Part 2 : Scientific Information

Main applicant:	Nierstrasz, Oscar
Project title:	Analyzing, capturing and taming software change

Contents

1 Summary

Complex software systems must change in order to keep pace with changing needs and requirements. Curiously, however, modern programming languages and environments provide little support for the fact that the systems being built will inevitably change. In fact, more emphasis is placed on mechanisms to enforce consistency and to limit the effects of change than on enabling change.

This research proposal targets the following questions:

- How can we *encapsulate change* in order to better specify, manipulate and control it?
- How can we *manage the scope of change*, especially in a running system?
- How can we *assess the impact of change* in a complex system?
- How can we *exploit change* to reveal implicit trends and emergent software artifacts?

To answer these questions, we propose to (i) introduce programming language constructs to package incremental modifications to complex software systems, and use these constructs to express both low-level (syntactic) and high-level (semantic) changes, (ii) develop a scoped approach to behavioural and structural reflection in which the visibility of reflective features, and thus of changes, can be controlled at a fine level of granularity, (iii) explore techniques for tracing the impact of changes back to their source by monitoring the flow of object references in a running system, and (iv) analyze the evolution of the software and related artifacts to identify higher-level semantic entities.

Changeboxes — present-day development tools and environments do not deal with change in an explicit way. Changes can only be identified *post facto* as differences between versions. We propose to encapsulate changes as first-class entities called Changeboxes and to capture them during the development process. Changeboxes represent units of modification that can be applied to some part of a software system. They can be replayed, or selectively applied to yield different versions of software entities that may coexist in a single system.

Scoped reflection — not only source-code, but running systems need to evolve as well. Structural and behavioural reflection are well-known techniques to enable run-time change, but they can also break a running system in catastrophic ways if they are applied without discipline. We propose to develop a notion of *scoped reflection* which provides a degree of control over which reflection mechanisms are available at what time and to which clients. We intend to explore how Changeboxes can be used to dynamically activate reflective capabilities, and limit both the use and the effect (of the reflective change) to a particular scope.

Object flow analysis — even small changes can break a system in unexpected ways. Traditional debuggers help one to analyze the immediate execution context where an error is detected, but the defect responsible for the error may be distant and no longer visible in the current execution context. We propose to track the flow of object references through the running system to enable analysis of object flow, and thereby better support object-centric (rather than stack-based) navigation and analysis of a changed system. This will facilitate analysis of the impact of applying a particular Changebox to a running system.

Evolution analysis — changes are performed by developers, and they are typically driven by change requests related to the domain concepts. We plan to apply a variety of techniques to locate domain concepts implicit in the code and to analyze their evolution. Our aim is to identify the information needed to support co-evolution of the domain concepts and the implementation, and to use this knowledge to develop higher level mechanisms on top of Changeboxes. Conversely, we plan to explore various tools and metaphors to support concurrent team development using Changeboxes by exploiting the ways in which developers drive evolution.

2 Research plan

2.1 State of Research in the Field

It is well-established that complex software systems must evolve if they are to continue to be useful [LB85]. If we carry this observation to its logical conclusion, the ability to plan needs to be augmented by the ability to change [Bec00]. In this section we briefly review the most relevant state of the art in (i) encapsulating and managing change, (ii) effecting and controlling runtime change, (iii) assessing and understanding the impact of change, and (iv) analyzing historical changes with a view to program comprehension.

2.1.1 Encapsulating and managing change

During the 1970's it became increasingly apparent that keeping track of software evolution was important, at least for very pragmatic purposes such as undoing changes. Early versioning systems like the Source Code Control System (SCCS) made it possible to record the successive versions of software products [Roc75]. At the same time, text-based delta algorithms were developed [HM76] for understanding where, when and what changes appeared in the system. Since then, versioning systems have become increasingly sophisticated and have gained in acceptance if not necessarily in standardization.

Currently popular versioning systems (*e.g.*, CVS, SourceSafe, Subversion) are snapshot-based [RL05]. This means that developers download a version from the central repository, make modifications locally, and then commit the new version to the repository. Usually, as a next step, the server computes the textual differences between the two versions. The problem with this approach is that we lose the actual modifications and the order in which the developers perform them.

Darcs, a change-oriented SCCS, has recently tried to attack this problem with the goal to improve merging and to be able to detect dependencies between individual modifications [Rou05]. The latter is necessary to be able to extract a specific feature or bug fix from a set of changes, *e.g.*, to apply it to another development branch. However, the solution turned out to be limited and complex mainly because it is file-based, hence semantic information about modifications is lost.

One such piece of lost information is the performed refactorings. Refactoring is a technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [FBB+99]. Most modern development environments support at least some forms of refactoring.

Several authors have studied the possibility of recovering the refactorings performed between versions [ADP04, ZG03]. Furthermore, Diwan and Henkel developed the CatchUp! prototype to capture refactorings of the API and to replay them on the client to support evolution [HD05]. In a recent study, it was shown that most of the changes that break the API of libraries and frameworks are in fact refactorings, and as such, capturing those refactorings and replaying them on the code of the client would dramatically reduce the upgrade effort [DJ05].

The PIE System [GB84] was an early experiment to merge the features found in frame-based knowledge representation languages with Smalltalk. PIE provided the programmer with a way of having multiple concurrent versions of the system available and enabled merging these versions. However, PIE did not allow different versions of a system to be active at the same time.

Aspects [KLM+97] can be seen as describing changes on a system: code describing cross-cutting concerns is weaved at join points, changing the original definition. However, aspects focus on expressing cross-cutting concerns, rather than on changes in general. In particular, aspects are not appropriate for expressing how software artifacts evolve through a series of versions.

2.1.2 Controlling runtime change

Reflection is the ability of a system to reason and to change itself at runtime. Reflection has been the focus of research for a long time, starting with Lisp [Smi82]. It was soon applied to object-oriented systems as well [Mae87, Fer89]. Reflection is important for building systems that can adapt themselves (or can be adapted by the programmer). But this power comes at a heavy price:

the use of reflection is potentially very dangerous, as it does away with any form of encapsulation. Every client can access and change everything. The problem of building systems (*e.g.*, in a multiuser environment) that are both secure and reflective has not been solved.

Besides breaking encapsulation, another dangerous aspect of reflection (and meta programming in general) is that it provides means to the normal programer to deeply change the semantics of the system. This power should be available, but it needs to be used with care. Today's reflective systems do not control access to meta facilities, as the meta and base layers are not cleanly separated and the use of reflection cannot be controlled.

Mirrors offer a first step towards controlled reflection. In this approach objects themselves do not have any reflective capability, but reflection is provided by *mirror objects* [BU04]. Mirrors thus separate the base from the meta layer and allow one to control the access to the reflective facilities. The combination of Mirrors with scopes has not yet been explored, and it is not currently possible to scope the effect of reflection with mirrors.

Reflex provides a meta object protocol with fine-grained temporal and spatial control over behavioral reifications [TNCC03].

The idea of having objects behave differently according to some form of scope has been the focus of research in the past. Subjective programming offers the notion that objects may behave differently depending on the subject (*i.e.*, client) that uses them [HO93]. US is a system where objects have subjective semantics and can respond to messages depending on the perspective of the caller [SU96]. The perspective defines a second dimension for method lookup: methods can be defined as part of a layer that is then taken into account when the lookup is done from the perspective that corresponds to that layer.

ContextL offers a layered system where the layers are enabled depending on the current context [CH05]. The notion of context here can be more general than the single perspective seen in US. However, ContextL is not concerned with reflection.

2.1.3 Dynamic analysis

Dynamic analysis covers a number of techniques for analyzing information gathered while running the program [Bal99, Sys00]. Dynamic analysis was first used for procedural programs for applications such as debuggers [Duc99] or program analysis tools [RS93].

As object-oriented technology became more wide-spread, it was only natural that procedural analysis techniques were adapted to object-oriented languages. In this context many dynamic analysis techniques focus on only the execution trace as a sequence of message sends [KG88, LHK03, ERSS02, ZCDP05, AG05].

However such approaches do not treat the characteristics of object-oriented models explicitly. Subsequent approaches considered object creation in addition to execution traces [SM04]. For example, De Pauw *et al.* exploited visualization techniques to present instance creations events and calls between classes [PHKV93]. Lange and Yuichi built the Program Explorer to identify patterns between instance calls for framework understanding [LN95].

A natural application of dynamic analysis is debugging. When debugging, besides the stack trace, we also need to know the state of the program at a given point in time. An especially interesting approach in this area is query-based debugging, where automatic queries are executed on the trace either at the end [LHS97], or during execution [LHS99, GOA05].

The Omniscient debugger implements the concept of "back-in-time" debugging. This is an advanced debugger that not only presents the stack trace and the current state of the program, but also allows the developer to roll back time arbitrarily and inspect the state at that point in time [Lew03].

2.1.4 Evolution analysis

Diff was the first tool used for comparing the differences between two versions of a file [MES03]. Diff is able to detect addition or deletion of lines of text and it reports the position of these lines in the file, but is not useful when the analysis requires finer-grained information about what

has happened in the system (*e.g.*, in terms of classes or functions). Xing and Stroulia used a Diff-like approach to detect changes between two versions of a software system represented in XMI [XS04a, XS04b]. Zou and Godfrey used string matching and entity fingerprints to detect refactorings [ZG03]. Antoniol and Di Penta detected refactorings based on an information retrieval technique that identifies the similarity in vocabulary of terms used in the code [ADP04].

Change requests are specified in terms of domain concepts. That is why it is important to map domain concepts to code. Fischer *et al.* mined versioning systems and bug report repositories to recover features based on bug reports [FPG03, FG04]. They associated features to different parts of the system. Čubranić and Murphy bridged information from several sources to form what they call a "group memory" to recommend artifacts related to a problem [uM03].

Changes are performed by developers. Different approaches have been developed to analyze the author information from the versioning system. Ball and Eick [BE96] have represented lines of code as lines and mapped colours to represent the authors. Xiaomin Wu *et al.* visualized the change log information to provide an overview of the active places in the system as well as of the authors activity [WMSL04]. Jingwei Wu *et al.* plotted the changes of each developer on the time axis showing the number of subsystems affected [WHH04].

To identify how developers change the code, we need to know what types of changes they make. Version control systems allow descriptions of the modifications to be logged. Mockus and Votta have analyzed these descriptions to classify the changes into corrective, adaptive, inspection, perfective, and other types [MV00].

References

[ADP04] Giuliano Antoniol and Massimiliano Di Penta. An automatic approach to identify class evolution discontinuities. In *Proceedings IEEE International Workshop on Principles of Software Evolution (IWPSE 2004)*, pages 31–40, Los Alamitos CA, September 2004. IEEE Computer Society Press.

[AG05] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance 2005*. IEEE Computer Society Press, September 2005.

[Bal99] Thomas Ball. The Concept of Dynamic Analysis. In *Proceedings of ESEC/FSE '99 (7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, number 1687 in LNCS, pages 216–234, sep 1999.

[BE96] Timothy Ball and Stephen Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of OOPSLA '04, ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[CH05] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming. In *Proceedings of the Dynamic Languages Symposium 2005*, 2005.

[DJ05] Daniel Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proceedings of 21st International Conference on Software Maintenance (ICSM 2005)*, pages 389–398, September 2005.

[Duc99] Mireille Ducassé. Coca: An automated debugger for C. In *International Conference on Software Engineering*, pages 154–168, 1999.

[ERSS02] Mohammad El-Ramly, Eleni Stroulia, and Paul Sorenson. Recovering software requirements from system-user interaction traces. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 447–454, 2002.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[Fer89] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 317–326, October 1989.

[FG04] Michael Fischer and Harald Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution*, 2004.

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 90–99, Los Alamitos CA, November 2003. IEEE Computer Society Press.

[GB84] Ira P. Goldstein and Daniel G. Bobrow. A layered approach to software design. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 387–413. McGraw-Hill, New York, 1984.

[GOA05] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 385–402, New York, NY, USA, 2005. ACM Press.

[HD05] Johannes Henkel and Amer Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings International Conference on Software Engineering (ICSE 2005)*, pages 274–283, 2005.

[HM76] James Hunt and Douglas McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill NJ, 1976.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 411–428, October 1993.

[KG88] Michael F. Kleyn and Paul C. Gingrich. GraphTrace — understanding object-oriented systems using concurrently animated views. In *Proceedings OOPSLA '88 (International Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 23, pages 191–205. ACM Press, November 1988.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[LB85] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[Lew03] Bill Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, October 2003.

[LHK03] D. Licata, C.D. Harris, and S. Krishnamurthi. The feature signatures of evolving programs. *Proceedings Automated Software Engineering*, pages 281–285, 2003.

[LHS97] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *Proceedings OOPSLA '97*, ACM SIGPLAN, pages 304–317, October 1997.

[LHS99] Raimondas Lencevicius, Urs Hölzle, and Ambuj Kumar Singh. Dynamic query-based debugging. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 135–160, Lisbon, Portugal, June 1999. Springer-Verlag.

[LN95] Danny B. Lange and Yuichi Nakamura. Interactive Visualization of Design Patterns can help in Framework Understanding. In *Proceedings of OOPSLA '95 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 342–357. ACM Press, 1995.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.

[MES03] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd., 2003.

[MV00] Audris Mockus and Lawrence Votta. Identifying reasons for software change using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 120–130. IEEE Computer Society Press, 2000.

[PHKV93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93*, pages 326–337, October 1993.

[RL05] Romain Robbes and Michele Lanza. Versioning systems for evolution research. In *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, pages 155–164. IEEE Computer Society, 2005.

[Roc75] Marc Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

[Rou05] David Roundy. Darcs: Distributed version management in haskell. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4, New York, NY, USA, 2005. ACM Press.

[RS93] Herbert Ritch and Harry M. Sneed. Reverse engineering programs via dynamic analysis. In *Proceedings of WCRE '93*, pages 192–201. IEEE, May 1993.

[SM04] Maher Salah and Spiros Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004.

[Smi82] Brian Cantwell Smith. Reflection and semantics in a procedural language. Technical Report TR-272, MIT, Cambridge, MA, 1982.

[SU96] Randall B. Smith and Dave Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, 1996.

[Sys00] Tarja Systä. Understanding the behavior of Java programs. In *Proceedings WCRE '00, (International Working Conference in Reverse Engineering)*, pages 214–223. IEEE Computer Society Press, November 2000.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[uM03] Davor Čubranić and Gail Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings 25th International Conference on Software Engineering (ICSE 2003)*, pages 408–418, New York NY, 2003. ACM Press.

[WHH04] Jingwei Wu, Richard Holt, and Ahmed Hassan. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89, Los Alamitos CA, November 2004. IEEE Computer Society Press.

[WMSL04] Xiaomin Wu, Adam Murray, Margaret-Anne Storey, and Rob Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 90–99, Los Alamitos CA, November 2004. IEEE Computer Society Press.

[XS04a] Zhenchang Xing and Eleni Stroulia. Data-mining in support of detecting class co-evolution. In *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE 2004)*, pages 123–128, New York NY, 2004. ACM Press.

[XS04b] Zhenchang Xing and Eleni Stroulia. Understanding class evolution in object-oriented software. In *Proceedings 12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pages 34–43, Los Alamitos CA, 2004. IEEE Computer Society Press.

[ZCDP05] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering.* IEEE Computer Society Press, 2005.

[ZG03] Lijie Zou and Michael Godfrey. Detecting merging and splitting using origin analysis. In *Proceedings 10th Working Conference on Reverse Engineering (WCRE '03)*, pages 146–154, Los Alamitos CA, November 2003. IEEE Computer Society Press.

2.2 Research Fields

The Software Composition Group carries out research in programming languages and software engineering methods to support the construction of flexible and open software systems. In recent years the research has focussed on techniques and mechanisms to support software evolution.

In this section we review the recent SCG publications that are especially relevant to this proposal. In particular, we summarize the research we have carried out in the following areas: (i) design of programming languages and mechanisms to support incremental software evolution, (ii) reflective techniques to support runtime evolution, (iii) dynamic analysis of software systems, (iv) evolution analysis to support program comprehension.

All papers are available in electronic form from: www.iam.unibe.ch/∼scg/cgi-bin/scgpubs.cgi

2.2.1 Encapsulating and managing change

Piccola [NAK03] is a minimal language for composing applications from software components. The operational semantics of Piccola are specified in terms of a calculus of first-class environments or *namespaces* [AN05]. One of the surprising results of this research was that these first-class namespaces were highly expressive, and could be used to model a large number of software abstractions [AN00, NA05]. By dynamically composing namespaces, one could express incremental modifications to a running system, and control the visibility of these changes to an arbitrary degree.

Traits [SDNB03] represent a fine-grained approach to software composition in which classes are composed from a number of software entities (Traits) that bundle a reusable set of related methods. Traits can be used very effectively to factor out common functionality in complex class hierarchies without the need to resort to code duplication, multiple inheritance or mixins. Multiple inheritance and mixins in particular have proven to be fragile in the presence of change, so that local changes may negatively impact distant clients. Traits, by contrast, do not suffer from this fragility, because clients are explicitly in control of the way Traits are composed. In specifying the formal semantics of Traits, we were also surprised to discover that first-class environments offer a good basis to define Trait composition [DNS+06]. In essence, a Trait encapsulates a set of local changes that can be applied to a class in a controlled manner.

Classboxes [BDNW05, BDN05] offer a coarse-grained mechanism to control the visibility of class extensions in a software system. A Classbox is a kind of module which may define classes, import classes from other Classboxes, and specify *extensions* (*i.e.*, new implementations of methods) for the imported classes. The key property of Classboxes is that of *local rebinding* which guarantees that extensions are only visible to the extending Classbox, and other Classboxes that import the extended classes. As a consequence, clients who do not require these extensions will not see them. In a complex system, therefore, multiple versions of the same class may coexist.

It is also natural to explore the combination of Traits and Classboxes. In this case, a Classbox can be used to bundle a set of related Traits. A Classbox thereby represents a collaboration, and a Trait represents a role within that collaboration [BD05b, MBCD05].

2.2.2 Controlling runtime change

Smalltalk provides a number of reflective features [Duc99], most of these being concerned with structure. For example, we can add or remove methods. Reflective facilities for changing behavior are only supported in a rudimentary way. There is no *meta object protocol* to allow for fine-grained control of behavior. For example, we cannot re-define what a message send is, and we cannot hook into variable access easily.

Over the past year, SCG has implemented ByteSurgeon [DDT06], a framework for transforming Smalltalk bytecode at runtime. Being able to annotate the bytecode of methods is useful for different purposes. In particular, we have explored two directions: (i) changing bytecode for providing behavioral reflection, and (ii) bytecode insertion for getting data about program runtime (execution tracing).

On top of ByteSurgeon, we have implemented Geppetto [Röt06], a dynamic runtime meta object protocol for behavioral reflection. Geppetto allows for a very fine-grained control of behavioral reflection. For example, we can reify message sends just for a single object.

Classboxes appear at a first glance to be static, but they fit nicely into a dynamic context. In this case, Classboxes can be dynamically slotted in or out, thus enabling or disabling sets of extensions to provide dynamic aspects [BD05a].

We have also explored how a dynamic change of context can determine the installation of aspects in a system [TGDB06]. The supported notion of context is very general: it can be anything that we can programmatically determine to be true or false. In this way, the context does not need to be limited to programming language entities (like the static scope of a package, or the dynamic scope of an execution), but can include the state of the domain model or even the state of the environment that the system is embedded in.

2.2.3 Dynamic analysis

Our early work in this area dealt only with analyzing the execution traces for reverse engineering purposes. We have built several visualizations [DLB04, GLW05, Wys05], used pattern matching to identify collaborations between classes [RD02], mapped features to code [GD05], and used the execution traces to order broken unit tests [GLNW04].

Recently, we have developed an approach to encode and execute tests directly on the execution traces [DGW06]. For this, besides the execution trace, we also stored the state of the objects before and after the message sends, much in the same way the Omniscient debugger does.

A project currently under development is a back-in-time debugger, an implementation of the Omniscient debugger. It uses ByteSurgeon to annotate the system to get a complete history of the program run, then uses this trace information for an enhanced debugger. This debugger allows one not only to navigate the current stack trace but the complete execution trace and it reconstructs the state of the program at any point in time.

In parallel we are working on a run-time model that complements the back-in-time debugger with object flow information. To trace the path of an object, each object reference is explicitly represented as a first-class entity named *object alias*. The trace information based on aliases allows us to precisely detect the path an object follows, *i.e.*, the sequence of passing it between objects and of storing it in instance variables.

2.2.4 Evolution analysis

In earlier work, we have applied simple software metrics to detect software refactorings between different versions of a system [DDN00]. We have also defined the Evolution Matrix as a high-level view to depict the evolution of classes of a system over time [LD02].

More recently we have shown that to analyze software evolution we need to model it as a first class entity [Gîr05]. We have developed the Hismo meta-model that defines history as a sequence of versions to encapsulate the evolution [GD06].

We have shown the usefulness of the approach by exercising the meta-model for several evolution analyses. We have defined Yesterday's Weather to measure the relevance of starting the analysis of a new system from the latest changed parts [GDL04]. We have presented a visualization to show how relationships between structural entities can also be manipulated from an historical perspective [GLD05]. In another application, we have shown how due to our abstraction, evolution information can easily be combined with structural information for analysis purposes [RDGM04].

We have not only analyzed the information statically extracted from the source code, but we have also applied our approach from other sources. We analyzed how authors have changed the system to gain insights into the team's behavior [GKSD05]. We also extended our feature analysis with a time dimension to consider the evolution of a system from a feature perspective. We detected changes in the functional roles of a class or method to identify changes inconsistent with the purpose of a software entity or misplaced code [GDG05].

We have also explored the use of information retrieval techniques to analyze the linguistic information found in the source code recover domain information [KDG05]. We have also explored the use of the same technique to deal with dynamic information to identify similarities between features [KGG05]. In this proposal we intend to apply these techniques to multiple versions of a software system to gain insight into the evolution of domain concepts over time.

Much of this work has been carried out in the context of the Moose reengineering environment [DGLD05, DT03], developed within the SCG since 1997. Moose is a generic infrastructure for reengineering [NDG05], and on top of it several tools have been implemented: CodeCrawler is a general purpose visualization tool [LD05], ConAn is a concept analysis tool [Aré05], Chronia is a tool for analyzing CVS repositories [See06], Hapax is a tool to analyze the linguistic information from the source code [KDG05], TraceScraper is a tool for dynamic analysis [GD05], and Van is a generic tool for history analysis [Gîr05].

References

[AN00] Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages*, volume 1897 of *LNCS*, pages 77–89, Zürich, Switzerland, September 2000. Springer-Verlag.

[AN05] Franz Achermann and Oscar Nierstrasz. A calculus for reasoning about software components. *Theoretical Computer Science*, 331(2-3):367–396, 2005.

[Aré05] Gabriela Arévalo. *High Level Views in Object-Oriented Systems using Formal Concept Analysis*. PhD thesis, University of Berne, Berne, January 2005.

[BD05a] Alexandre Bergel and Stéphane Ducasse. Scoped and dynamic aspects with Classboxes. *RSTI – L'Objet (programmation par aspects)*, 11(3):53–68, 2005.

[BD05b] Alexandre Bergel and Stéphane Ducasse. Supporting unanticipated changes with Traits and Classboxes. In *Proceedings of Net.ObjectDays (NODE'05)*, pages 61–75, Erfurt, Germany, September 2005.

[BDN05] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189, New York, NY, USA, 2005. ACM Press.

[BDNW05] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4):107–126, May 2005.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 166–178, New York NY, 2000. ACM Press.

[DDT06] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, July 2006.

[DGLD05] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[DGW06] Stéphane Ducasse, Tudor Gîrba, and Roel Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*. IEEE Computer Society Press, 2006. To appear.

[DLB04] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of CSMR 2004 (Conference on Software Maintenance and Reengineering)*, pages 309–318, 2004.

[DNS+06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *Transactions on Programming Languages and Systems*, March 2006. To appear.

[DT03] Stéphane Ducasse and Sander Tichelaar. Dimensions of reengineering environment infrastructures. *International Journal on Software Maintenance: Research and Practice*, 15(5):345–373, October 2003.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[Gîr05] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, November 2005.

[GD05] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*, pages 314–323. IEEE Computer Society Press, 2005.

[GD06] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *International Journal on Software Maintenance: Research and Practice (JSME)*, 2006. To appear.

[GDG05] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance)*, pages 347–356. IEEE Computer Society Press, September 2005.

[GDL04] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, 2004. IEEE Computer Society Press.

[GKSD05] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE)*, pages 113–122. IEEE Computer Society Press, 2005.

[GLD05] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[GLNW04] Markus Gaelli, Michele Lanza, Oscar Nierstrasz, and Roel Wuyts. Ordering broken unit tests for focused debugging. In *20th International Conference on Software Maintenance (ICSM 2004)*, pages 114–123, 2004.

[GLW05] Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing feature interaction in 3-d. In *Proceedings of Vissoft 2005 (3th IEEE International Workshop on Visualizing Software for Understanding)*, September 2005.

[KDG05] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of Working Conference on Reverse Engineering (WCRE 2005)*, pages 113–122, Los Alamitos CA, November 2005. IEEE Computer Society Press.

[KGG05] Adrian Kuhn, Orla Greevy, and Tudor Gîrba. Applying semantic analysis to feature execution traces. In *Proceedings of Workshop on Program Comprehension through Dynamic Analysis (PCODA 2005)*, pages 48–53, November 2005.

[LD02] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of Langages et Modèles à Objets (LMO 2002)*, pages 135–149, Paris, 2002. Lavoisier.

[LD05] Michele Lanza and Stéphane Ducasse. Codecrawler–an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74–94. Franco Angeli, Milano, 2005.

[MBCD05] Florian Minjat, Alexandre Bergel, Pierre Cointe, and Stéphane Ducasse. Mise en symbiose des traits et des classboxes : Application à l'expression des collaborations. In *Proceedings of LMO 2005*, volume 11, pages 33–46, Bern, Switzerland, 2005.

[NA05] Oscar Nierstrasz and Franz Achermann. Separating concerns with first-class namespaces. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit, editors, *Aspect-Oriented Software Development*, pages 243–259. Addison-Wesley, 2005.

[NAK03] Oscar Nierstrasz, Franz Achermann, and Stefan Kneubühl. A guide to JPiccola. Technical Report IAM-03-003, Institut für Informatik, Universität Bern, Switzerland, June 2003.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[RD02] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of ICSM '2002 (International Conference on Software Maintenance)*, October 2002.

[RDGM04] Daniel Raţiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Proceedings Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 223–232, Los Alamitos CA, 2004. IEEE Computer Society.

[Röt06] David Röthlisberger. Geppetto: Enhancing Smalltalk's reflective capabilities with unanticipated reflection. Masters Thesis, University of Bern, January 2006.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[See06] Mauricio Seeberger. How developers drive software evolution. Diploma Thesis, University of Bern, January 2006.

[TGDB06] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, LNCS, Vienna, Austria, March 2006. To appear.

[Wys05] Christoph Wysseier. Interactive 3-D visualization of feature-traces. MSc. thesis, University of Berne, Switzerland, November 2005.

2.3 Detailed Research Plan

The proposed research explores ways to enable the software developer to manage software change more effectively than is currently possible. We propose four closely related tracks to achieve this end:

1. develop a language construct, called CHANGEBOXES, to encapsulate change,
2. develop an approach, called *scoped reflection*, to manage and delimit the scope of change,
3. develop a dynamic analysis technique, called *object flow analysis*, to assess the impact of changes, and
4. exploit the history of change, by means of *evolution analysis*, to expose domain concepts implicit in the code.

We will outline the goals of each of these tracks and describe the steps we envisage to achieve these goals.

2.3.1 Changeboxes

There is currently a gap between programming languages and development environments with respect to supporting evolution. Although integrated development environments such as Eclipse help in integrating different aspects of the development process (*i.e.*, revision control, code refactoring) the current solutions are far from ideal because they are not based on a common infrastructure. They do not support well:

- Capturing information about changes. Most revision control systems are only snapshot based and do not provide any semantical information about the changes into account.
- Concurrent development in a team. Short synchronization cycles and switching between different versions of a software are hampered by a tedious, file and snapshot based, update-merge-commit procedure.
- Incremental adaption of a system to cope with a specific change. Large refactorings can affect many parts of a system and therefore may need a considerable amount of work to be pushed through. While unfinished, the system is not in a consistent shape and cannot be integrated with other work, hindering the evolution or even preventing developers from applying the refactoring.

We plan to investigate means to better support evolution of software systems by explicitly modeling changes at the level of the development environment and programming language. The specific question we want to answer is: What are the appropriate abstractions to explicitly model the evolution of a software system?

Over the past few years, SCG has developed Classboxes, a scoping mechanism to support unanticipated changes. Classboxes are coarse-grained components that enable developers to locally extend software in a uniform way without impacting other users of that software. As Classboxes have been designed to enable scoping of class extensions, we plan to apply the same principle not only to scope structural changes but also to scope changes over time in an uniform way.

As a result of our research into software evolution, we have concluded that evolution should be modeled as a first class entity. We propose to explore a model for fine-grained changes as first-class entities, and we propose to use this mechanism, called CHANGEBOXES, to express both time-based and structural scoping mechanisms. CHANGEBOXES will allow several versions of the same software artifact to coexist and to be runnable at the same time.

We plan to model changes as ubiquitous events in the evolution of a system, that is, each change brought by a developer will be immediately accessible by any other team member. The difference to changes having global effect is that each change is performed within a specific scope, and the environment allows the developer to switch between these scopes as appropriate. We believe this will achieve a more transparent development process, and will provide a rich model for integrating tools and performing analysis for reverse and re-engineering.

We will start by implementing a Smalltalk prototype of CHANGEBOXES which will express the possibility of add or remove methods. The first prototype will be based on the Classbox implementation. The next step will be to enable the addition or removal of classes, instance variables and changes in inheritance relationships.

A key validation of CHANGEBOXES will be to express not only low-level modifications to specific versions, but to also express higher-level refactorings which can eventually be applied to different versions of different artifacts. A more important long-term goal is to be able to use CHANGEBOXES as a mechanism to push changes through a system in a controlled way, using scoped reflection.

As we are actively involved in developing Squeak Smalltalk prototypes, and as CHANGEBOXES will be implemented in Squeak, we plan to validate the CHANGEBOX model by using the environment in our day-to-day implementation process. After an initial period, we will release the environment for the Squeak community and collect experience reports.

2.3.2 Scoped Reflection

Software systems are typically changed by modifying the source code and recompiling the whole system. This rather old-fashioned model of software development suffers from two outmoded assumptions that are increasingly in conflict with the reality of modern software applications. First is the assumption that the system can be stopped, recompiled and restarted. Many of today's software systems must be up virtually all the time. Second is the assumption that the universe is consistent. Software systems today must cope with the fact that libraries, components and peer systems may be based on incompatible versions of interfaces, protocols and standards. We therefore need mechanisms to enable (i) runtime changes, and (ii) scoped visibility of changes.

Reflection is the ability of a system to change itself at runtime. Smalltalk is a reflective system in which the structure of the system is described by classes and can be changed anytime. Aside from these structural reflective capabilities, however, there is no *meta object protocol* to support fine-grained control of behavior. We have previously implemented Geppetto, a dynamic runtime meta object protocol for behavioral reflection.

We plan to extend the behavioral reflection framework of Geppetto with the notion of scoped behavioral reflection. This means we want not only to control *what* (spatial) and *when* (temporal) to reflect, but in addition to control the reifications based on the control flow. For example, we want to specify that reifications are active only if the control flow originates in a specified sub-system.

Based on the scoped behavioral reflection framework, we will implement a second version of the back-in-time debugger. This version will not use bytecode insertion directly, but rather the intermediate layer of the behavioral reflection framework. This will allow for scoping the execution tracing towards the client that we want to debug, and thus it will be a validation for scoped behavioral reflection.

As the next step we plan to explore a reflective system that not only scopes behavioral reflection, but in addition provides scoping capabilities for structural reflection. In this track we plan to leverage the work on CHANGEBOXES. We plan to use the behavioural reflection framework as a mechanism to selectively scope the application of CHANGEBOXES dependent on a given context. Conversely, we also plan to use CHANGEBOXES as a mechanism for controlling the availability of the reflective mechanisms themselves. Depending on the current context, then, reflective capabilities may be available, or safely locked away.

In the long term, we plan to explore how the combination of scoped reflection and CHANGEBOXES can enable a more general model of context-oriented programming, in which the structure and behaviour of software artifacts may change dynamically but in a controlled way depending on the dynamic context.

2.3.3 Object Flow Analysis

Changes can introduce problems. To fix a problem the developer has to understand the connection between the change that introduced the problem, the missing adaption that is the actual source

of the problem and the location where the problem manifests itself.

Dynamic analysis provides exact information about the control flow of the program and is widely used to support the understanding of the execution traces (*i.e.*, the sequence of methods calls) of a program. Execution traces have their origin in procedural programming and were later mapped to object-orientation. However, most approaches only introduced minor adaptations to cope with the special characteristics of the object-oriented model, *e.g.*, they take the sender and the receiver into account or they track instantiation and destruction events of objects. While control flow information in such traces illustrates the sequence and nesting of method invocations, it falls short of showing the interaction between objects. The cause of this shortcoming is that execution traces observe the stepwise program behavior at a lower level of abstraction rather than at the level of the objects which are the language's core elements. We are forced to focus on the run-time stack trace rather than the object interactions.

We plan to investigate a run-time model that complements execution traces with object flow information. We are currently working on a prototype for run-time analysis which introduces the concept of *object alias* to represent explicitly the references to an object.

In comparison to the Omniscient debugger which traces the execution of the program at a low level, our prototype will additionally provide information about the interaction between objects. For example, we will provide the path an object was passed along, or different locations from which an object was referenced or modified.

Based on this model we plan to build a high-level *object-centric* debugger. By tracing the flow of objects the debugger can provide shortcuts between the chronological sequence of method executions and thus help to bridge the cause/effect gap of errors. For example, when the program breaks because of an incorrectly assigned instance variable, the location where the bug occurs, *i.e.*, a message that is not understood by the instance variable, is very likely to be in a distant branch of the execution trace. Based on the object's flow trace, the developer can navigate in the execution history along the object flow path to find the location where the field was incorrectly set.

Changeboxes will allow for several versions to coexist in the same system. To identify the impact of changes from one version to another, we will implement analyses to compare the dynamic information of the execution of the two versions. In this way we can get closer to identifying the cause of a bug.

2.3.4 Evolution Analysis

The evolution of software systems is driven by changes at the level of the domain (*e.g.*, features). We plan to use several techniques to locate domain concepts from the code and then to analyze how the domain concepts have evolved in the code as opposed to how they are linked conceptually.

We have already started to analyze how the changes in the code relate to the features implemented, by using dynamic analysis. We plan to further investigate this path to recover domain interpretations for the changes in the code, that is, to identify the reasons why the code was changed.

We will use our information retrieval technique over several versions of a system to detect when the concepts appear in the system, how they spread and perhaps how they die. We will use the Hapax tool for semantic analysis and combine it with the Van evolution analysis tool.

Semantic Clustering is based only on the vocabulary found in the code. We also plan to complement it and use an ontology to detect how the links between concepts in the ontology map to the implementation relationships.

Our main goal is to identify the kind of semantics useful for being encapsulated in Changeboxes. A Changebox encapsulates a change. Current versioning systems allow for a textual description to be attached to every new version, but this free-form is difficult to analyze automatically. We plan to allow the developer to relate his changes with the domain concepts using Changeboxes. On the one hand, we want to identify what mechanisms we need for modeling the domain knowledge and relate them to the code changes. On the other hand, we plan to use our automatic analyses to provide hints for the relevant concepts for a change.

Software evolution is driven by developers. That is why we need to get more insights into how developers work together to better understand how to support their activities. We have already started several experiments to detect patterns of how developers change the system, by analyzing CVS repositories. A particular focus is to combine the developer analysis with the concept location analysis to assign domain concepts to developers. In this way, we link the reasons for change with the different behaviors. By analyzing the developers patterns, we expect to gather requirements for building tools on top of the Changeboxes to facilitate the distributed team development.

2.4 Timetable

We plan that Mr. Lukas Renggli will start his PhD work in October on Changeboxes. Mr. Marcus Denker started his PhD in May of 2004, and will continue to work on the topic of Scoped Reflection. Mr. Adrian Lienhard started his PhD research in January 2005, and will be working on Object Flow Analysis. Mr. Adrian Kuhn has recently completed his Masters' thesis, and will start his PhD this Spring on Evolution Analysis.

We expect to obtain the following results over the two years of the project. We expect a rough correspondence between bullet items below and publishable units.

Year 1	
Changeboxes	– implement a first version of Changeboxes (in Smalltalk) encapsulating addition and deletion of methods – formalize Changeboxes in terms of first-class namespaces – extend Changeboxes to addition and deletion of instance variables and classes
Scoped Reflection	– use ByteSurgeon and Geppetto to implement scoped behavioural reflection – implement omniscient debugging using scoped reflection
Object Flow Analysis	– implement first object flow analysis tool by extending the existing prototype of Omniscient debugger – develop the concept object-centric debugging views to exploit object flow analysis
Evolution Analysis	– apply information retrieval techniques to multiple versions to identify relevant concepts emerging over time – mine version repositories to detect patterns of how developers change systems
Year 2	
Changeboxes	– use Changeboxes to express high-level refactorings – use scoped reflection to propagate changes through a running system – experiment with Changeboxes on real implementation case studies
Scoped Reflection	– implement scoped structural reflection – combine scoped reflection with Changeboxes
Object Flow Analysis	– extend object flow analysis to compare two different versions for detecting the impact of changes – integrate the impact analysis with Changeboxes
Evolution Analysis	– develop ontologies to link concepts emerging over different versions, and correlate to developer vocabularies – identify the meta-model to capture domain concepts and implement a mechanism for representing them in Changeboxes

2.5 Significance of the Research

As the research we propose is fundamental in nature, the key venues for disseminating the results are the international, peer-reviewed conferences and journals in which we have consistently published our results.

Proof that these venues can be highly effective is given by the fact that our previous work on *Traits* has already had an impact on the design of mainstream programming languages such as Perl and Smalltalk (*cf.* our previous and ongoing SNF projects 2000-067855.02 "Tools and Techniques for Decomposing and Composing Software", and 200020-105091/1 "A Unified Approach to Composition and Extensibility"). Traits have also been incorporated into newer languages such as Fortress (Sun Microsystems) and Scala (EPFL). We have also recently completed a pilot project with Microsoft Research to investigate the incorporation of Traits into C#.

Our involvement in the development of the Squeak Smalltalk platform also provides us with an excellent opportunity to influence the evolution of the Smalltalk language and environment. Besides an implementation of Traits, we have designed a change notification framework that provides a service to notify interested objects when the system changes (*e.g.*, additions of classes or methods). This has been integrated into the Squeak release and is used throughout the system to keep all the tools in sync with the changes the developers perform. We have also helped to design and integrate a method annotation framework for Squeak 3.9, allowing programmers to add meta data to methods easily.

Part of the proposed research is more concerned with analysis of software artifacts rather than with programming technology. In addition to publishing results in established venues, we also actively seek collaborations with industry, in particular to apply our analysis techniques to live case studies consisting of many versions of an evolving software product. We are currently completing a pilot project with Harman/Becker Automotive Systems (Germany), and we are initiating a pilot project with PostFinance (Switzerland).