05261 Abstracts Collection
Multi-Version Program Analysis
— Dagstuhl Seminar —

Thomas Ball[1], Stephan Diehl[2], David Notkin[3] and Andreas Zeller[4]

[1] Microsoft Research - Redmond, US
tball@microsoft.com
[2] KU Eichstätt, DE
diehl@cs.uni-sb.de
[3] Univ. of Washington, US
notkin@cs.washington.edu
[4] Univ. Saarbrücken, DE
zeller@cs.uni-sb.de

Abstract. From 26.06.05 to 01.07.05, the Dagstuhl Seminar 05261 "Multi-Version Program Analysis" was held in the International Conference and Research Center (IBFI), Schloss Dagstuhl. During the seminar, several participants presented their current research, and ongoing work and open problems were discussed. Abstracts of the presentations given during the seminar as well as abstracts of seminar results and ideas are put together in this paper. The first section describes the seminar topics and goals in general. Links to extended abstracts or full papers are provided, if available.

Keywords. Software engineering, data mining, software processes, software archives, version control, bug database, experimentation, measurement, verification

05261 Summary – Multi-Version Program Analysis

Change is an inevitable part of successful software systems. Software changes induce costs, as they force people to repeat earlier assessments. On the other hand, knowing about software changes can also bring benefits, as changes are artifacts that can be analyzed.

In the last years, researchers have begun to analyze software together with its change history. There is a huge amount of historical information that can be extracted, abstracted, and leveraged:

- Knowing about earlier versions and their properties can lead to incremental assessments.
- Analyzing the history of a product can tell how changes in software are related to other changes and features.

Dagstuhl Seminar Proceedings 05261
Multi-Version Program Analysis
http://drops.dagstuhl.de/opus/volltexte/2006/560

– Relating properties to changes can help focusing on changes that cause specific properties.

In this Dagstuhl seminar, researchers that analyze software and its history have met and discussed for a full week, exchanging their ideas, and combining and integrating the techniques to build a greater whole. Clearly, understanding history can play a major role when it comes to understand software systems.

Keywords: Software engineering, data minig, software processes, software archives, version control, bug database, experimantation, measurement, verification

Joint work of: Ball, Thomas; Diehl, Stephan; Notkin, David; Zeller, Andreas

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2006/559

Software Evolution and Reliability @ Microsot

Thomas Ball (Microsoft Research - Seattle, USA)

My talk focuses on the application of multi-version analysis technology at Microsoft, encapsulated in three technologies that are widely used in the company: a tool (Sleuth) for determining which (of many) tests to run to test a change to the code; a tool (Max) for keeping track of the evolution of dependences in the code; the Watson error reporting infrastructure.

The future of mining software repositories

Daniel M. Germán (University of Victoria, CDN)

"Mining Software Repositories" is a new area of research. I will argue, however, that it is not the act of mining that matters, and instead, it is the analysis and the results found with the information mined. I will describe the work we have done in this area: understanding the software development process of open source projects, visualization of historical information, and the definition of metrics for fine-grained change.

Keywords: Historical information, visualization, metrics

Modeling History to Understand Software Evolution

Tudor Girba (Universität Bern, CH)

The histories of software systems hold useful information when reasoning about the systems at hand or when reasoning about general laws of software evolution. Over the past 30 years more and more research has been spent on understanding software evolution.

However, the approaches developed so far do not rely on an explicit meta-model, and thus, they make it difficult to reuse or compare their results.

In this presentation we argue that there is a need for an explicit meta-model for software evolution analysis. We define, Hismo, a meta-model in which history is modeled as a first class entity.

We demo our tool called Van and we show how the different analyses can be combined. Van is based on the Moose reengineering environment built at the Software Composition Group, University of Bern.

Keywords: Software evolution, meta-modeling, history, first class entity

A Graph Based Approach for Studying Change Propagation in Software Systems

Ahmed E. Hassan (University of Waterloo, CDN)

Software systems contain entities, such as functions and variables, which are related to each other. As a software system evolves to accommodate new features and repair bugs, changes occur to these entities. Developers must ensure that related entities are updated to be consistent with these changes.

This talk explores the question: How does a change in one source code entity propagate to other entities? We examine the benefits and costs associated with using a number of heuristics to predict change propagation.

We study the change propagation problem through a graph based approach which uses several graph theory concepts and ideas like Steiner Trees, graph density and diameter. We validate our results empirically using data obtained by analyzing the development history for large open source software systems. Our results show that historical information derived from the development history of a project is likely to better support software developers in propagating changes than simply using static code dependencies such as call, use, and define.

Keywords: Change Propagation, Source Control, Mining Software Repositories, Steiner Tree

Analyzing code that changes on the fly

Michael Hicks (University of Maryland - College Park, USA)

Software systems are imperfect, so software updates are a fact of life. While typical software updates require stopping and restarting the program in question, many systems cannot afford to halt service, or would prefer not to. Dynamic software updating (DSU) addresses this difficulty by permitting programs to be updated while they run.

DSU is appealing it is quite general and requires no redundant hardware. The challenge is in making DSU flexible, and yet safe and easy to use.

My students and I, along with collaborators at Cambridge, have developed two kinds of multi-version program analysis to improve the safety and ease of use of DSU systems for C programs. First, we have developed an "updateability analysis" which analyzes a dynamically updateable program prior to deployment and identifies code positions at which future updates may occur. For each position in a given candidate set, the analysis calculates sufficient restrictions on the form of updates that would ensure type safety. Second, we have developed a simple tool that identifies syntactic differences between program versions. For example, it indicates which functions or variables have changed, which of these have changed type, and so on.

We have used this tool to study the evolutionary trends of some C programs. It is also the foundation of another tool that semi-automatically generates a "dynamic patch" needed to upgrade a running program to the most current version.

There is still much work needed to achieve an ideal DSU system, particularly in trying to apply updates safely. While the updateability analysis can be used to ensure type-safety, we would prefer stronger guarantees. One idea is to go beyond pre-deployment analysis and compare the old and new program versions to discover particularly well-suited update points. The hope is that by examining the differences between code versions, rather than dependences within a single version, we can approximate points in a program's execution at which assumptions are the same in both versions. Another idea is that multi-version code analysis can allow more of a dynamic patch to be generated automatically; right now, the tool does a fairly simple syntactic comparison of types, but very little of code. Both of these problems, and DSU itself for that matter, are undecidable in general, so good heuristics based on practical experience will be critical for success.

Keywords: Dynamic software updating, updateability analysis

An Empirical Study of Code Clone Genealogies

Miryung Kim (University of Washington, USA)

It has been broadly assumed that code clones are inherently bad and that eliminating clones by refactoring would solve the problems of code clones. To investigate the validity of this assumption, we developed a formal definition of clone evolution and built a clone genealogy tool that automatically extracts the history of code clones from a source code repository. Using our tool we extracted clone genealogy information for two Java open source projects and analyzed their evolution.

Our study contradicts some conventional wisdom about clones. In particular, refactoring may not always improve software with respect to clones for two reasons. First, many code clones exist in the system for only a short time; extensive refactoring of such short-lived clones may not be worthwhile if they are likely

diverge from one another very soon. Second, many clones, especially long-lived clones that have changed consistently with other elements in the same group, are not easily refactorable due to programming language limitations. These insights show that refactoring will not help in dealing with some types of clones and open up opportunities for complementary clone maintenance tools that target these other classes of clones.

This work will be also presented at the joint meeting of the 10th European Software Engineering Conference and the 13th Symposium of Foundations of Software Engineering.

Keywords: Code clone, software evolution, refactoring

Joint work of: Kim, Miryung;Sazawal, Vibha;Notkin, David; Murphy, Gail;

Full Paper:
http://www.cs.washington.edu/homes/miryung/

See also: I am a graduate student working with Dr. David Notkin at the University of Washington

Is this Good/Bad Software Evolution?

Sung Kim (Univ. California - Santa Cruz, USA)

Software evolves over time. By observing its evolution pattern, could we identify whether an evolution pattern is good or bad? If it is feasible to find good and bad patterns, we can encourage following the good evolution patterns. In the talk we try to identify good and bad evolution patterns by observing the occurrences of defects during software evolution. We associated the number of defects to the function level called defect density. Based on the defect density we grouped functions into two groups: stable group and unstable group. Using defect related factors, such as number of changes, number of authors, signature changes, and software complexity metrics, we tried to find unique evolution patterns of each group. We observed that functions in the unstable group have high number of signature changes, LOCs, and signature ordering changes.

A Song and Dance Number

Stephen Kobourov (University of Arizona, USA)

Real-world programs are large, complex, evolving, distributed, interactive, and resource-demanding. They are built by continuously evolving software engineering teams with members of varying skill levels, and kept up-to-date by different teams of software maintenance engineers. The Programmer's Cockpit is our vision of a system that integrates what are typically thought of as disparate activities: browsing program text, debugging, and performance evaluation. In

particular the Programmer's Cockpit will gather performance data, extract historical information from a version control database, as well as visualize and auralize the program under consideration.

Call graphs, inheritance graphs, heap graphs, and package graphs extracted from a program under study can be visualized using a computer game-style geographic metaphor. In call graphs, for example, the vertices of the graph can be represented as cities and the edges as roads or rivers. Many open source game engines are available, allowing for appealing alternative visualizations. Extending existing visualization techniques for both static and dynamics graph to the geographic metaphor poses many interesting research problems.

Audio output can be used to augment the typically visual output as human hearing is very sensitive to the differences in periodic and aperiodic signals and can detect small changes in the frequency of continuous signals. Rapidly evolving data can easily be missed in a graphical display, while it can be easily heard in the audio output. Certain sound cues are well suited for representing certain types of program information, such as repeated patterns in the control flow of the program. Also, unlike visual perception, sound perception does not require the focus of the listener, allowing audio cues to augment the perception of a user whose eyes are already busy with a visual task.

A Conservative Approximation of Program Analysis

Shriram Krishnamurthi (Brown Univ. - Providence, USA)

This is an introductory talk on the rudiments of program analysis, presented in a multi-program context, with a brief survey of some applications of program analysis to multiple program versions.

Keywords: Program analysis, dataflow, slicing, dicing, chopping, multi-version, differencing

Multi-Version Challenges in Post-Deployment Debugging

Ben Liblit (University of Wisconsin - Madison, USA)

The Cooperative Bug Isolation Project (CBI) is developing a suite of tools and analysis techniques that leverage the power of large user communities to iteratively diagnose and fix bugs in widely deployed software systems. Critical to the CBI approach is the concept of statistical debugging: bug hunting algorithms that diagnose failures by identifying statistical trends in noisy, incomplete feedback reports from end users. We discuss the general CBI approach to software quality and present one statistical debugging algorithm in detail. Using this as motivation, we consider several challenges and opportunities that arise in applying statistical debugging to multiple versions of released software. These include feedback aggregation, simulated bug fix impact prediction, and longitudinal tracking of bug life cycles.

Keywords: Bug isolation, feature selection, statistical debugging, invariants, dynamic analysis, random sampling, data mining, distributed feedback

Full Paper:
http://www.cs.wisc.edu/cbi/

A Framework for Multi-Version Program Analysis

Welf Löwe (Växjö University, S)

Each software comprehension tool defines an abstract software model, views on this model, analyses creating the model, and a mapping between model and view. For creating new comprehension tools, we configure online analyses, models, views and their mappings instead of hand-coding them. The talk introduces an architecture allowing such an online configuration and, as a proof of concept, a framework implementing this architecture. In several examples, we demonstrate generality and flexibility of our approach.

Does it generalize to Multi-Version Program Analysis, as well?

Keywords: Framework for Multi-Version Program Analysis

Joint work of: Löwe, Welf; Panas, Thomas; Lincke, Rüdiger

Software Changes and Software Engineering: Why Not?

Audris Mockus (Avaya - Basking Ridge, USA)

I discuss things that are to be avoided when analyzing software repositories. I illustrate the issues of irrelevant topic, narrow audience, and gross errors based on (mostly) my own previous reports.

Leveraging Program Multiplicities to Support (Distributed Continuous) Quality Assurance

Adam Porter (University of Maryland - College Park, USA)

Quality assurance (QA) tasks, such as testing, profiling, and performance evaluation, have historically been done in-house on developer-generated workloads and regression suites. The shortcomings of in-house QA efforts are well-known and severe, including (1) increased QA cost and schedule and (2) misleading results when the test-cases, input workload, software version and platform at the developer's site differs from those in the field. Consequently, tools and processes are being developed to improve software quality by increasing user participation in the QA process. A limitation of these approaches is that they focus on isolated mechanisms, not on the coordination and control policies and tools needed to

make the global QA process efficient, effective, and scalable. To address these issues, we have initiated the Skoll project, which is developing and validating novel software QA processes and tools that leverage the extensive computing resources of worldwide user communities in a distributed, continuous manner to significantly and rapidly improve software quality.

In this talk, I will present an infrastructure, algorithms and tools for developing and executing through, transparent, managed, and adaptive DCQA processes. I will also discuss how our DCQA processes leverage program multiplicities to improve process effectiveness and efficiency.

Keywords: Distributed Continuous Quality Assurance Empirical Software Engineering

Predictive Software Models and Software Change

Jelber Sayyad-Shirabad (University of Ottawa, CDN)

Software engineering is a decision intensive discipline. Experience plays an essential role in making the right decision. There is a wealth of knowledge and past experience hidden in resources such as software repositories. Machine learning provides techniques and algorithms to learn from the results (experience) of the past to improve future performance.

A predictive software model is any model built from software engineering data that can be "readily "used to make a prediction regarding some aspect of the software. Classic examples of such models are defect prediction and software cost estimation models. We will discuss how an abstraction called Relevance Relations can be used to capture relations between entities in a software system. Wed then show how one can learn a Relevance Relation from software engineering data by building a predictive software model.

Some of the results we have obtain from learning two relevance relations in the context of software change will also be presented.

Keywords: Data Mining, Predictive Software Models, Software Engineering, Software Repositories, Inductive Learning, Machine Learning, Software Maintenance

Change Classification and its Applications

Max Störzer (Universität Passau, D)

During program development, testing and code editing are interleaved activities. When tests unexpectedly fail, the changes that caused the failure are not always easy to find.

We present an analysis that automatically classify changes as Red, Yellow, or Green, indicating the likelihood that they contributed to a test's failure.

We implemented these techniques as an extension of the JUnit testing framework, and evaluated their effectiveness in two case studies. Our results indicate that change classification can effectively focus programmer attention on failure-inducing changes, improving on current manual searching/debugging techniques.

Change classification can also determine untested changes, to inform programmers that additional tests are needed. Furthermore, change classification can determine those changes that can be committed safely to a version control repository without breaking any tests, even if a developer's local workspace contains failing tests. Early adoption of edited code avoids inefficient parallel implementations of the same functionality and possible conflicts when merging independent changes later in the development process.

Keywords: Change impact analysis, change classification, debugging, tool support

Joint work of: Störzer, Maximilian; Ryder, Barbara G.; Ren, Xiaoxia; Tip, Frank

Full Paper:
http://www.research.ibm.com/people/t/tip/

Mining non-traditional artifacts for related code

Annie Ying (IBM TJ Watson Research Center - Hawthorne, USA)

Changing large software systems is hard. One of the challenges is knowing what needs to be changed in a modification task. A modification can involve changes to source code that spread across the system. To help identify the relevant parts of the code for a given task, a developer may use a tool that statically or dynamically analyzes dependencies between parts of the source. Such analyses can help a developer locate code of interest, for example, code that has call dependencies, but they cannot always identify all the code related to the change. For example, these analyses cannot typically identify dependencies between related code that involves different platform versions. Our belief is that it is valuable to augment existing static and dynamic analyses of helping programmers finding relevant code by analyzing artifacts other than code structure and program executions. One example is our previous work on analysis of development of history to find change patterns. We have shown that change patterns can reveal valuable recommendations to the programmers on two open source systems: Mozilla and Eclipse. Beside the development history data, another possible artifacts that can be analyzed is code comments, which often captures relationships between code which can not be easily captured by programming language constructs.

Full Paper:
http://www.research.ibm.com/people/a/aying/papers/2004-tse-mine-change.pdf

Don't program on Fridays

Thomas Zimmermann (Universität Saarbrücken, D)

As a software system evolves, programmers make changes that sometimes cause problems. We analyze CVS archives for *fix-inducing changes*—changes that lead to problems, indicated by fixes. We show how to automatically locate fix-inducing changes by linking a version archive (such as CVS) to a bug database (such as BUGZILLA). In a first investigation of the ECLIPSE history, it turns out that fix-inducing changes show distinct patterns with respect to the day of week they were applied. A preliminary correlation with source code shows that arrays, exceptions and anonymous classes increase the likelihood of risky changes.

Understanding Change

Filip van Rysselberghe (University of Antwerp, B)

Suppose you are a software maintainer who wants to improve the design and implementation of a deployed system, where would you actually start? To help answering this question we present a couple of techniques that allow us to study how previous developers did it. We for example, study the correlation between the number of changes of a class and its coupling to see how coupling impacts the evolution of your system. Besides this we also give a quick view on how we study how functionality is moved through the system since we feel and observe it as one of the most important change operations.