1$,..., $\langle db\text{-}int\text{-}expr \rangle_i$,..., $\langle db\text{-}int\text{-}expr \rangle_n$)	**if** (no exception thrown in path by this INSERT) { **sig** freshAlloyVar **in** $\langle id \rangle$ {} **fact** { **one** e:$\langle id \rangle$ | freshAlloyVar=alloyName+e && e·att$_*$ = alloyOf($\langle db\text{-}int\text{-}expr \rangle_*$) } **fact** { **no** e:alloyName | e·pk =alloyOf($\langle db\text{-}int\text{-}expr \rangle_{\mathsf{pk}^{pos}}$) } Primary key is **not** violated **fact** { **one** e:fk_*^{tab} | e·fk_*^{pk} = alloyOf($\langle db\text{-}int\text{-}expr \rangle_{\mathsf{fk}_*^{pos}}$) } Every foreign key is **not** violated }**else**{Logical disjunction between every possible constraint violation given the database schema and this insert: The inserted primary key value already exists in the table: **fact** { **one** e:alloyName | e·pk =alloyOf($\langle db\text{-}int\text{-}expr \rangle_{\mathsf{pk}^{pos}}$) } An inserted foreign key value references no row: **fact** { **no** e:fk_i^{tab} | e·fk_i^{pk} = alloyOf($\langle db\text{-}int\text{-}expr \rangle_{\mathsf{fk}_i^{pos}}$) } An inserted attribute can violate a check constraint: **fact** {!($\langle db\text{-}int\text{-}expr \rangle_{\mathsf{co}^{pos}}$ co^{right})} }
UPDATE $\langle id \rangle$ SET $\langle id \rangle_{att}$=$\langle db\text{-}int\text{-}expr \rangle$ WHERE $\langle db\text{-}cond \rangle$	**if** (no exception thrown in path for this UPDATE) { **sig** freshAlloyVar **in** $\langle id \rangle$ {} **fact** { **all** e:alloyName | (alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle$, e) && (**one** y:freshAlloyVar | y·$\mathsf{att}_{*-\{\langle id \rangle_{att}\}}$= e·$\mathsf{att}_{*-\{\langle id \rangle_{att}\}}$ && y·$\langle id \rangle_{att}$ = alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$,e))) || (!(alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle$, e)) && (**one** y:freshAlloyVar | equal$\langle id \rangle$[y,e])) } **fact** { **all** y:freshAlloyVar | **one** e:alloyName | (alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle$, e) && y·$\mathsf{att}_{*-\{\langle id \rangle_{att}\}}$= e·$\mathsf{att}_{*-\{\langle id \rangle_{att}\}}$ && y·$\langle id \rangle_{att}$ = alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$,e))) || (!(alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle$, e)) && (**one** y:freshAlloyVar | equal$\langle id \rangle$[y,e])) } **fact** { **all disj** a, b:freshAlloyVar | !(a·pk = b·pk) } Primary key is **not** violated If $\langle id \rangle_{att}$ = fk_i, updated rows should still reference a row **fact** { **all** a:freshAlloyVar | **one** b:fk_i^{tab} | a·$\langle id \rangle_{att}$ = b·fk_i^{pk} } If $\langle id \rangle_{att}$ = pk, **none** of the updated rows should have been referenced by another row **fact** { **all** e:alloyName | **no** f:ifk_*^{tab} | (alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle$, e) && e·$\langle id \rangle_{att}$ = f·ifk_*^{att}) } } **else** { Logical disjunction between every possible constraint violation given the database schema and this update: Update on primary key leads to duplicate primary keys: **fact** { **some disj** a,b:alloyName | ((alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, a) && alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$,b) && (alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$,a) = alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$, b))) || (!(alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, a)) && alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$,b) && (a·$\langle id \rangle_{att}$ = alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$,b)))) } Trying to update the primary key of a referenced row: **fact** { **some** a:alloyName | alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, a) && (**some** ifk_j^{tab} | (b·ifk_j^{att} =a·$\langle id \rangle_{att}$)) } Update on foreign key let row without row to reference: **fact** { **some** a:alloyName | alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, a) && (**no** b:fk_i^{tab} | b·fk_i^{pk}=alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$,a)) } An inserted attribute violates an arithmetic constraint: **fact** { **some** a:alloyName | alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, a) && !(alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$,a))$\mathsf{co}_i^{right}$)}} }
DELETE FROM $\langle id \rangle$ WHERE $\langle db\text{-}cond \rangle$	**if** (no exception thrown in path for this DELETE) { **sig** freshAlloyVar **in** $\langle id \rangle$ {} **fact** {freshAlloyVar = alloyName − {e:alloyName | alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, e)}} Not trying to delete a referenced row **fact** {**all** e:alloyName | **no** f:ifk_j^{tab} | (alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, e) && e·pk = f·ifk_j^{att})} } **else** { Logical disjunction between every possible constraint violation given the database schema and this update: Trying to delete a referenced row: **fact** {**some** e:alloyName | alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, e) && (**some** f:ifk_j^{tab} | e·pk = f·ifk_j^{att}) } } }

TABLE III. Abbreviations list

Abbreviation	Meaning
freshAlloyVar	A new Alloy variable name that has still not been used in the Alloy code generated so far.
$\mathsf{alloyName}^{superscript}_{subscript}$	**if** ($\langle id\rangle^{superscript}_{subscript}$ refers to a database table name) **then** The symbolic variable that represents the current content of table $\langle id\rangle^{superscript}_{subscript}$) **else** The symbolic variable that represents the current content of the Java variable $\langle id\rangle^{superscript}_{subscript}$
$\mathsf{numberOfCallsToNext}_{sub}$	Number of call made to the next() method of the ResultSet object in variable $\langle id\rangle_{sub}$
att_i	Name of the i_{th} attribute in the list of attributes of table $\langle id\rangle$
$\mathsf{pk}^{superscript}_{subscript}$	Name of the primary key attribute of table $\langle id\rangle^{superscript}_{subscript}$.
pk^{pos}	Position of primary key in the list of attributes of table $\langle id\rangle$
fk^{tab}_i	Name of the table referenced by the i_{th} foreign key in the list of foreign keys of table $\langle id\rangle$
fk^{pk}_i	Name of the primary key attribute of the table referenced by the i_{th} foreign key in the list of foreign keys of table $\langle id\rangle$
fk^{pos}_i	Position of the foreign key attribute, declared by the i_{th} foreign key in the list of table $\langle id\rangle$, in the list of attributes of table $\langle id\rangle$
fk_i	Name of the foreign key attribute declared by the i_{th} foreign key in the list of table $\langle id\rangle$
ifk^{tab}_i	Name of the table where is declared the i_{th} foreign key referencing table $\langle id\rangle$ in the whole schema
ifk^{att}_i	Name of the foreign key attribute declared by the i_{th} foreign key referencing table $\langle id\rangle$ in the schema
co^{pos}_i	Position of the attribute constrained by the i_{th} check constraint declared in table $\langle id\rangle$
co^{right}_i	Right part of the i_{th} check constraint declared in table $\langle id\rangle$ (i.e. right part of "a>0" is ">0")
xx_* means "for each xx_i" and $\mathsf{xx}_{*-\{y\}}$ means "for each xx_i except from y"	

TABLE IV. Statistics for the selected samples

#	SLOC	SQL statements	Criterion	Maximal scope value	Variables	Constraints	Solving time
1	45	1	Branch coverage	3	67	113	9245 ms
2	56	18	Statement coverage	20	47	120	499685 ms
3	72	4	Branch coverage	3	120	180	16437 ms

execute list management primitives, SQL statements and transactions.

The first sample is a Java method which performs repeated manipulations of integers and lists using assignment, If and While statements. First, two lists are received as parameters. If they are both not null and have the same size, the method reads as many integers from the outside world as the number of elements in the lists. A third list is created using these integers in the order in which they were read. The elements of the three lists are then compared. If the second list is an inverted version of the first one and if the third list is a copy of the first list where the value of each element was doubled, then the three lists are inserted into a database table. The database schema and the way the lists are inserted in this database constrain the elements in the first list to be different from each other and their value to range between one and five.

The second sample is a Java method which performs repeated reads and writes in a database containing four tables that represent customers making purchases of products. Customers with few purchases are prospect customers. First, the program inserts a new customer and new purchases into the database. Then it computes the total number and cost of the purchases made by each customer, as well as the total number of purchases for each product, and then updates the corresponding customer and product attributes. All unpurchased products are deleted, and the name of the customers having made no purchase is changed. Customers whose total count and cost of purchase is lower than two are registered as prospect customers. Finally, a product is replaced by another one in every purchase details, and this product is deleted.

The third sample is a Java method which mixes SQL statements with traditional Java code and uses SQL transactions. The database contains two tables that represent authors writing theater plays. The code contains two transactions. During a first transaction, some authors are added and some removed from the database. During a second transaction, plays are added for the previously added authors, and some statistics are computed for each author. If a database schema constraint is violated by a SQL statement in one of the two transactions, this whole transaction is cancelled and the database is rolled back to the state it was when the transaction was launched.

The source code of these three samples, the Alloy constraints generated by our algorithm and the resulting generated test cases, as well as all the detailed performance-related information can be found on the web[1]. The main information are synthesized in table IV, including the SLOC and the number of SQL statements in the sample, the coverage criterion satisfied by the generated test set, the maximal scope value used in the Alloy analyzer to solve the constraints produced for the sample, the total number of relational variables and constraints solved to find a test set for the sample, and the time for solving them. These measures were realized on a dual core Intel Core i5 processor at 1.8GHz (256 KB L2 cache per core and 3 MB L3 cache) with 8GB of dual channel DDR3 memory at 1600 MHz using the MiniSat solver. The oder-of-magnitude difference in constraints solving time for the second sample is due to the use of a larger maximal scope value in the Alloy analyzer.

The subset of Java/SQL supported by our algorithm allows integers as only primary type in Java code and database tables. This choice was adopted to make the modeling and use of the constraint generation rules conceptually simpler. This does not limit the power of the proposed technique since all other usual primitive types such as booleans, strings, and floating point numbers, but also data structures such as sets, arrays and matrices, and Java objects can be mapped to integers, simulated using lists of integers, or directly modeled into Alloy (see e.g. [23]). These reasons explain why the previously described

[1]See http://info.fundp.ac.be/ mmr/scam13

samples manipulate integer values, like author or play names, which are not usually typed as integers.

V. Conclusion and related work

In this work, we have proposed and demonstrated a complete algorithm to execute symbolically simple static Java methods using SQL CRUD statements and transactions to interact with a relational database, subject to integrity constraints. Given the schema of the database, the code of the method and an execution path in this method, the algorithm generates a relational symbolic variable for each potential value taken by a method variable or database table before and during the path execution. It generates as well an Alloy relational constraints model constraining these relational symbolic variables to guarantee the execution of the considered path. Any solution to the produced relational constraints describes input data for the method (as well as the corresponding output), with respect to which the program can be executed and is guaranteed to follow the considered execution path. Given a set of execution paths to test in the database program chosen using an existing algorithm [12] to satisfy a given code coverage criterion [13], the proposed algorithm can be used to generate test data, including database initial and final states, for each path in the set.

An early approach that has considered test data generation for imperative programs interacting with a relational SQL database is [19]. The paper proposes to transform the program, thereby inserting new variables representing the database structure, and translating all SQL statements (and thus integrity checks) into imperative program code. Classical white-box testing approaches can then be applied to the modified program. In [20], the authors propose an algorithm for testing an imperative program performing SELECT queries on a relational SQL database, based on concolic execution [11]. Concolic execution is an extension [12] of the classical symbolic execution used in this work. It mixes symbolic execution with a simultaneous concrete dynamic execution of the considered execution paths. In [20], the program is first run on random input data and on a randomly populated input database. Given such a dynamic exploration of an execution path of the program, the authors model and solve the problem of finding other inputs, allowing to explore dynamically another execution path, as a set of integer and string constraints over the quantity and field contents of the records in the database and over the input variables of the tested program. In [24], authors adopt a similar concolic approach where the program is executed on a parameterized mock database. In [25] and [26], authors adapt this approach to testing of programs running on an existing database, so that input test data can be selected in this database instead of being generated from scratch. In [27], the same concolic approach is applied considering advanced code coverage criteria. Finally, translation between database schemas/programs/queries and Alloy has been considered in other contexts [28], [29], [30].

Compared to [20], our approach does not need to transform the original program by adding potentially complex pieces of imperative code, to simulate the execution of SQL statements by a DBMS. Compared to [20], [24], [25], [26], [27], our approach handles Insert, Update and Delete statements, as well as transactions management primitives (Commit and Rollback) and database integrity constraints (like primary keys or foreign keys) which are crucial components of database applications. On the other hand, our approach only considers SQL statements whose structure is completely defined statically, where the concolic process used in [20], [24], [25], [26], [27] can allow to account, at least at some extent, for dynamically crafted SQL statements. This point should be tempered by the fact that our approach evaluates a particular path in the program at a time, which makes easier to reassemble statically the SQL statements supposed to be dynamically assembled during the execution of the path.

In future work, it would be relevant to evaluate how and up to which extent the symbolic execution mechanism proposed can be practically generalized to a larger part of the Java and SQL languages. Similarly, it should be investigated how and up to which extent fully-dynamic SQL can be integrated with our approach, possibly relying on static analysis [31], [32], [33] or on concolic execution. These tasks notably pave the way towards an extensive evaluation of the method on large scale industrial systems. Secondly, it happens frequently that SQL statements have a non-deterministic behavior, either because the statement has a non-deterministic semantics, or because the database is modified concurrently by several programs. How the approach proposed here can handle at best such non-deterministic behaviors remains a topic for further research. Thirdly, our approach allows to be used with respect to any classical code coverage criterion based on the notion of an execution path. Nevertheless, several works [34], [35], [36], [37], [38] propose test adequacy criteria particularly adapted to the testing of database-driven programs. Integrating such particular coverage criteria into our constraint-based approach is a topic of ongoing research. Finally, [39] and [40] propose new approaches to solve and evaluate the satisfiability of sets of relational constraints. It would be relevant to study how the technique proposed here could benefit from these approaches, notably to detect unfeasible paths more efficiently or to offer improved constraints solving time performance.

Acknowledgment

This work has been funded by the Belgian Fund for Scientific Research (F.R.S.-FNRS). The authors would like to thank Vincent Englebert for useful discussions and the anonymous reviewers for their valuable comments.

References

[1] P. C. Jorgensen, *Software Testing: A Craftsman's Approach, Third Edition*, 3rd ed. AUERBACH, 2008.

[2] C. Kaner, H. Q. Nguyen, and J. L. Falk, *Testing Computer Software*, 2nd ed. New York, NY, USA: Wiley & Sons, 1993.

[3] R. Ramler and K. Wolfmaier, "Economic perspectives in test automation: balancing automated and manual testing with opportunity cost," in *Proceedings of the 2006 international workshop on Automation of software test*, ser. AST '06. New York, NY, USA: ACM, 2006, pp. 85–91.

[4] J. Malburg and G. Fraser, "Combining search-based and constraint-based testing," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 436–439.

[5] P. McMinn, "Search-based software test data generation: a survey: Research articles," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS*, 2006.

[7] F. Degrave, T. Schrijvers, and W. Vanhoof, "Towards a framework for constraint-based test case generation," in *Proceedings of the 19th international conference on Logic-Based Program Synthesis and Transformation*, ser. LOPSTR'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 128–142.

[8] A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques," *SIGSOFT Softw. Eng. Notes*, vol. 23, no. 2, pp. 53–62, Mar. 1998.

[9] C. Meudec, "ATGen: automatic test data generation using constraint logic programming and symbolic execution," *Software Testing Verification and Reliability*, vol. 11, no. 2, 2001.

[10] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation," *Softw. Pract. Exper.*, vol. 29, no. 2, pp. 167–193, Feb. 1999.

[11] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 263–272, Sep. 2005.

[12] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communication of the ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[13] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997.

[14] S. Bardin and P. Herrmann, "Pruning the search space in path-based test generation," in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ser. ICST '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 240–249.

[15] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[16] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Softw. Eng.*, vol. 2, no. 3, pp. 215–222, May 1976.

[17] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991.

[18] C. Date, *An Introduction to Database Systems*, 8th ed. Boston, MA, USA: Addison-Wesley Longman Pub. Co., Inc., 2003.

[19] M. Y. Chan and S. C. Cheung, "Testing database applications with sql semantics," in *In Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*. Springer, 1999, pp. 363–374.

[20] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *Proceedings of the 2007 international symposium on Software testing and analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 151–162.

[21] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut, "Test input generation for database programs using relational constraints," in *Proceedings of the Fifth International Workshop on Testing Database Systems*, ser. DBTest '12. New York, NY, USA: ACM, 2012, pp. 6:1–6:6.

[22] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[23] M. L. Crane and J. Dingel, "Runtime conformance checking of objects using alloy," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 2 – 21, 2003.

[24] K. Taneja, Y. Zhang, and T. Xie, "Moda: Automated test generation for database applications via mock objects," in *In Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE 2010), short paper*, 2010.

[25] C. Li and C. Csallner, "Dynamic symbolic database application testing," in *Proceedings of the Third International Workshop on Testing Database Systems*, ser. DBTest '10. New York, NY, USA: ACM, 2010, pp. 7:1–7:6.

[26] K. Pan, X. Wu, and T. Xie, "Generating program inputs for database application testing," in *Proc. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, November 2011.

[27] ——, "Database state generation via dynamic symbolic execution for coverage criteria," in *Proceedings of the Fourth International Workshop on Testing Database Systems*. New York, NY, USA: ACM, 2011.

[28] A. Cunha and H. Pacheco, "Mapping between alloy specifications and database implementations," in *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, ser. SEFM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 285–294.

[29] D. Jackson and M. Vaziri, "Finding bugs with a constraint solver," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 5, pp. 14–25, Aug. 2000.

[30] S. A. Khalek and S. Khurshid, "Systematic testing of database engines using a relational constraint solver," in *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 50–59.

[31] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, Sep. 2007.

[32] S. Thomas, L. Williams, and T. Xie, "On automated prepared statement generation to remove sql injection vulnerabilities," *Inf. Softw. Technol.*, vol. 51, no. 3, pp. 589–598, Mar. 2009.

[33] A. Christensen, A. M,àö,àèller, and M. Schwartzbach, "Precise analysis of string expressions," in *Static Analysis*, ser. Lecture Notes in Computer Science, R. Cousot, Ed. Springer Berlin Heidelberg, 2003, vol. 2694, pp. 1–18.

[34] W. G. J. Halfond and A. Orso, "Command-form coverage for testing database applications," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 69–80.

[35] G. M. Kapfhammer and M. L. Soffa, "A family of test adequacy criteria for database-driven applications," in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-11. New York, NY, USA: ACM, 2003, pp. 98–107.

[36] M. J. Suárez-Cabal and J. Tuya, "Using an sql coverage measurement for testing database applications," in *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, ser. SIGSOFT '04/FSE-12. New York, NY, USA: ACM, 2004, pp. 253–262.

[37] M. J. Suarez-Cabal and J. Tuya, "Structural coverage criteria for testing SQL queries," *Journal of Universal Computer Science*, vol. 15, no. 3, pp. 584–619, 2009.

[38] C. Zhou and P. Frankl, "Mutation testing for java database applications," in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ser. ICST '09. Washington, DC, USA: IEEE Comp. Soc., 2009, pp. 396–405.

[39] A. A. El Ghazi and M. Taghdiri, "Relational reasoning via smt solving," in *Proceedings of the 17th international conference on Formal methods*, ser. FM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 133–148.

[40] E. Torlak, F. S.-H. Chang, and D. Jackson, "Finding minimal unsatisfiable cores of declarative specifications," in *Proc. of the 15th international symposium on Formal Methods*, ser. FM '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 326–341.

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

Ontological Interpretation of Object-Oriented Programming Abstractions

Arvind W Kiwelekar*, Rushikesh K Joshi
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai-400076, India
Email: {awk,rkj}@cse.iitb.ac.in

***Abstract*—Assigning ontological categories to OO programming abstractions is a task of ontological interpretation. In this paper, we describe an approach for ontological interpretation in which the task of interpretation is seen as a *classification problem*. The task of interpretation is performed in two stages, namely syntactic feature identification and ontological categorization. An analysis technique called Ontological Cards (Onto-card) is developed to identify ontologically relevant syntactic features from programs. The ontological card includes a set of syntactic and relational features describing the nature of program elements. A rule-based classifier system is implemented to classify programming abstractions into ontological categories. The approach is assessed with the help of two different Java-based applications. The descriptions make use of examples outlining issues and challenges involved in the approach.**

I. Introduction

The motivation behind the approach for ontological interpretation of programming abstractions discussed in this paper is drawn from the ways employed to interpret symbols in natural language text, and by observing how the interpreted text is used to support natural language processing. In the context of natural languages, the process of assigning meanings to language symbols is referred to as *interpretation*. The task of interpretation is primarily carried out for meaning extraction and also to achieve effective communication of ideas especially when participants use two different languages for exchanging information.

In the context of software engineering activities, various stakeholders such as domain analysts, system architects and system programmers use different modeling languages and notations. Furthermore, they need to relate program structures and modeling abstractions based on their intended meanings to extract high-level design information from system implementations. A commonality between the two tasks of natural language translation and the software engineering interpretation activity is that they need to bridge the semantic gap caused by the use of different languages.

Drawing on these similarities between natural language translation and software engineering interpretation activities such as architecture recovery and integration of multiple levels of abstractions, the paper presents a technique of *ontological interpretation* to assign meanings to programming abstractions. The proposed technique is a light-weight adaptation of the approach of *ontological semantics* in the context of natural language text [1]. In their approach, Nirenburg et al. use models of reality i.e. ontology, to extract and represent meanings of natural language symbols. The ontological approach is adopted by them with two intentions- firstly to understand natural language text, and secondly to develop techniques for automated language translation and text summarization.

Similarly, in our case, gaining an understanding of program organization and deriving high-level design information from system implementations are the prime motives for adopting an ontological approach to interpret programming abstractions. Some of the potential applications of the proposed technique include architecture knowledge discovery and achieving *semantic interoperability* during model translation activity. The approach presented in this paper adopts the ontology of Bunge-Wand-Weber (BWW) [3], [4] to interpret OO programming abstractions. A set of classification rules are formulated to assign an appropriate ontological category to a given programming abstraction.

The rest of the paper is organized as follows. Conventional ways of assigning meanings to language symbols are reviewed in Section 2, which is followed by a brief outline of our approach in Section 3. Sections 4 and 5 present the design of the ontological approach and the issues involved in it. Section 6 develops the approach of onto-cards. Sections 7 and 8 describe the implementation and evaluation of the rule-based approach built upon onto-cards. A comparison of the approach presented in this paper with existing approaches for classifying program elements is presented in Section 9. Finally, the paper concludes with a summarizing discussion on the approach pointing to current limitations and the scope for future work.

II. Background

Ogden and Richards formulate the relationship between natural language symbols and real world phenomena through two postulates in [5]: Firstly, a symbol represents a thought of reference, and secondly, the thought of reference refers to a referent, which is an object from the reality. They capture these facts through the *meaning triangle* shown in Figure 1. The meaning triangle relates referent objects, mental images of the objects and symbols used to describe them.

* Arvind W Kiwelekar is presently working with Dr. Babasaheb Ambedkar Technological University, Lonere-402103, Raigad (MS), India.

978-1-4673-5739-5/13/$31.00

	Greenberg [1]	Lewis [2]	Our Adaptation
Symbol	Sentence	Proposition	Ontological Category
Thought	Sentential Meaning	Intension	Implementation Conceptualization
Referent	Fact, Situation, State of Affairs	Extension	Programming abstractions
Symbolizes	Symbolizes	Symbolizes	Symbolizes
Refers to	Refers to	Refers to	Refers to
Stands for	Stands for	Stands for	Ontological Interpretation

TABLE I. COMPARISON OF EARLIER AND OUR ADAPTATIONS OF OGDEN AND RICHARDS' TRIANGLE

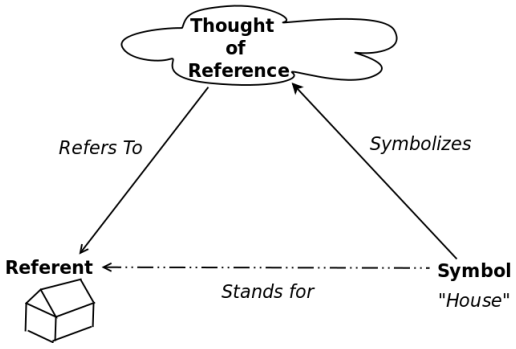


Fig. 1. Ogden and Richards' Meaning Triangle

Later on, the meaning triangle has been adapted by linguists to assign meanings to symbols from natural languages [2] and formal languages such as logical expressions [1]. The adaptations of the basic meaning triangle in natural language processing and logical expressions are summarized in Table I. The table shows the approach specific adaptations of entities and relations in the meaning triangle.

The base theme behind the meaning-triangle based approaches is the association of an object from the reality to a language symbol. Various kinds of referent objects and phenomena found in the reality are generally referred to as *ontological categories*. As it has also been observed in [1], one can not directly interpret the symbols expressed in natural or artificial languages in terms of referent objects in the reality without the use of a meta-level understanding which forms the basis of this interpretation. In our approach, ontological categories are chosen as the base meta-level understanding for interpretation of symbols from language expressions. *Thing*, *Property* and *Event* are the three primary ontological categories from Bunge's Ontology [3] that have been considered as the basis of the study reported in this work.

The category *Thing* conceptualizes various kinds of concrete objects. Examples of this category are $Person$, $Book$ and $Bank\ Account$. Properties are used to characterize things. For example, {*date of birth X name*}, {*price X title* } and {*Account number X balance*} can be specified as the corresponding property sets characterizing the above examples of things.

Properties may be unchanging rigid properties, or they may also represent changing states of things. In real life, properties have no distinct existence as individual objects. However, in a program, while a class may represent a thing having dynamic behavior, another may represent a set of properties alone.

An *event* is said to have occurred when a state change takes place as a thing undergoes changes in the values of its properties. For example, *borrowing of a library book* is an event capturing a value change for the property $status$ of a book, a thing, from value $onTheRack$ to value $issued$.

III. OUR APPROACH

The meaning triangle in Figure 1 provides the theoretical basis to interpret programming abstractions. In our approach, programming abstractions from a given software system are the referents, and ontological categories are the symbols interpreting the meanings of programming abstractions. Ontological categories are selected for interpretation as they capture different types of phenomena observed in the reality.

In this paper, we further develop a *rule-based* technique to automate the process of ontological interpretation. The decision rules to assign ontological categories to programming abstractions are learnt from a manual analysis of around two hundred classes from an open source software called $DSpace$ [6]. The rules thus formulated form the basis of an automated classifier system. Then the effectiveness of the classifier system is evaluated through performance metrics such as recall, precision and accuracy, which are typically used to evaluate the performance of techniques of information retrieval from databases.

IV. ISSUES IN ONTOLOGICAL INTERPRETATION

Figure 2 shows the partial implementations of three programming abstractions from the application $HealthWatcher$ [7], which is an open source application software from the domain of health informatics. These abstractions are selected to illustrate the fact that programming abstractions represent various kinds of phenomena observed in the reality.

The kind of phenomenon from the reality a given programming abstraction represents is called its *ontological kind*. The process of identifying the ontological kinds for programming abstractions is the process of ontological interpretation. In Figure 2, the abstractions $Employee$, $Address$, and $InsertEmployee$ represent ontological kinds $Thing$, $Property$ (precisely speaking, *Attribute* in the BWW ontology) and $Event$ respectively.

Listing 1. Employee.java
```
package healthwatcher.model.employee;
public class Employee implements Serializable,
                                  Subject {
    private String name;
    private String login;
    public Employee(String login, String password,
                    String name) {..}
    public String getName() {..}
    public void setName(String name) {..}
}
```

Listing 2. Address.java
```
package healthwatcher.model.address;
public class Address implements java.io.Serializable {
     private int code;
     private String street;
     private String complement;
     public Address() {}
     public String getCity() {..}
     public void setCity(String city) {..}
}
```

Listing 3. InsertEmployee.java
```
package healthwatcher.view.command;
public class InsertEmployee extends Command {
     public InsertEmployee(IFacade f) {..}
     public void execute() throws Exception {..}
}
```

Fig. 2. Examples of Programming Abstractions from $HealthWatcher$

The reason for this interpretation is that the first programming abstraction $Employee$ refers to substantial objects having spatio-temporal existence in the reality. The second abstraction $Address$ refers to descriptive objects having no separate existence from the things to which they associate. The third abstraction $InsertEmployee$ refers to ontological objects that have only temporal existence.

Figure 2 shows only a subclass of programming abstractions representing application domain concepts such as *employee*, *address of an employee* and operation *inserting employee record*. This class of programming abstractions realize what are typically called as *functional requirements* of an application. However, our observation is that the majority programming abstractions from an OO application are implemented to realize non-functional requirements such as supporting serialization tasks, providing distribution of objects and adapting functionality of an interface to existing programming abstractions. The approach presented in this paper attempts to interpret both types of programming abstractions i.e. abstractions representing application domain core concepts (classes realizing functional requirements) as well as those representing non-functional features.

A. Reference Ontological Kinds

Before proceeding with the task of interpretation, a finite set of ontological kinds against which the programming abstractions are interpreted needs to be formulated. For this purpose, the BWW ontology has been chosen as a reference ontology. The BWW ontology includes nearly fifty categories out of which the categories of $Thing$, $Property$, $Event$, and $Process$ are selected as reference ontological kinds. The first three categories are referred to as *intrinsic categories* in the classification of BWW categories presented in [8]. The rest of the categories from the BWW ontology are either specialized categories of the intrinsic categorizes or composite categories defined from them. Category $Process$ is a specialized category of $Event$. Once the programming abstractions are interpreted as intrinsic categories, further interpretation techniques can be developed to interpret programming abstractions into more specialized ontological kinds.

Within the scope of this work, we interpret software objects with properties associated to them as things. These may turn out to be physical things such as employees or health records, or they may also be conceptual things such as internal data structures, relational tables and files that have existence inside the memory of the computer.

B. Unit of Interpretation

While performing ontological interpretation of a programming abstraction, the construct $class$ is taken as a unit of interpretation, and its parts such as *attribute declaration* and *method declaration* are considered for interpretation of the class.

However, the parts are not individually categorized further as independent units of interpretation. If methods and attributes are to be considered as units of interpretation, they may be interpreted through categories such as $Property$ or $Event$. An attribute represents a property when its intention is to store the values. Similarly, a method represents a property when it is intended for accessing the stored values. Examples of such methods are simple accessor methods (e.g., $get()$), or complex accessors which compute and return a property value in terms of local variables. If a method is an accessor method, it may be further interpreted through specialization as *Functional Property*, a category from the BWW ontology. In contrast, in our case, instead of assigning categories to attributes and methods, they are assigned to their holder classes capturing the role played by the parts.

C. Strategies of Interpretation

A simple strategy is to interpret a class as a $Thing$ in which an attribute declaration is interpreted as $Property$, a method declaration as $Event$, with $Thing$ as participant of $Event$. However, such a simplified strategy of interpretation does not always hold in practice as it can be seen from classes $Address$ and $InsertEmployee$ in Figure 2.

Also, it can be noted that since the notion of $class$ is also a mechanism to realize a user defined data-type, it cannot be assigned a fixed ontological category. Therefore, an analysis of the internal features of a class is required for the interpretation of the class.

D. Class in Programming Languages versus Class in Ontology

It is to be noted that the notion of *Class* in OO programming languages and in the BWW ontology capture semantically different concepts. In the BWW ontology, $Class$ is a collection category that groups things together sharing a characteristic property. For example, the ontological class *Teenager* is a group of persons whose members are in the age group from thirteen to nineteen. In OO programming languages, the $Class$ construct is more of an implementation

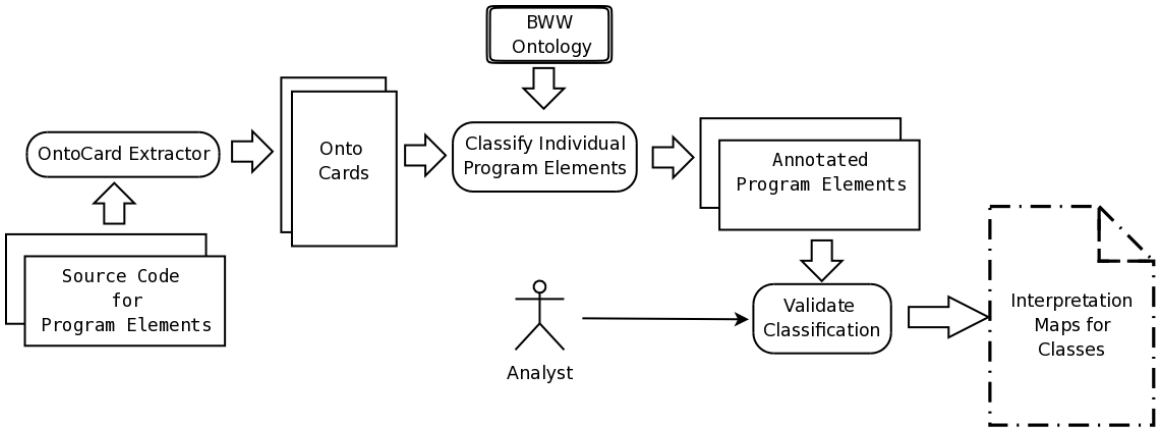


Fig. 3. Information flow in $OntoClassifier$ for Interpreting Programming Abstractions

mechanism which can represent any one of the ontological categories.

E. Single versus Multiple Interpretations

The interpretation strategy elaborated in this paper is based on the observation that in order to increase the re-usability of a programming abstraction and to control its size, programmers often factor out various features of a substantial entity into multiple smaller sized classes. An example of such a case can be seen in classes $Employee$ and $InsertEmployee$ shown in Figure 2. In this case, separate ontological categories are assigned to them. On the other hand, it may be a case that a programmer opts to represent the properties of a substantial thing and the events it participates in, through a single class. In such a case, the programming abstraction is interpreted as $Thing$, and no multiple assignments are made.

F. Role of Class Identifiers

In many cases, the linguistic category of names of programming abstractions (as in names $Employee$, $Address$, $InsertEmployee$ in Figure 2) may provide a hint to its corresponding ontological kind. In the automated scheme for deciding the ontological kinds discussed in this paper, no particular role is assigned to class identifiers of a programming abstraction since linguistic categorization of names may not be uniformly enforced by programmers. It has also been observed that programmers often violate the rules for well-formed identifiers [9]. But the linguistic categories of names can indeed be considered to improve the effectiveness of the classifier system when appropriate naming conventions are in effect. Furthermore, a human analyst may use class identifier categories in validating the ontological kinds assigned by an automated scheme.

V. $OntoClassifier$: A Rule-based Classifier System

This section describes the design of a classifier system which suggests the ontological category of a given programming abstraction. In this context, a classifier system is an algorithm, or a tool, or a technique that assigns categories to program elements. Few examples of classifier systems are rule based classifiers, clustering algorithms, and learning classifier systems [10]. To come up with strategies for systematic ontological interpretation of OO programming abstractions, first a manual analysis of around two hundred and fifty classes from the application $DSpace$ [6] was performed.

This phase can be broadly viewed as *learning phase* in the approach to information interpretation. The purpose of the *learning phase* has been to answer the following questions: (i) Can one judge the ontological category of the given programming abstraction based on implementation features? (ii) If the answer is "yes", what are those implementation features? (iii) How do the interactions among implementation features influence ontological categories? The outcomes of the learning phase are twofold. Firstly, it has resulted in identification of syntactic *features* necessary for ontological interpretation. These features are captured through an analysis card called *Onto-card* which is described in Section VII-B. Secondly, it has resulted in identification of a set of mapping rules to interpret programming abstractions into ontological categories given their ontological features captured through Onto-cards.

The activity of ontological interpretation is organized into three phases as shown in Figure 3. Firstly, the set of program features included in the *Onto-card* are extracted from individual programming abstractions. Secondly, each programming abstraction is classified into one of the four categories from the BWW ontology, which are $Thing$, $Property$, $Event$ and $Process$. An Eclipse-based plug-in is developed to automate the task of categorization. The plug-in extracts the *Onto-card* features from the source code. The plug-in then applies categorization rules to identify ontological kinds of programming abstractions. Finally, the ontological categorization performed by the plug-in is validated by a human analyst.

VI. Ontological Cards

Programming abstractions include implementation details such as name of the abstraction, type definitions for attributes and methods, implementations of methods and keywords and

Fully Qualified Name of a Class		
Member Analysis		
A class has attributes declaration.	$hasAttributes$	Y/N
A class has methods declaration.	$hasMethods$	Y/N
All the attributes and methods defined in a class are $static$.	$hasAllMembersStatic$	Y/N
A class has the method $main$ defined in it.	$hasMain()$	Y/N
A class has the method run defined in it.	$hasRun()$	Y/N
A class has at least one method containing a looping construct.	$hasMethodContainingLoop$	Y/N
A class implements a non-null constructor method.	$hasConstructor$	Y/N
A class invokes methods from another class.	$hasInteractions$	Y/N

Fig. 4. Template for the Onto-card

comments. The task of identifying an ontological kind becomes difficult in the presence of such detailed representation. With an intention to overcome this difficulty, *Onto-cards* are introduced. Ontological cards document the presence or absence of certain syntactic and relational features. The cards include only those features that either directly capture some ontological postulates from the BWW ontology or form finer syntactic features that can differentiate classes that actually belong to different ontological kinds in spite of they having largely similar syntactic characteristics.

Figure 4 shows the template of an ontological card. The feature $hasAttributes$ indicates the presence of attributes and the feature $hasMethods$ indicates the presence of methods in a class definition. They capture the ontological postulates *things have properties*, *things are in states* and *events change states of a thing*. Properties and states often get represented through attributes, while methods provide mechanism to change states. The feature $hasAllMembersStatic$ indicates that the class's all members are static. This feature identifies the class as a singleton. The singleton class refers to a situation where a programmer intends to create only one instance of the class. (Note: The term singleton does not convey a solution to the problem of creating a single instance as captured through the singleton pattern in [11]) The feature $hasInteractions$ indicates the presence of outgoing interaction, and the feature $hasMethodContainingLoop$ indicates the presence of iteration. They are included to distinguish between classes that represent things and classes that solely represent abstract properties. The classes that represent abstract properties normally have accessor methods that contain no looping constructs and no external method calls. The feature $hasConstructor$ indicates the presence of an explicit constructor. It distinguishes classes that represent things and classes that represent events. Explicit non-null constructors are not usually found in classes that represent events. The features $hasMain()$ and $hasRun()$ indicate the presence of methods $main()$ and $run()$. They capture event composition by encapsulating invocation sequences.

VII. Categorization Rules

A *decision rule*-based technique is used for ontological categorization. The rules are described using the form $OCF^n \Rightarrow Ontological\ Category$, where OCF^n is a conjunction of n ontological features from the *Onto-card* including predicates indicating both presence and absence of certain syntactic features. The inference is shown on the right hand side of the inference operator $\Rightarrow$. The rules of categorization are discussed in this section, while an analysis of application of these rules is discussed in the next section.

Inclusion of a syntactic feature in a given rule is guided by two criteria. Firstly, inclusion of the syntactic feature should support some ontological postulates described in [3]. For example, the feature $hasAttributes$ is included in $Rule\ 5$ because it supports the ontological postulate *things have properties*. Secondly, inclusion of a feature is based on the association of a feature to a certain category. For example, classes having no explicit constructors are found to represent events, and this fact is used in $Rule\ 8$. Such patterns of associations have been learnt during the learning phase i.e. the manual analysis of application $DSpace$. In the learning phase, a feature matrix and ontological interpretation for every class was manually prepared in order to gain an understanding on the correlation of combinations of programming features and ontological categories. It may be noted that the decision rules were constructed via a manual analysis in order to get the first insight into identification of the set of features laying a basis for this approach. Also the postulates from the BWW ontology were kept in mind while formulating this set of features.

A. Rules to Identify Properties

Classes with only attributes defined in them or classes that do not interact with other classes are interpreted as properties. Classes having only attributes are not considered as things, since things are different from properties in the sense that things are expected to possess behavior. Also, classes having only attributes are not events as they do not represent changes of states in things.

$Rule\ 1$ described below captures a scenario in which classes with only attributes are mapped as properties.

$$\neg hasMain() \wedge \neg hasRun() \wedge hasAttributes \wedge \neg hasMethods \Rightarrow Property \quad (\textbf{Rule 1})$$

In the BWW ontology, things are interacting objects. Hence, classes without external interactions may not always represent things. Other features are explored in such a case. $Rule\ 2$ and $Rule\ 3$ given below identify $Properties$ from among such classes.

$$\neg hasMain() \wedge \neg hasRun() \wedge hasAttributes \wedge hasMethods \wedge \neg hasAllMembersStatic \wedge \neg hasConstructor \wedge \neg hasInteractions \Rightarrow Property \quad (\textbf{Rule 2})$$

$Rule\ 3$ given below differs from $Rule\ 2$ with respect to the presence or absence of non-null constructors. When such a constructor is present, the class may be a candidate for both categories $Thing$ and $Property$. In this case, the feature

$hasMethodContainingLoop$ is used to distinguish classes representing properties from classes representing things.

$$\neg hasMain() \wedge \neg hasRun() \wedge hasAttributes \wedge hasMethods \wedge \neg hasAllMembersStatic \wedge hasConstructor \wedge \neg hasInteractions \wedge \neg hasMethodContainingLoop \Rightarrow Property \quad (\textbf{Rule 3})$$

B. Rules to Identify Things

In the BWW ontology, things have properties, things are in states that undergo changes, and things interact with other things. Classes that have attributes, mechanisms to change states, method invocations on objects of other classes, and explicit non-null constructors are considered to represent things. The following $Rule$ 4 maps classes having above mentioned syntactic features to things.

$$\neg hasMain() \wedge \neg hasRun() \wedge hasAttributes \wedge hasMethods \wedge \neg hasAllMembersStatic \wedge hasConstructor \wedge hasInteractions \Rightarrow Thing \quad (\textbf{Rule 4})$$

In the following $Rule$ 5, a scenario in which classes that do not have external interactions but are likely to represent things are captured. Though such classes do not interact on their own, they define methods for external things to interact with them. The defined methods are expected to be complex as indicated by the presence of the feature $hasMethodContainingLoop$. Besides detecting the presence of a loop, other methods such as structural complexity can also be used to formulate refinements of this rule.

$$\neg hasMain() \wedge \neg hasRun() \wedge hasAttributes \wedge hasMethods \wedge \neg hasAllMembersStatic \wedge hasConstructor \wedge \neg hasInteractions \wedge hasMethodContainingLoop \Rightarrow Thing \quad (\textbf{Rule 5})$$

In the following $Rule$ 6, a scenario in which classes that have all members static is captured. Such classes are mapped to things. The programmer's intention behind making members static is mainly to create a singleton thing. A singleton thing is a thing for which only one instance of the class exists, or in other words, the class itself is used as an object. For example, a single instance of authorization manager may be used a unique authority for authorization of a request. Such classes interpreted as things also have static attributes.

$$\neg hasMain() \wedge \neg hasRun() \wedge hasAttributes \wedge hasMethods \wedge hasAllMembersStatic \wedge hasMethodContainingLoop \Rightarrow Thing \quad (\textbf{Rule 6})$$

C. Rules to Identify Events

In the BWW ontology, changes in states of things represent events. Events in reality are mainly short-lived objects or the objects that exist for a while as compared to things that participate in events. For example, *running for a short distance* and *reading a book* are examples of events that exist for certain duration, and the duration for which these events exist is smaller than the lifetime of participant things.

While mapping classes to events, methods defined in a class are interpreted as mechanisms that cause changes in states of things which may even be supplied in parameters. Classes that encapsulate only methods are mapped to events by the following $Rule$ 7. In $Rule$ 7, the use of feature $\neg hasMain()$ indicates the absence of the method $main()$ in a class. It minimizes the possibility of mis-categorization of a class that represents $Process$ to $Event$. Similarly, the use of feature $\neg hasAttributes$ indicates absence of attributes. It minimizes the possibility of a misclassification of classes representing $Thing$ and $Property$ to $Event$.

$$\neg hasMain() \wedge \neg hasRun() \wedge \neg hasAttributes \wedge hasMethods \Rightarrow Event \quad (\textbf{Rule 7})$$

The following $Rule$ 8 considers the case of classes having both methods and attribute definitions. In this rule, the feature $\neg hasConstructor$ is included on the observation that a large number of classes that do not define a non-null constructor represent events.

$$\neg hasMain() \wedge \neg hasRun() \wedge hasAttributes \wedge hasMethods \wedge \neg hasAllMembersStatic \wedge \neg hasConstructor \wedge hasInteractions \Rightarrow Event \quad (\textbf{Rule 8})$$

In the following $Rule$ 9, a scenario in which programmers develop static classes to define utility methods such as methods for converting data formats is captured. Such classes are mapped to events and the feature $hasMethodContainingLoop$ is used to differentiate utility method classes from singleton things.

$$\neg hasMain() \wedge \neg hasRun() \wedge hasAttributes \wedge hasMethods \wedge hasAllMembersStatic \wedge \neg hasMethodContainingLoop \Rightarrow Event \quad (\textbf{Rule 9})$$

D. Rules to Identify Processes

In the BWW ontology, processes are complex objects specifying sequential composition of events. Processes, like events, are short-lived objects as compared to things that participate in them. In following $Rule$ 10 and 11, the presence of special methods such as $main()$ and $run()$ is considered as mechanisms to compose events sequentially.

$$\neg hasMain() \wedge hasRun() \Rightarrow Process \quad (\textbf{Rule10})$$
$$hasMain() \Rightarrow Process \quad (\textbf{Rule 11})$$

VIII. Evaluation of the Approach

A. Application Softwares

The following two application softwares are used to evaluate the approach.

Class Name	Category Assigned By		
	Rule	Classifier	Domain Expert
HealthWatcher			
lib.exceptions.ExceptionMessages	Rule1	Property	Property
healthwatcher.model.address.Address	Rule3	Property	Property
healthwatcher.model.complaint.DiseaseType	Rule3	Property	Property
healthwatcher.business.healthguide.HealthUnitRecord	Rule4	Thing	Thing
lib.concurrency.ConcurrencyManager	Rule5	Thing	Thing
healthwatcher.view.command.InsertEmployee	Rule7	Event	Event
healthwatcher.view.servlets.HWServlet	Rule8	Event	Thing
healthwatcher.Constants	Rule9	Event	Property
healthwatcher.data.factories.RepositoryFactory	Rule9	Event	Thing
healthwatcher.business.HWServer	Rule11	Process	Thing
ConStore			
con.core.ConStoreConstants	Rule1	Property	Property
con.clustering.CliqueClustering	Rule2	Property	Event
con.io.FileAdapter	Rule3	Property	Thing
con.ds.Node	Rule3	Property	Thing
con.caching.MRUCache	Rule4	Thing	Thing
con.net.FileConceptNet	Rule4	Thing	Thing
con.util.Int2DHeapSort	Rule6	Thing	Event
con.indexing.IndexBuilder	Rule7	Event	Event
con.util.Int2DBinarySearch	Rule7	Event	Event
con.core.RelationInstance	Rule8	Event	Property
con.core.DataType	Rule9	Event	Property
con.clustering.InstanceCluster	Rule11	Process	Process
con.core.EntityInstance	Rule12	null	Thing

TABLE II. ONTOLOGICAL INTERPRETATION OF A FEW CLASSES

1) HealthWatcher: The application $HealthWatcher$ [7] is a Java-based health information system that can be used by health offices to store and handle health related complaints. Employees of health office and citizens act as users of this application. Citizens can use this application to file complaints, query the status of complaints and get disease related information. Employees can use the application for handling complaints, and accessing user related information. The application is chosen because it has earlier been used by many researchers [12] as a test-bed.

2) ConStore: The application $ConStore$ [13] is a small sized Java-based open source storage software to manage and access knowledge bases represented through concept maps. The application $ConStore$ allows creation, storage and retrieval of entities and relations in concept maps. Querying and clustering of concepts are supported.

Ontological categorization of some of the programming abstractions from the applications of $ConStore$ and $HealthWatcher$ are listed in Table II

B. Performance Measures

The effectiveness of the categorization rules can be measured through the measures that are used to evaluate automated rule-based classifier systems. These measures are (i) Confusion Matrix, (ii) Recall and Precision of the classifier system, and (iii) Accuracy of individual rules. The measures evaluate the performance of the automated classifier system against a validated classification.

Confusion matrix [14] is a table-based technique to assess classifier systems. An entry in the cell C_{ij} of the confusion matrix represents the number of programming abstractions belonging to i^{th} category predicted as j^{th} category by the classifier. Non-zero non-diagonal elements of the confusion matrix represent errors in classification. Table III shows the confusion matrix obtained after applying mapping rules to classes from the applications $ConStore$ and $HealthWatcher$.

Definitions of *recall* and *precision* as used in document categorization and information extraction systems [15] are adopted to evaluate the effectiveness of the BWW ontological classifier system.

$$Recall = \frac{No.\ of\ classes\ correctly\ assigned\ to\ Cat_X}{No.\ of\ classes\ \textbf{belonging}\ to\ Cat_X}$$

$$Precision = \frac{No.\ of\ classes\ correctly\ assigned\ to\ Cat_X}{No\ of\ classes\ \textbf{assigned}\ to\ Cat_X}$$

$$Accuracy\ of\ a\ Rule = \frac{No.\ of\ correct\ assignments\ made}{Total\ no.\ of\ assignments\ made}$$

The values of recall and precision are calculated from the confusion matrix. Table III depicts the performance results after applying the approach to applications $ConStore$ and $HealthWatcher$.

From Table IV, it can be observed that accuracy of the rules that implement the mapping strategy of classes having only attributes to *Properties* (i.e. Rule 1), classes having only methods to *Events* (i.e. Rule 7) and classes having attributes and methods to *Things* (i.e. Rule 4) have greater accuracy. Rules 1, 4 and 7 are purely based on ontological postulates.

Accuracy of the rules that use a combination of implementation features such as $hasAllMembersStatic$, $hasInteractions$ and $hasConstructor$ is relatively low. Precise identification of an ontological category becomes difficult

Confusion Matrix

HealthWatcher

	Property	Thing	Event	Process
Property	4	7	19	1
Thing	1	35	14	1
Event	0	9	43	0
Process	0	0	0	0

ConStore

	Property	Thing	Event	Process
Property	4	3	2	0
Thing	13	16	1	0
Event	1	2	8	0
Process	0	0	0	2

Recall and Precision

HealthWatcher

	Recall	Precision
Property	0.12	0.8
Thing	0.69	0.69
Event	0.82	0.57
Process	0	0

ConStore

	Recall	Precision
Property	0.45	0.22
Thing	0.53	0.76
Event	0.72	0.72
Process	1	1

TABLE III. EVALUATING PERFORMANCE OF $OntoClassifier$ TOOL THROUGH CONFUSION MATRIX

	HealthWatcher			ConStore		
Rule No.	Total Decisions	Correct Decisions	Accuracy	Total Decisions	Correct Decisions	Accuracy
Rule1	1	1	1	1	1	1
Rule2	0	0	N. A.	2	0	0
Rule3	4	3	0.75	15	3	0.2
Rule4	50	34	0.68	16	14	0.875
Rule5	1	1	1	0	0	N. A.
Rule6	0	0	N. A.	5	2	0.4
Rule7	69	41	0.59	9	8	0.88
Rule8	2	2	1	1	0	0
Rule9	5	0	0	1	0	0
Rule10	0	0	N. A.	0	0	N. A.
Rule11	2	0	0	2	2	1
Rule12	1	0	0	1	0	0
Overall Accuracy	135	82	0.60	53	30	0.56

TABLE IV. ACCURACY OF CLASSIFICATION RULES

when programmers factor out the properties of an ontological thing and events that happen to the thing among multiple classes. Furthermore, such classes include definitions of both methods and attributes, which makes an $Event$ or a $Property$ appear like a $Thing$. Combination of additional implementation features such as $hasAllMembersStatic$, $hasInteractions$ and $hasConstructor$ are required to identify the ontological category in such cases. It can be noted that for some cases, differentiating between things and properties is difficult even for the analyst, especially when a distinction between a *role* and a *substantial thing* is to be made. For example, a particular instance of *a student* is a thing, but at the same time *being a student* is the property. Simpler classes tend to get identified as $Properties$.

It can be seen that in the two applications, there was no case of applicability of Rule 9 that has been introduced with an intention to distinguish classes representing $Thing$ from classes representing $Event$. Use of enriched features such as cyclomatic complexity can improve the accuracy of Rule 9.

One of the reasons for low values for recall and precision for some of the rules is that the categorization rules use only a limited set of syntactic features. For example, method and attribute definitions in the super classes, and access patterns are not considered presently. However, the effectiveness of the classifier system is satisfactory considering that only a few syntactic features are employed for ontological categorization, and that linguistic knowledge is not used. The following techniques are suggested for performance improvement: (i) Rule refinement through additional features. (ii) Association mining to correlate *Onto-card* features with ontological categories. (iii) Use of *continuous learning* classifier, in which the classifier system can evolve over a period of time by tracking the interpretations for correcting errors observed in previous runs. (iv) Use of semantic features such as linguistic categories of program terms, and domain knowledge.

IX. RELATED WORK

The task of assigning ontological categories to OO programming abstractions is viewed as a *classification problem* in the approach presented in this paper. The existing approaches that classify OO program elements can be described along three dimensions as shown in Table V. These are (i) the *goals* supported by a classification approach, (ii) the types of *classification categories* used in an approach and (iii) the type of *classifier system* employed.

Three classification approaches which are explicit about these dimensions are reviewed in this section. The classification approach proposed by Damiani et al. [16] uses a set of behavioral features to classify OO program elements. Some of the examples of behavioral features are methods defined in a program element (e.g., classes and interfaces) and their relevance to the overall functionality of the program element. The approach semi-automatically assigns multiple categories to program elements. Clarke et al. define a classification approach which identifies an appropriate *testing technique* to test a given class [17]. Manza et al. suggest a classification approach to improve *the understanding of classes* revealing their internal structure [21]. The approach employs a colored

Sr. No.	The Approach by	Goals	Features of the Classifier System used	Example Classification Categories
1	Damiani et al. [16]	The classification approach is aimed at improving reusability of classes by grouping them according to the services provided by them. It also aims to effectively retrieve a desired component from a component repository.	Classification categories are *semi-automatically* assigned to OO classes. Firstly, classification categories are manually assigned to the root class of a classification hierarchy. The functionalities provided by classes decide the categories. Secondly, sub-classes inherit the classification categories of super-classes. Thirdly, new categories are manually assigned to sub-classes based on the functionalities specific to a sub-class.	(i) File system services, (ii) Inter process communication services, (iii) User interface development services, and (iv) Graphic and drawing support services.
2	Clarke et al. [17]	To select an appropriate testing technique for a given class.	OO classes and classification categories (i.e. testing techniques) are described through a common template. The template consists of fields such as modifiers (e.g., public, final, virtual) which are normally associated to methods and attributes, and a field to indicate whether the class is a subclass. A *mapping algorithm* is used to assign an appropriate testing technique. The algorithm matches the features describing a class under test and features in the template describing a testing technique. A testing technique with the highest priority among the selected testing techniques is assigned as the most appropriate technique to test the given class.	(i) The *Harold's Data Flow Testing Technique* [18] which is appropriate for component interaction testing. (ii) The *Harold's Incremental Testing Technique* which is appropriate [19] for classes with no superclasses. (iii) The *Kopol's integration testing* [20] which is appropriate for classes with concurrent methods.
3	Manza et al. [21]	To improve understanding of classes by visually revealing their internal structure and by annotating them through meaningful labels.	Classification categories are *manually* assigned to OO classes. A metric-based analysis is performed. Program element metrics such as LOC of a method body, the number of internal invocations made by a method, the number of direct accesses of attributes from within and outside the class are used to assign categories.	(i) Data storage, (ii) Wide interface, and (iii) Large implementation.

TABLE V. APPROACHES FOR CLASSIFYING OO PROGRAM ELEMENTS

visualization scheme and a set of descriptive labels to reveal internal structure of classes. These approaches are summarized in Table V.

From Table V, it can be observed that the classification categories used in these approaches focus on low level implementation details. Classifying OO classes along these dimensions is useful from the points of view of software reuse and supporting down-stream software engineering activities such as software testing and maintenance. On the other hand, the approach presented in this paper interprets ontological kinds of programming abstractions for supporting up-stream software engineering activities of architecture reconstruction and recovery with classification categories being ontological interpretations.

Ontological interpretation of programming abstractions is one of the ways to ontologically analyze source code. Performing *similarity* (an ontological relationship) analysis is another way to analyze source code ontologically. Similarity analysis of the source code is mainly performed to detect codes which are high-level concept clones [22], to detect mismatches between UML design and OOPL implementation [23], and to cluster source code components [24].

Source code can also be analyzed either to extract or to build an application ontology, i.e. to extract either high-level or programming concepts, and relationships between them in a given application. Two such approaches are by Abebe et al. [25] and Würsch et al. [26]. Lexicons and taxonomies are primitive forms of ontologies in which concepts are identified and their meanings are defined. Host et al. [27] build a lexicon of method names and Gil et al. [28] build a taxonomy of patterns of Java classes.

X. DISCUSSION

Performing ontological analysis of *software modeling languages* with an aim to evaluate them is a well recognized idea [29], [30], [31], [32], [33], [34]. The approach presented in this paper extends the idea of applying ontological analysis to *existing application softwares*. The motivation behind performing ontological analysis of application softwares comes from the observation that program elements comprising application softwares represent real life phenomena. If we establish the correspondence of program elements with the categories of objects found in the reality, we can better understand the roles played by program elements and relationships among them.

The main contribution of the approach presented in this paper is that it formulates the problem of *ontological interpretation of OO programs* as a *classification problem*. The approach proposes a rule based solution to the problem of interpretation. To interpret programming abstractions, classes from OO applications are abstractly represented through an ontological card called *Onto-card*. The *Onto-card* uses a small number of syntactic features. We have restricted ourselves to not to use any semantic features such as user assigned names to classes and their meanings, attributes and method implementations for deciding the ontological category. The decision rules employed in the approach formally capture the rationale behind assigning ontological categories to program elements.

The use of interpretivist approaches i.e. meaning based approaches [35] have been observed to suffer from the limitation of *subjectivity* [36]. Despite this limitation, the interpretivist approaches have been found to be useful in processes such as natural language translation. One of the reasons behind multiple meanings getting assigned to a piece of information is the use of multiple criteria for the task of interpretation. For an example, three different meanings have been assigned to the UML abstraction $Actor$ by two different researchers. The meanings assigned to the notion of $Actor$ are $Mutual\ Property$ and $Kind$ by G. Irwin et al. [31] and $Thing$ by Opdahl et al. [34]. This conflict in interpretation is due to the fact that Irwin et al. base their interpretation on

structural properties used to realize the notion of *Actor* in UML metamodel, while Opdahl et al. base their interpretation on modeler's intention when a modeler uses the notion of *Actor* to represent a fact from reality. The techniques that are used to handle the problem of subjectivity in the field of natural languages [37] can be explored and applied to interpretation of modeling languages. Thus the act of interpreting programming abstractions that handles the issue of subjectivity requires new techniques.

One of the possible applications of the technique developed in this paper is recovery of higher-level software models from given source code. During ontological interpretation, we assign ontological categories to programming abstractions. Because ontological categories are the most generic categories describing various real-world phenomena, they provide an effective metaphor that can be uniformly applied throughout software system analysis and design. Such an analysis can assist in ensuring *conceptual integrity* among software elements at multiple levels of abstractions such as implementation, design, architecture and requirements.

REFERENCES

[1] S. Nirenburg and V. Raskin, *Ontological Semantics (Language, Speech, and Communication)*. The MIT Press, 2004.

[2] D. Lewis, "General semantics," in *The Semantics of Natural Language*, D. Davidson and Harman, Eds. Amsterdam: Kluwer, 1972.

[3] M. Bunge, *Treatise on Basic Philosophy (Vol 3): Ontology I : The Furniture of the World*, 1st ed. D. Reidel Publishing Company, 1977.

[4] W. Yair and R. Weber, "On the deep structure of information systems," *Information System Journal*, no. 5, pp. 203–223, 1995.

[5] C. K. Ogden and I. Richards, *The meaning of meaning*. London: Trubner & Co, 1923.

[6] www.dspace.org, "DSpace."

[7] www.comp.lancs.ac.uk, "Healthwatcher system java implementation."

[8] A. W. Kiwelekar and R. K. Joshi, "An object oriented metamodel for bunge-wand-weber ontology," in *In Proc. of SWeCKa 2007, Workshop on Semantic Web for Collaborative Knowledge Acquisition at IJCAI 2007*, January 2007.

[9] D. Lawrie, H. Feild, and D. Binkley, "An empirical study of rules for well-formed identifiers," *Journal of Software Maintenance*, vol. 19, no. 4, pp. 205–229, 2007.

[10] T. Mitchell, *Machine Learning*, 1st ed. McGraw Hill, 1997.

[11] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[12] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid, "On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study," in *ECOOP 2007 Object-Oriented Programming*, ser. Lecture Notes in Computer Science, E. Ernst, Ed. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2007, vol. 4609, ch. 9, pp. 176–200.

[13] www.cse.iitb.ac.in/ čonstore/, "Constore a storage and retrieval system for concept-nets."

[14] R. Kohavi and F. Provost, "Glossary of terms," *Machine Learning*, vol. 2, no. 3, pp. 271–274, 1998.

[15] M. Junker, A. Dengel, and R. Hoch, "On the evaluation of document analysis components by recall, precision, and accuracy," in *ICDAR '99: Proceedings of the Fifth International Conference on Document Analysis and Recognition*. Washington, DC, USA: IEEE Computer Society, 1999, p. 713.

[16] E. Damiani, M. G. Fugini, and C. Bellettini, "A hierarchy-aware approach to faceted classification of objected-oriented components," *ACM Trans. Softw. Eng. Methodol.*, vol. 8, no. 3, pp. 215–262, 1999.

[17] P. J. Clarke and B. A. Malloy, "A taxonomy of oo classes to support the mapping of testing techniques to a class," *Journal of Object Technology*, vol. 4, no. 5, pp. 95–115, 2005.

[18] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," in *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*. New York, NY, USA: ACM, 1994, pp. 154–163.

[19] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick, "Incremental testing of object-oriented class structures," in *ICSE '92: Proceedings of the 14th international conference on Software engineering*. New York, NY, USA: ACM, 1992, pp. 68–80.

[20] P. V. Koppol, R. H. Carver, and K.-C. Tai, "Incremental integration testing of concurrent programs," *IEEE Trans. Softw. Eng.*, vol. 28, no. 6, pp. 607–623, 2002.

[21] M. Lanza and S. Ducasse, "A categorization of classes based on the visualization of their internal structure: the class blueprint," in *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2001, pp. 300–311.

[22] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *ASE*. IEEE Computer Society, 2001, pp. 107–114.

[23] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella, "Design-code traceability for object-oriented systems," *Ann. Software Eng.*, vol. 9, pp. 35–58, 2000.

[24] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *ICSE*, H. A. Müller, M. J. Harrold, and W. Schäfer, Eds. IEEE Computer Society, 2001, pp. 103–112.

[25] S. L. Abebe and P. Tonella, "Natural language parsing of program element names for concept extraction," in *ICPC*. IEEE Computer Society, 2010, pp. 156–159.

[26] M. Würsch, G. Ghezzi, G. Reif, and H. Gall, "Supporting developers with natural language queries," in *ICSE (1)*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 165–174.

[27] E. W. Høst and B. M. Østvold, "The programmer's lexicon, volume i: The verbs," in *SCAM*. IEEE, 2007, pp. 193–202.

[28] J. Gil and I. Maman, "Micro patterns in java code," in *OOPSLA*, R. E. Johnson and R. P. Gabriel, Eds. ACM, 2005, pp. 97–116.

[29] J. Evermann, "The association construct in conceptual modelling - an analysis using the Bunge ontological model," in *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, 2005, pp. 33–47.

[30] P. Green, M. Rosemann, M. Indulska, and C. Manning, "Candidate interoperability standards: An ontological overlap analysis," *Data Knowl. Eng.*, vol. 62, pp. 274–291, August 2007.

[31] G. Irwin and D. Turk, "An ontological analysis of use case modeling grammar," *J. AIS*, vol. 6, no. 1, 2005.

[32] A. L. Opdahl and B. Henderson-Sellers, "Grounding the OML metamodel in ontology," *J. Syst. Softw.*, vol. 57, pp. 119–143, June 2001.

[33] P. F. Green, M. Rosemann, and M. Indulska, "Ontological evaluation of enterprise systems interoperability using ebXML," *IEEE Transactions on Knoweledge and data Engineering*, vol. 17, no. 5, pp. 713–724, May 2005.

[34] A. Opdahl and B. Henderson-Sellers, "Ontological evaluation of the UML using the Bunge-Wand-Weber model," *Software and Systems Modeling J.*, vol. 1, no. 1, pp. 43–67, 2002.

[35] G. Walsham, "The emergence of interpretivism in IS research," *Information System Research*, vol. 6, December 1995.

[36] M. R. de Villiers, "Three approaches as pillars for interpretive information systems research: development research, action research and grounded theory," in *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, ser. SAICSIT '05. , Republic of South Africa: South African Institute for Computer Scientists and Information Technologists, 2005, pp. 142–151.

[37] C. Akkaya, J. Wiebe, and R. Mihalcea, "Subjectivity word sense disambiguation," in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1 - Volume 1*, ser. EMNLP '09. Stroudsburg, PA, USA: Association for Computational Linguistics, 2009, pp. 190–199.

List of Authors