Part 2 : Scientific Information

Main applicant:	Nierstrasz, Oscar
Project title:	Agile Software Assessment

Contents

1 Summary of the research plan

A significant portion of software development effort is devoted to reading and understanding code. Unfortunately, mainstream integrated development environments (IDEs) focus on low-level programming tasks rather than on supporting program comprehension and decision-making during software evolution. Analysis tools, on the other hand, usually have a narrow scope of applicability.

This project aims *to enable software developers to quickly and effectively analyze complex software systems with the help of tools to rapidly construct, query and manipulate software models.* We refer to this goal as *agile software assessment*, since developers are under constant pressure to assess the state of the system at hand in a timely fashion in order to carry out development and evolution tasks. The expected long term benefits of this research are improved developer efficiency, enhanced tool support during software development, and better quality software.

To this end, we propose four related research tracks: (i) Meta-Tooling will enable developers to rapidly develop custom tools to support decision-making, (ii) Agile Modeling will simplify the task of constructing and refining software models from source code and other data sources, (iii) Large-Scale Software Analysis will allow developers to process and exploit the large amount of additional data related to a project, and (iv) Architectural Monitoring will help developers track the evolution of architectural constraints in complex software.

— *Meta-Tooling.* Developers ask detailed and domain-specific questions about the software systems they are developing and maintaining. Specialized tools are needed to effectively answer these questions. Most IDEs do not provide suitable functionality to answer detailed questions about the design and implementation of software, as they focus on programming rather than software modeling. We plan to carry out empirical studies to better understand what questions developers ask about code, and what tools and techniques are most effective at supporting program understanding and software assessment. *Meta tools* are tools for building tools. We propose to construct an experimental *meta-tooling environment* to enable rapid composition of customized software assessment tools to support effective decision-making in the development process.

— *Agile Modeling.* A key bottleneck to effective software assessment is the rapid construction of appropriate software models from program source code and the associated data sources. We propose to develop techniques to construct such models with the help of reusable and composable parsers targeting common programming language classes, user-guided parser refinement using island grammars, and semi-automated inference of structural features of program and data sources by analyzing lexical features such as indentation and textual content.

— *Large-Scale Software Analysis.* Complex software systems generally exist within an even larger software ecosystem consisting of older versions of the systems, variants, and other client applications of the system or its parts. Being able to query and mine this large resource of information can be very helpful for developers to make informed decisions concerning the project at hand. We propose to develop an infrastructure to model and query large software data sets, including historical information and variants. We will seek to adapt and exploit *big data* analysis techniques to the particularities of large source code repositories and software ecosystems.

— *Architectural Monitoring.* The *architecture* of a software system consists of the design constraints that guarantee non-functional properties, such as ease of evolution, good run-time performance, and rapid build times. Unfortunately architecture is rarely explicit in code, hence it must be recovered and tracked, sometimes at great cost in developer time. We propose to carry out empirical studies to identify and classify architectural constraints that arise in practice in software systems and let these studies influence the way in which we design mechanisms for the specification of constraints. We propose to develop techniques to monitor architectural evolution and integrate monitoring into the development process and environment; this promises to improve system stability, quality, and robustness.

2 Research plan

2.1 Current state of research in the field

It is well-acknowledged that developers often spend as much time understanding code as they do writing new code [LVD06]. As established by Lehmann and Belady [LB85], real-world software systems become more complex over time as they are adapted to fulfill new requirements, unless effort is invested to simplify their design. Decision-making during development and evolution consequently becomes more difficult as the system at hand becomes more complex. The goal of this project is to aid software developers in understanding and assessing complex software systems in a timely fashion while carrying out development and evolution tasks.

We review related work in building dedicated software analysis tools, in constructing software models from source code and other data, in applying "big data analysis" techniques to large sources of software information, and in specifying software architectures and tracking their evolution.

Meta-Tooling In this section we motivate the need for *meta-tooling*, that is, tools to help build software assessment tools, as well as research into exactly what features such meta-tools should target.

Although developers spend a great deal of time reading and trying to understand complex software systems, the development environments tend to focus on low-level programming tasks, not program comprehension or decision-making. While developers clearly would benefit from dedicated software assessment tools, it is not clear what tools are needed nor how they can best be put to use within an integrated environment. Whereas controlled empirical studies are often carried out in an attempt to assess the impact of a new technique or tool, recent studies have focused more on trying to better understand current practice in the field. LaToza *et al.*, for example, carried out surveys and interviews that show that developers are often forced to recover implicit knowledge by exploring code and discussing with teammates [LVD06]. Sillito *et al.* have carried out qualitative studies to determine what kinds of questions programmers ask while carrying out typical software evolution tasks [SMDV06]. Fritz *et al.* have developed so-called "degree-of-interest" models based on empirical studies to model what programmers know about a code base [FMH07].

Countless tools have been developed to aid developers in evaluating, understanding and analyzing software systems. Entire conferences, such as ASE (Automated Software Engineering) and ICPC (International Conference on Program Comprehension) are devoted to the topic. Unfortunately, most of the tools and techniques proposed at these venues have a narrow scope of applicability and are not integrated into IDEs.

Most IDEs do not offer much help in integrating new tools. For example, Eclipse, a widely-used open-source IDE [BSM$^+$03], offers a general modeling framework and a means to integrate new tools as "plug-ins". However developing an Eclipse plug-in is non-trivial, as is exploiting Eclipse models from such a plug-in. What is missing is an easy way to develop customized software assessment tools to query, manipulate and present models of complex software systems managed by the IDE.

A meta-tooling environment would offer a way to quickly develop software assessment tools within the IDE. Various techniques have been developed over the years to facilitate the composition of applications from reusable parts. Component-oriented programming [Szy02] focuses on abstractions at a higher level than objects that can be composed using standard interfaces. Domain Specific Languages (DSLs) [Fow10] are high-level languages for configuring functionality related to a specific domain. Very often DSLs simply reflect the compositional interface of an underlying component framework. Model-driven engineering (MDE) has emerged as a way to rapidly generate and adapt applications for multiple platforms by transforming specifications at the model level to concrete implementations [Sch06]. Naked Objects refers to an approach in which domain objects encapsulate business logic and user interfaces are fully generated from the composition of the domain objects [PM02]. Instead of writing meta-software to produce an application, naked objects are directly composed to form a running application.

Agile Modeling With "Agile Modeling" we refer to the challenge of automatically constructing software models from source code and other data sources. Since building a full parser by hand for a general-purpose programming language is time-consuming and expensive, there is a need for techniques to streamline this process. Furthermore, parsing technology generally focuses on constructing an exact representation of program source code for compilation or interpretation purposes, rather than on transforming multiple data source towards models suitable for analysis and decision-making.

There is a long history of work on "grammar induction" or "grammar inference". For example, the International Colloquium on Grammatical Inference and Applications has been running regularly since 1993. The focus of this work has been, however, on learning regular grammars and deterministic finite automata, not on software modeling [dlH05].

Lämmel and Verhoef have developed a practical approach to semi-automatic grammar recovery in which they scavenge full-blown grammars for existing languages from a variety of sources, including documentation [LV01]. Although they report good success, it still took them several weeks to reconstruct a full grammar for a language of the complexity of COBOL. Kraft *et al.* take a similar approach in recovering grammars from the parse trees generated by the parser itself [KDM09].

In software modeling it is typically not necessary to completely parse all parts of the source code in order to perform many kinds of analyses. Techniques like fuzzy parsing [Kop97] and island grammars [Moo01] have been used to develop robust parsers that analyze selected portions of source code (*i.e.*, the "islands", which are of interest) and ignore others (*i.e.*, "sea", which is ignored).

So-called "language workbenches", like *Stratego* [BKVV08] and Xtext [EV06], provide means to rapidly develop parsers. PEGs (parsing expression grammars) [For04] offer means to construct and compose parsers, in sharp contrast to traditional parsers generated from grammar descriptions which frequently produce conflicts. Scannerless parsers [Vis97] furthermore eliminate the need for a separate lexical scanning phase in the parser. This is especially interesting for developing parsers for mixed languages, which do not share a common set of lexical tokens.

Finally, other sources of information can be gleaned from source code without making use of a full parser. Hindle *et al.*, for example, have shown that indentation can be used as a reliable proxy for structure and thus for complexity in source code [HGH08].

What is missing is a unifying framework to enable rapid and iterative construction of software models suitable for analysis from multiple program and data sources.

Large-Scale Software Analysis Recently there has been a surge of interest in pushing software analysis beyond the level of individual systems, one of the reasons being the new availability of data. Indeed software is entering the age of big data, which is characterized by increasing volume (amount of software), velocity (speed of software generation), and variety (range of data sources).

There is already a rich tradition of research in mining software repositories, but this usually is concerned with mining versioning repositories of individual systems to detect problems with the source code [SZZ05, WH05], mining a single developer's interactions with his IDE to improve the IDE [KM05, RL08], or to predict bugs [DLLR11].

Research has recently shifted towards mining larger corpora of projects. One direction is analyzing related projects, another is the large scale analysis of unrelated projects.

In their work on Codebook, Begel *et al.* conducted a study with Microsoft engineers to discover the needs of the developers working in a large corporate ecosystem. They discovered that some of the important needs are related to awareness and impact [BKZ10]. Their solution, Codebook is a proprietary approach to modeling the variety of artefacts that are associated with the source code in an ecosystem. Mileva *et al.* [MDBZ09] studied the evolution of libraries in the apache ecosystem and discovered that the *wisdom of the crowds* can be helpful when deciding which version of a library to use.

Ossher *et al.* started from another developer need that commonly arises in the open-source world, which is to build a newly downloaded artifact. Accordingly, they studied whether they could cross-reference a project's missing types with a repository of candidate artifacts [OBL10].

Multiple system analysis need not consider only projects that are part of an ecosystem. Empirical studies have been carried out to analyze the way programming languages are used in practice as well as to identify the actual needs of developers. Callau *et al.* [CRTR11] studied the usage of reflective programming features in a large Smalltalk codebase in order to assess how much these features are actually being used in practice.

One other application of analysis at a large scale is code search, a direction which has been approached by both industry and academia. The first academic search engines were Sourcerer [BNL$^+$06] and S6 [Rei09], and a generation of tools that used search engines as a backbone followed. Thummalapenta and Xie [TX07] developed PARSEWeb, an approach that interacts with a code search engine to gather relevant samples that would allow them to help developers find reusable components. CodeConjurer also taps into the results of code search to deliver reuse recommendations [HJA08]. Code Genie searches for code that is specified as a series of unit tests [LLBO07]. In order to detect clones, Koschke indexes the source code using suffix trees [Kos12].

The challenge when realizing the infrastructure for such data is to balance the trade-offs between providing a simple representation of the data and optimizing for a particular task. Mockus reported on his experience in amassing a very large index of version control systems and the associated challenges: the data from the SourceForge CVS occupies more than 1TB, extracting large amounts of files from the Mercurial database took more than one month, using standard database techniques that would work for the code in a large corporation would lead to processing times of many months, *etc.* [Moc09]. Therefore, in the realm of big software data, an intelligent infrastructure can bring large savings in processing and retrieval time.

Architectural Monitoring As the environment of a successful software system evolves, the system too has to evolve, and typically, so must its architecture. To keep track of the architectural evolution, one needs a way to specify the architecture to enable the monitoring of its evolution.

Software architecture has emerged as an important branch of software engineering in the past twenty years. Shaw and Garlan [SG96] have characterized software architectural styles in terms of coarse-level system *components*, high-level *connectors* that mediate the interaction between the components, and *constraints* that govern the connections. The goal of a software architecture is to support certain *analyses* and to guarantee certain desirable *properties* (such as performance, robustness and maintainability).

One of the difficulties in reasoning about software architecture is that architecture is largely implicit in the source code. Aside from coarse-grain package structure and naming conventions, very few indicators exist in source code that reveal the architecture. This fact has led to two separate research activities, namely work on *specifying* software architectures, and work on *recovering* architecture. A third activity, *monitoring* the evolution of software architecture is in its infancy.

Shaw and Garlan [SG96] provide a detailed survey of so-called "architectural description languages" (ADLs) which are intended to augment the source code of a software system with a specification of its architecture. Classical ADLs such as Rapide and Wright defined architecture in terms of components and connectors [LKA+95, AG96]. More recent work continues in the same tradition: Dashofy *et al.* [DHT05] focused on developing an XML-based ADL that is modular and composable, but at the same time remark that ADLs are not having a significant impact on software engineering practice. In his work on ArchJava [Ald08], Aldrich extends the Java programming language with component classes, which describe objects that are part of an architectural description. Also in the case of ArchJava the architecture is described in terms of components and connectors.

Many techniques have been developed to recover software architectures. Best-known perhaps is Murphy's top-down *reflexion modeling* approach [MN97], in which a software reverse engineer hypothesizes an architecture, and refines the hypothesis (or refutes it) iteratively to arrive at an architectural description. Another technique proposed by Pinzger visualized both structural and evolutionary characteristics of software systems [Pin05]. Various other tools have been developed both in research and industry to detect violations of architectural constraints. Sotograph, for example, attempts to identify violations of layered architectures [BKL04]. IntensiVE is an environment that monitors internal quality of software systems with the help of *intensional views* [MK06], which specify architectural and design constraints using a dedicated logic programming language.

All existing recovery techniques are currently designed as throw-away approaches. The analyst recovers the architecture, and then the process is over. There is a great opportunity for moving towards an *architecture monitoring* model drawing on the extensive experience of techniques designed for recovery. One such example is the work of Knodel [KMR08] who proposes *constructive compliance checking*, an approach to providing live feedback to developers when they violate architectural specifications which, authors show in a controlled experiment with students, improved the quality of the resulting system [Kno11].

References

[AG96] Robert Allen and David Garlan. The Wright architectural specification language. CMU-CS-96-TB, School of Computer Science, Carnegie Mellon University, Pittsburgh, September 1996.

[Ald08] Jonathan Aldrich. Using types to enforce architectural structure. In *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, WICSA '08, pages 211–220, Washington, DC, USA, 2008. IEEE Computer Society.

[BKL04] Walter Bischofberger, Jan Kühl, and Silvio Löffler. Sotograph – a pragmatic approach to source code architecture conformance checking. In *Software Architecture*, volume 3047 of *LNCS*, pages 1–9. Springer-Verlag, 2004.

[BKVV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008.

[BKZ10] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[BNL$^+$06] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682, New York, NY, USA, 2006. ACM.

[BSM$^+$03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[CRTR11] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How developers use the dynamic features of programming languages: The case of Smalltalk. In *Proceedings of the 8th working conference on Mining software repositories (MSR 2011)*, pages 23–32, New York, NY, USA, 2011. IEEE Computer Society.

[DHT05] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Softw. Eng. Methodol.*, 14(2):199–245, April 2005.

[dlH05] Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

[DLLR11] Marco D'Ambros, Michele Lanza, Mircea Lungu, and Romain Robbes. On porting software visualization tools to the web. *In Journal on Software Tools for Technology Transfer*, 13:181 – 200, 2011.

[EV06] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, September 2006.

[FMH07] Thomas Fritz, Gail C. Murphy, and Emily Hill. Does a programmer's activity indicate knowledge of code? In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 341–350, New York, NY, USA, 2007. ACM.

[For04] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, September 2010.

[HGH08] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Reading beside the lines: Indentation as a proxy for complexity metrics. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 133–142, Washington, DC, USA, 2008. IEEE Computer Society.

[HJA08] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *Software, IEEE*, 25(5):45–52, 2008.

[KDM09] N.A. Kraft, E.B. Duffy, and B.A. Malloy. Grammar recovery from parse trees and metrics-guided grammar refactoring. *Software Engineering, IEEE Transactions on*, 35(6):780 –794, November 2009.

[KM05] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM Press.

[KMR08] Jens Knodel, Dirk Muthig, and Dominik Rost. Constructive architecture compliance checking – an experiment on support by live feedback. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM 2008)*, pages 287–296, 2008.

[Kno11] Jens Knodel. *Sustainable Structures in Software Implementations by Live Compliance Checking*. PhD thesis, Univ. of Kaiserslautern, Computer Science Department, 2011.

[Kop97] Rainer Koppler. A systematic approach to fuzzy parsing. *Software: Practice and Experience*, 27(6):637–649, 1997.

[Kos12] Rainer Koschke. Large-scale inter-system clone detection using suffix trees (to appear). In *Proceedings of CSMR2012*, 2012.

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

[LKA$^+$95] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[LLBO07] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. Codegenie:: a tool for test-driven source code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 917–918, New York, NY, USA, 2007. ACM.

[LV01] Ralf Lämmel and Chris Verhoef. Semi-automatic grammar recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[LVD06] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 492–501, New York, NY, USA, 2006. ACM.

[MDBZ09] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, IWPSE-Evol '09, pages 57–62, New York, NY, USA, 2009. ACM.

[MK06] Kim Mens and Andy Kellens. IntensiVE, a toolsuite for documenting and checking structural source-code regularities. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10 pp. –248, mar 2006.

[MN97] Gail C. Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 8:29–36, 1997.

[Moc09] Audris Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Proceedings of MSR 2009*, pages 11–20, 2009.

[Moo01] Leon Moonen. Generating robust parsers using island grammars. In Elizabeth Burd, Peter Aiken, and Rainer Koschke, editors, *Proceedings Eight Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE Computer Society, October 2001.

[OBL10] J. Ossher, S. Bajracharya, and C. Lopes. Automated dependency resolution for open source software. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 130 –140, may 2010.

[Pin05] Martin Pinzger. *ArchView — Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Vienna University of Technology, 2005.

[PM02] Richard Pawson and Robert Matthews. *Naked Objects*. Wiley and Sons, 2002.

[Rei09] Steven P. Reiss. Semantics-based code search. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[RL08] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[SMDV06] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM.

[Szy02] Clemens A. Szyperski. *Component Software — Beyond Object-Oriented Programming.* Addison Wesley, second edition edition, 2002.

[SZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. Hatari: raising risk awareness. *SIGSOFT Softw. Eng. Notes*, 30:107–110, September 2005.

[TX07] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[Vis97] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[WH05] C.C. Williams and J.K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *Software Engineering, IEEE Transactions on*, 31(6):466 – 480, June 2005.

2.2 Current state of own research

The Software Composition Group has extensive experience in the research areas covered by this proposal. In this section we summarize some of the prior research most relevant to this proposal. All cited papers are available from the SCG web site: `http://scg.unibe.ch/publications`.

Meta-Tooling Software developers need tools to help them understand and analyze complex software systems. For example, integrating run-time analyses into the IDE can significantly improve the efficiency of certain software maintenance tasks [RHV$^+$11]. The kind of analysis required will depend on the domain and the development context, but building custom analysis tools is typically an expensive and time-consuming task. As a consequence, there is a need for meta-tooling, the facility to rapidly craft custom tools from available building blocks and to customize general tools to a specific task.

We have developed Moose, a platform for analyzing software and data [NDG05]. FAMIX, the Moose meta-model, is extensible, thus allowing custom tools to be built that extend the meta-model with new concepts. Examples include modeling version histories [GD06], semantic clustering of classes and packages by lexical content [KDG07], and feature views that track which software artifacts support which user features [GDG06].

General tools are also needed to present and visualize software models. Polymetric views [LD03] map software metrics to simple visualizations. They can be used to generate a wide range of useful analyses, such as clone evolution [BGM06] (showing the evolution of clones over time), distribution maps [DGK06] (showing how features relate to artifacts), and ownership maps [GKSD05] (showing who has worked on which parts of a system over time).

Mondrian [MGL06] is a visualization engine that makes it much easier to build such visualizations. Since its host language, Smalltalk, has a very simple syntax, composing new visualizations resembles writing scripts in a DSL. A lightweight, interactive UI allows developers to immediately see the result of changes to their Mondrian scripts. Mondrian supports agile software assessment by enabling developers to generate new visualizations in minutes instead of days.

Data exploration is a fundamental component of data analysis. *Glamour* [Bun09] offers a generic engine to construct custom browsers for arbitrary models (not just source code). A browser is built up from a number of components, each of which renders some fragment of a model, and *transmits* information about that fragment to another connected component. A classical source code browser with panels displaying packages, classes, methods and source code can be constructed as a compact Glamour script. Dedicated browsers for other kinds of models can similarly be scripted with low effort.

These examples illustrate the benefits of meta-tooling, but they only constitute some small steps towards an expressive meta-tooling environment to support agile software assessment.

Agile Modeling A key bottleneck for agile software assessment is the automated construction of software models. Although Moose, for example, can readily import models from Java source code, importing models from other languages, such as PHP, Cobol or PL/1, has proven to be problematic, since a custom parser must be developed to generate FAMIX models. This can entail days or even weeks of effort.

We have explored several innovative techniques to speed up this process, such as generating parsers automatically from examples [NKG+07], recovering models from the abstract or concrete syntax trees generated by existing parsers [LV08], and even using genetic programming to generate grammars [Zan09]. Although some degree of success was achieved in each case, we are far from having practical techniques to rapidly import models from new languages or dialects.

We have also worked on the development of language workbenches. Helvetia [RGN10] is a PEG-based language workbench built on top of PetitParser, and designed to extend a host language with new, embedded DSLs in such a way that the development tools of the host language can easily be adapted to the new embedded languages. This is achieved by making the compiler toolchain, the browser, and debugger aware of language transformations.

A related issue is that modern software systems are rarely implemented using a single source language. Java enterprise applications, for example, include not only Java source code but also HTML, JavaScript, XML, SQL and possibly other kinds of artifacts. To analyze such systems, we have constructed software models that combine all these sources of information [PGN10, APL+11].

We have also carried out considerable research that exploits the presence of other implicit information in code, such as topics revealed by pure lexical analysis of the source text [KDG07].

Large-Scale Software Analysis We have studied large corpora of systems in order to discover general principles of software evolution. In searching for good predictors for software change we discovered that contrary to common wisdom, the classes with the highest fan-in are also the ones that change the most [VSN07]. We have also shown that the majority of classes undergo only minor changes during the evolution of the system [VSNW08] which appears to be at odds with the "law of continuous change" of Lehman.

As other examples, we have analyzed bug reports to establish developer expertise [MKN09], and we have analyzed cross-project activity of developers to establish code trustability [GK10].

At the opposite end of the spectrum of software corpus analysis is the analysis of software ecosystems — systems that are developed with common technology, and that evolve together in a common context. A number of problems that are relevant for individual system analysis remain relevant at the ecosystem level. The importance of some of these problems is even augmented. And some of these problems can be better solved if the ecosystem is taken into account.

In one study we evaluated several techniques for automatically extracting dependencies between the systems in an ecosystem through static source code analysis and discovered that simple techniques can go a long way but often code duplication and dynamic programming languages can hinder the analysis [LRL10]. We further investigated in a case study how critical is the phenomenon of ripple effects at the ecosystem level and discovered that some changes in libraries and frameworks can have a very large impact on an ecosystem: dozens of projects and developers can be forced to update to new version of a library, but the developers of these libraries and frameworks lack tools which would allow them to predict the impact of a change [RL11].

We have proposed a technique for reverse engineering a software ecosystem through the recovery of *ecosystem viewpoints* from the versioning systems of the component projects [Lun09]. To support our methodology we implemented tool support in our web-based prototype dubbed The Small Project Observatory [LLGR10].

Since in earlier work dependency analysis was hindered by the presence of duplicated code, we shifted our attention towards large scale detection of duplicated code [SLR12]. We discovered that more than 14% of the methods in a large open source Smalltalk ecosystem were clones. Our clone detection technique detects type I, II, and III clones. Our approach is the only one we are aware of that performs cloning analysis on all the versions of all the systems in an ecosystem. To be able to scale to the dimensions of our data we had to employ techniques of *big data* research and build an infrastructure based on map-reduce algorithms which allowed us to massively distribute the computation.

Architectural Monitoring Our research to date has encompassed the study of architecture specification to support the process of forward engineering. In the context of this research we developed *Piccola*, a language for specifying applications as composition of components conforming to a particular architectural style [NA00]. With Piccola a style can be specified as a set of operators over various types of components. Piccola serves as an executable ADL, specifying how components in the host language, Java, are composed and interact.

Another major focus of our research is on developing techniques to reverse engineer complex software systems. This work has culminated in a practical handbook of *reverse engineering patterns* [DDN08] for recovering design and architecture. Several PhD theses undertaken in our group have expressly addressed architectural recovery. Richner combined static and dynamic information to recover software architectures [RD99]. Arévalo applied formal concept analysis to recover implicit patterns in software systems [ABN04]. Greevy used run time information to correlate user features with software components that support them [GDG06]. Lungu used automatic clustering techniques [LL06], multi-version analysis [LL07], and interaction-based collaborative tool support to recover aspects of architecture from source code [LN12].

We are currently extending our architecture recovery and visualization prototype, Softwarenaut, with dashboard-based visualizations of the evolving structure of a software system. We are also extending the tool with collaborative features such as the Global Architectural View Repository, an online repository hosted in the cloud which allows developers to share and discover architectural views indexed by the version and the system they are analyzing [LLN12].

We have also developed techniques to monitor run-time behavior towards the goal of understanding the system under analysis. Live feature analysis uses reflection to instrument and annotate the running system with the goal of obtaining more detailed information about the use of features than is possible with classical post-mortem analysis techniques [DRGN10].

References

[ABN04] Gabriela Arévalo, Frank Buchli, and Oscar Nierstrasz. Detecting implicit collaboration patterns. In *Proceedings of WCRE '04 (11th Working Conference on Reverse Engineering)*, pages 122–131. IEEE Computer Society Press, November 2004.

[APL+11] Amir Aryani, Fabrizio Perin, Mircea Lungu, Abdun Naser Mahmood, and Oscar Nierstrasz. Can we predict dependencies using domain information? In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE 2011)*, October 2011.

[BGM06] Mihai Balint, Tudor Gîrba, and Radu Marinescu. How developers copy. In *Proceedings of International Conference on Program Comprehension (ICPC 2006)*, pages 56–65, 2006.

[Bun09] Philipp Bunge. Scripting browsers with Glamour. Master's thesis, University of Bern, April 2009.

[DDN08] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns.* Square Bracket Associates, 2008.

[DGK06] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[DRGN10] Marcus Denker, Jorge Ressia, Orla Greevy, and Oscar Nierstrasz. Modeling features at runtime. In *Proceedings of MODELS 2010 Part II*, volume 6395 of *LNCS*, pages 138–152. Springer-Verlag, October 2010.

[GD06] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, 18:207–236, 2006.

[GDG06] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 18(6):425–456, 2006.

[GK10] Florian S. Gysin and Adrian Kuhn. A trustability metric for code search based on developer karma. In *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2010. SUITE '10.*, 2010.

[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 2005)*, pages 113–122. IEEE Computer Society Press, 2005.

[KDG07] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, March 2007.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[LL06] Mircea Lungu and Michele Lanza. Softwarenaut: Exploring hierarchical system decompositions. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 351–354, Los Alamitos CA, 2006. IEEE Computer Society Press.

[LL07] Mircea Lungu and Michele Lanza. Exploring inter-module relationships in evolving software systems. In *Proceedings of CSMR 2007 (11th European Conference on Software Maintenance and Reengineering)*, pages 91–100, Los Alamitos CA, 2007. IEEE Computer Society Press.

[LLGR10] Mircea Lungu, Michele Lanza, Tudor Gîrba, and Romain Robbes. The Small Project Observatory: Visualizing software ecosystems. *Science of Computer Programming, Elsevier*, 75(4):264–275, April 2010.

[LLN12] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. Evolutionary and collaborative software architecture recovery with Softwarenaut. *Science of Computer Programming (SCP)*, page to appear, 2012.

[LN12] Mircea Lungu and Oscar Nierstrasz. Recovering software architecture with Softwarenaut. *ERCIM News*, 88, January 2012.

[LRL10] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *ASE'10: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM Press, 2010.

[Lun09] Mircea Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, November 2009.

[LV08] Daniel Langone and Toon Verwaest. Extracting models from IDEs. In *2nd Workshop on FAMIX and Moose in Software Reengineering (FAMOOSr 2008)*, pages 32–35, October 2008.

[MGL06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[MKN09] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 131–140. IEEE, 2009.

[NA00] Oscar Nierstrasz and Franz Achermann. Supporting Compositional Styles for Software Evolution. In *Proceedings International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 11–19, Kanazawa, Japan, November 2000. IEEE.

[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'05)*, pages 1–10, New York, NY, USA, 2005. ACM Press. Invited paper.

[NKG$^+$07] Oscar Nierstrasz, Markus Kobel, Tudor Gîrba, Michele Lanza, and Horst Bunke. Example-driven reconstruction of software models. In *Proceedings of Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 275–286, Los Alamitos CA, 2007. IEEE Computer Society Press.

[PGN10] Fabrizio Perin, Tudor Gîrba, and Oscar Nierstrasz. Recovery and analysis of transaction scope from scattered information in Java enterprise applications. In *Proceedings of International Conference on Software Maintenance 2010*, September 2010.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 13–22, Los Alamitos CA, September 1999. IEEE Computer Society Press.

[RGN10] Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Embedding languages without breaking tools. In Theo D'Hondt, editor, *ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming*, volume 6183 of *LNCS*, pages 380–404, Maribor, Slovenia, 2010. Springer-Verlag.

[RHV$^+$11] David Röthlisberger, Marcel Härry, Alex Villazón, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. Exploiting dynamic information in ides improves speed and correctness of software maintenance tasks. *Transactions on Software Engineering*, 2011. To appear (preprint online).

[RL11] Romain Robbes and Mircea Lungu. A study of ripple effects in software ecosystems (nier). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 904–907, May 2011.

[SLR12] Niko Schwarz, Mircea Lungu, and Romain Robbes. On how often is code cloned across repositories. In *Proceedings of ICSE 2012*, page to appear, 2012.

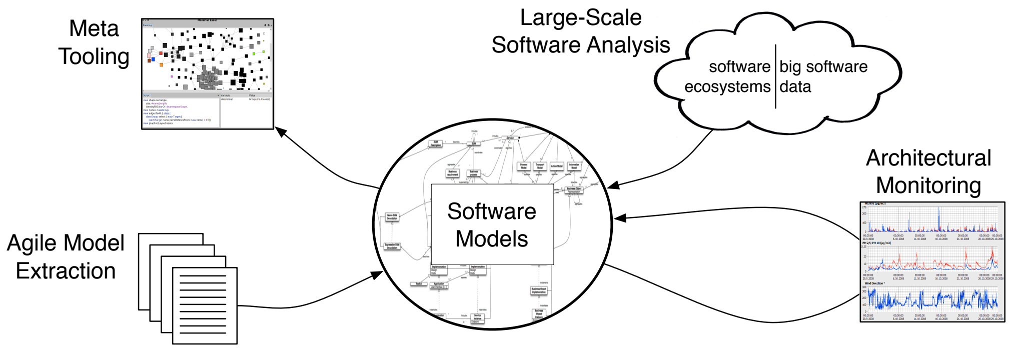
[VSN07] Rajesh Vasa, Jean-Guy Schneider, and Oscar Nierstrasz. The inevitable stability of software change. In *Proceedings of 23rd IEEE International Conference on Software Maintenance (ICSM '07)*, pages 4–13, Los Alamitos CA, 2007. IEEE Computer Society.

[VSNW08] Rajesh Vasa, Jean-Guy Schneider, Oscar Nierstrasz, and Clint Woodward. On the resilience of classes to change. In Tom Mens, Maja D'Hondt, and Kim Mens, editors, *Proceedings of 3d International ERCIM Symposium on Software Evolution (Software Evolution 2007)*, volume 8. Electronic Communications of the EASST, 2008.

[Zan09] Sandro De Zanet. Grammar generation with genetic programming — evolutionary grammar generation. Master's thesis, University of Bern, July 2009.

2.3 Detailed Research Plan

The proposed research is organized into four complementary tracks that address support for software developers in rapidly and efficiently assessing complex software systems. Each track can be seen as a producer or a consumer of software models, or both.



1. *Meta-Tooling* focuses on agile development of customized software assessment tools.
2. *Agile Modeling* addresses the problem of efficiently and effectively constructing high-level software models from program source code and other data sources.
3. *Large-Scale Software Analysis* explores the application of "big data" analysis techniques to extract information useful for developers from the broader ecosystem of a software project.
4. *Architectural Monitoring* explores how to characterize and monitor architecture, violations of architectural constraints, and architectural evolution.

2.3.1 Meta-Tooling

Considerable time is devoted to reading, understanding and assessing code to effectively support decision-making in the development process. Existing integrated development environments (IDEs), however, focus mainly on low-level programming tasks, and offer little to the developer to support software assessment. Although numerous analysis tools have been developed over the years, few of these tools are well-integrated into the development lifecycle and the decision-making process. What is missing is an effective and efficient way to obtain the right tool for the problem at hand.

In this research track we propose to explore the theme of *Meta-Tooling, i.e.*, tools for developing tools, as a means to support agile software assessment. Instead of focusing on integrating a fixed set of software analysis and assessment tools into the IDE, we focus on integrating a set of key software assessment mechanisms, and offering a meta-tooling framework that allows these mechanisms to be easily configured to address a particular software assessment task, such as identifying error-prone code, locating components that support user features, or identifying the impact of a proposed change. This work is inspired by our previous work on generic tools like Mondrian and Glamour, which allow users to quickly develop visualizations and model browsers using a dedicated DSL.

We envision three activities within this track: first, empirical studies are needed to explore which assessment tasks are most important to support developers in their daily work, and how developers can be supported in those tasks. Second, we plan to explore the design space of mechanisms for software assessment by developing experimental tools and mechanisms that enable developers to better exploit and interpret the data at hand. Third, we plan to develop a meta-tooling environment to allow developers to rapidly compose and configure custom assessment tools.

Study: what assessment tools do developers need? We plan to carry out empirical studies with developers to determine what kinds of assessment tools they need to support routine development and evolution tasks.

We plan to use a variety of different techniques to gather this information, including interviews, field studies and controlled studies. We seek to better understand what kinds of questions are posed by both experienced developers as well as novices when they must develop and maintain complex systems, and we want to understand and classify the techniques they use to routinely answer these questions.

With the help of controlled studies, we also hope to confirm that such tools and mechanisms can help developers with varying degrees of experience to become more productive.

Software assessment mechanisms. In tandem with the empirical work, we plan to develop novel tools and mechanisms to offer software assessment support to developers.

Developers must cope with large amounts of information coming from a variety of sources. Before developers can use this information to support development tasks, they must find it, understand it, process it, and relate it to their task at hand. We plan to develop generic tools and transformation engines to support software assessment. Some of the ideas we plan to explore include:

— *Software exploration:* Glamour supports a rather rigid model of browsers built around a workflow in which information flows from one pane of the browser to another. We need to explore more general models of browsers in which, for example, a browser view may directly influence and manipulate the underlying model.

— *Views:* Model elements and information sources can be presented in different ways, depending on how the data are to be used. An approach based on naked objects could be used to directly associate multiple views with model elements.

— *Bridging data sources:* We need mechanisms to integrate multiple data sources. One possible mechanism is to make use of tables (or spreadsheets) some of whose fields are simply views of existing data sources. Tables can be used to aggregate and analyze such heterogeneous sources, as well as to pass processed data on to alternative, graphical views.

We plan to experiment with mechanisms that can be motivated by our empirical studies, and conversely plan to carry out controlled studies to assess the effectiveness of the developed mechanisms.

Meta-tooling environment. Most IDEs do not easily accommodate new functionality. An environment like Eclipse, for example, allows users to rearrange and customize the individual windows of the environment, and new tools can be downloaded and installed as "plug-ins", but developing a new plug-in is highly non-trivial. We envision, by contrast, a "malleable" IDE which can be easily extended with new software assessment tools composed from the more basic mechanisms described above. This activity requires the development of a *meta tooling environment*, *i.e.*, tools for building tools.

The focus of this work is to develop a *unifying meta-model* for software assessment mechanisms. Such a meta-model will determine how various tools, mechanisms and data sources play with one another. This meta-model will be validated experimentally through the development of a component framework for composing tools and their parts. Compositions may be specified in various ways, such as through the use of a high-level DSL or scripting language, or even through drag-and-drop direct manipulation.

2.3.2 Agile Modeling

A key prerequisite for analyzing software models is to construct the model from the available program and data sources. For example, tools are available to import Moose models for many programming languages, including Java, Smalltalk and C++. The situation is very different, however, in the case of dialects of known languages, source code in new languages, source code containing embedded code in one or more DSLs (domain specific languages), and hybrid source code in multiple languages (such as Java, JSP, XML, SQL *etc.*, as is common in enterprise applications).

Developing a parser for a new language is time-consuming and expensive. Even though techniques such as scavenging grammars from various sources can reduce this time, still several weeks may be needed to develop a usable parser for a typical mainstream language. This cost can pose a major impediment to adopting software assessment tools for an evolving project, as new components potentially implemented in different languages and dialects are integrated over time. The goal of this track is to develop techniques that would enable new program and data sources to be modeled within a single day. Today, no current available tools or environments meet this goal.

The following characteristics of the problem lead us to believe this goal is reachable:

— Most programming languages can be classified into groups with similar features. Using appropriate techniques, one can decompose parsers into parts that resemble one another.

— Complete parsers are not needed for software modeling. In an initial phase, it may be enough to capture coarse structure. A grammar can be refined as more details are needed.

— Typically, a large base of existing code samples are available for analysis. This fact can be exploited to automatically generate and test parsers.

In particular, we propose to explore the following techniques to support agile modeling:

Reusable parser fragments. PEGs (Parsing Expression Grammars) support the definition of parsers from composable parts. Scannerless parsers furthermore avoid the need for a separate lexical pass, and avoid the need to commit to lexemes of a given language. By composing scannerless parsers for different languages, one can cope with heterogeneous code bases using multiple embedded languages.

We plan to exploit similarities between programming languages to develop generic classes of parser fragments that can be adapted and composed to extract models from languages.

In this track we plan to analyze a number of existing programming languages and categorize them according to the similarities (and differences) of their language fragments. We plan to implement scannerless PEGs for common language fragments, and experiment with way to parameterize them, or even to generate them, taking language differences into account. We also plan to experiment with heterogeneous code to test the ability of reusable parser fragments to deal with embedded code in a reusable way.

Guided parser refinement. Since complete parsers are not needed for initial software analysis, we plan to explore an approach in which an initial grammar is defined as a coarse island grammar that can be iteratively refined by the software modeler. The refinement process will be guided by the user, for example, by highlighting sample code fragments and specifying their mapping to model elements (*i.e.*, classes, methods, statements *etc.*).

We will explore techniques to automatically generate PEG parser fragments for examples specified by the user. Since we will be based on PEGs rather than (say) LALR parsing, no conflicts can arise. Since the parsers are scannerless, we need not commit to a set of language tokens up front. And since we will use island grammars, we can guarantee that all code can always be parsed robustly. Feedback must be provided to the user, however, to indicate how much code is effectively modeled (*i.e.*, as "islands") and how much is ignored (*i.e.*, as "sea").

Structural inference While user intervention will be needed to guide the parser refinement process, automation can help to provide the user with hints how to proceed. We plan to explore a series of techniques to mine information useful for parsing from the existing base of source code.

A few promising avenues are as follows:

— Exploiting indentation to identify structural elements to model.

— Analyzing recurring names to distinguish potential keywords from identifiers.

— Analyzing source code text to identify comment conventions.

— Exploiting language similarities by generating island grammars for common language fragment parsers.

— Mutating parser fragments or instantiating them based on source code analysis.

Finally, we also intend to explore the application of these to techniques to structured data related to the development lifecycle, such as profiling data and bug reports.

2.3.3 Large-Scale Software Analysis

The volume (amount of data), velocity (speed with which the data is generated), and variety (range of data types, sources) of available information associated with evolving software systems is growing. Data sources include bug reports, mailing list archives, issue trackers, dynamic traces, navigation information extracted from the IDE, and meta-annotations from the versioning system. All these sources of information have a time dimension, which is tracked in versioning control system, and which adds an order of magnitude to the amount of available information about a given software system.

Research has benefited from increased resource capacity, network bandwidth and raw computational power to analyze all of these artifacts at possibly massive scales. Therefore it follows naturally that current software engineering research can use the new wealth of information to improve the productivity of software developers and the quality of their software.

There are two directions that we plan to pursue in the context of analyzing large numbers of projects: big software data and ecosystems research. Big software data considers information at its lowest abstraction level as files and lines of source code and applies data analysis techniques on it without caring about the system organization. Ecosystem analysis on the other hand focuses on solving the inherent problems existent in the context of interacting software systems.

We plan to pursue several research directions in this track:

Modeling ecosystem evolution. Although the big software data is still not as big as the data that other sciences (*e.g.*, astronomy, meteorology) are handling on a daily basis, there still is a lack of infrastructure that would enable fast querying of an evolving ecosystem.

The infrastructure should be tailored for the particularities of evolving software:

— Source code evolves *forward* in time. Once a version is committed to the versioning repository, it does not change anymore. Every structural artifact in the software (*e.g.*, method, class, package) once published, is frozen. This suggests that data warehousing approaches may be applied.

— Source code duplication is common, from micro-duplication patterns, and boilerplate code, to product families and branches in the versioning system. We can use this information to minimize the analysis effort, and optimize storage (*e.g.*, hashing files by their content eliminates the need for storing duplicates)

— Software data is a combination of highly structured and unstructured data. The highly structured data are the source files and the unstructured are the associated artifacts (*e.g.*, documentation, emails from the mailing lists). One can extract a large number of relationships between the different artifacts of the source code (*e.g.*, containment, calling, *etc.*).

We propose to devise techniques to model an evolving ecosystem to allow the fast access and querying of the data, and to leverage the specific properties of big software data to improve the productivity of software developers and the quality of their software.

One of the first family of queries that we would like to support is related to impact analysis at the ecosystem level. The developers of a system should be able to easily query the way their source code is used by other projects and the users of a library should be able to easily query usages of that project. Currently there is no infrastructure that supports this.

To allow the reuse of frequent queries, and avoid costly analysis on the client side, we will explore the possibility of hosting the data and the analysis in the cloud.

Large-scale static analysis. The history of the source code is an invaluable resource for the study and improvement of software engineering techniques. The scope of big software data analysis can be much broader than that of the ecosystem and include all the source code that has ever been recorded for a given programming language or in a given super-repository of source code, or even all the available source code. In one research track we will focus on performing static analysis on the big software data. Based on such analysis one can achieve a broad variety of goals from improving the IDE to improving the programming languages themselves:

Embedding intelligence and adaptation in the IDE. Modern IDEs are rigid pieces of software: with very few exceptions, they do not learn from their usage. We plan to explore the ways in which the IDEs can be improved through big data analysis. Several directions worthy of exploration are:

— Suggesting code completion and code navigation based on recording the IDE interaction data of large numbers of developers

— Suggesting code snippets, and documentation that might be useful for the code that the user is writing at the moment. Given the large amounts of code and the complexity of current libraries (*e.g.*, the Java SDK of Sun contains more than 3,500 classes organized in more than 200 packages) this can be very useful.

— Suggesting tests based on existing tests for similar code

Evolving programming languages based on mining their usage. By mining large source code repositories, one can discover various properties of programming languages:

— Frequency of use for different programming language features

— Usability aspects of the programming languages

— Recurring patterns of boilerplate code

The first two types of information can guide the design of new programming languages or dialects. The third type can be used to generate libraries of high-level abstractions that can replace boilerplate code.

Large-scale dynamic analysis. Run time data of a software system can be a rich source of information that can improve the quality of the system as well as the quality of the development process. Dynamic analysis tools and techniques are conventionally limited to a single run of a single project. There is nothing inherent in these techniques that requires this to be so, aside from a question of scale. By creating a central repository of run time information, we can enable new analyses and improve the precision of existing ones:

— *Statistical typing.* Polymorphism and dynamic binding prevent the IDE from knowing what concrete types variables assume. By creating a unique, centralized repository of typing information in the cloud, indexed by the version of every system — since a version once published is immutable — as soon as a given version is loaded in an IDE the corresponding statistical annotations can be loaded from the cloud. This would allow for better tool support during development.

— *Temporal specification mining.* Data mining the central repository of program traces will allow temporal specifications to be extracted from the source code. This information can be used during development time to detect potential errors. The challenge here is in managing the large amounts of data that are generated in dynamic analysis, finding a way of compressing these data, and at the same time providing fast access.

We plan to explore these and other possible scenarios for exploiting large amounts of run time data to support developers. The key challenges in this research direction are discovering how to efficiently generate, transmit, and store the dynamic information in a central repository without impacting the performance of the running applications.

2.3.4 Architectural Monitoring

Although much work has been dedicated to extracting and recovering the current architecture of a system from the source code and other available artifacts, little is being done on re-using these techniques in the context of forward engineering. Monitoring the evolution of architecture is one such approach and it promises to improve system stability, quality, and robustness. To monitor the evolution of an architecture presupposes the existence of an architectural specification, or its recovery through reverse engineering.

Study: classifying software architecture in the wild. There has been considerable prior work on so-called Architectural Description Languages (ADLs), but their absence from the software engineering mainstream leads us to question their relevance to real software systems. We believe that one of the main reasons is the failure of current ADLs to capture properties of the system that are actually important for software engineers. Indeed, we are not aware of any empirical studies that have attempted to study and classify best practices in specifying real software architectures. We therefore propose to study both open source and industrial systems and interview developers. The goal will be to collect architectures "in the wild", and to characterize and classify them. By eliciting information from the developers we will be able to understand what aspects of software architecture are especially relevant in practice. Based on this we will be able to propose requirements for a means to express the dominant concerns of the developers and facilitate the monitoring of the system's evolution. We plan to publish the collected architectures so they will serve as a ground truth, or benchmark, for further comparing and assessing the effectiveness of architectural recovery techniques.

Monitoring architectural evolution. Software systems evolve over time, and when the architecture is disconnected from the source code, the two will follow diverging evolution paths, resulting in what is called architectural drift.

Whether the architecture is still valid and the design needs to be brought in line, or whether the original architecture is obsolete and needs to be updated to current needs, it is crucial to track architectural evolution to detect as early as possible that a problem exists.

We will address the following research questions related to monitoring software architecture:

— How to *best specify the architecture of a system?* We plan to develop techniques to specify architecture in ways that reflect best practice and actual needs of developers based on our case studies of open source and industrial systems. We plan to explore ways to encode architectural constraints (and the properties they are intended to ensure) that are flexible enough to express both the structure of the system and its behavior. We also plan to explore various means to express architectural constraints formally within the programming language, such as annotations and coding conventions, as opposed to external representations in UML or an ADL. One core aspect of our work on specifying architecture will be usability. We will insure that the mechanisms we propose for constraint specification make the most common architectural constraints easy to express.

— How to *enforce the correct evolution of the system*? The specified architecture, or the recovered architecture must evolve together with the system. To prevent the architecture from drifting away from the code, the constraints encoded in the architecture must be enforced during system evolution. This suggests that feedback mechanisms would be useful to inform developers when proposed changes conflict with the system's architecture. When violations to the architecture are inevitable for any reason, mechanisms must be in place to allow the exceptions to be made explicit, so that long term maintenance is facilitated.

— How to *present the evolving architecture of a system*? Starting from the empirical study we will work towards the design of an effective presentation layer for the evolving architecture of a system. We will design and evaluate an *architectural dashboard* that would allow the different stakeholders to monitor the co-evolution of architecture and source code in a system. We plan to experiment with simple and lightweight visualizations at different levels of abstraction to indicate architectural constraints and their violations, and more generally to indicate the co-evolution of the system and its architectural description.

— How to *monitor architectural evolution in the ecosystem*? In our previous work we have observed the need to support impact analysis at the ecosystem level referring mainly to changes that impact the API of a system. We plan to study how to monitor architectural evolution in the ecosystem. Specifically we will study how to enable the evolution of a system when changes that will impact its architecture occur in upstream libraries or frameworks. By monitoring the entire ecosystem, and data mining the way the other projects adapt to changes to the same upstream, one should be able to estimate the effort required to adapt as well as to recommend changes required for the adaptation.

2.4 Schedule and milestones

Here we provide a coarse timeline for each of the planned research tracks.

Year 1	
Meta-Tooling	Study: establish developer needs for software assessments Develop experimental software assessment mechanisms
Agile Modeling	Classify languages; develop reusable parser fragments Generate island grammars from example to model mappings
Large-Scale Software Analysis	Collect information needs for developers working in open source software ecosystems Develop infrastructure for fast querying of ecosystem models
Architectural Monitoring	Study: collect and curate a repository of architectural descriptions Experiment with techniques for architecture recovery
Year 2	
Meta-Tooling	Establish a unifying meta-model for software assessment mechanisms Develop an experimental environment for composing custom assessment tools
Agile Modeling	Exploit lexical features to identify structural elements Experiment with parameterized parser fragments
Large-Scale Software Analysis	Experiment with the static analysis of big software data; experiment with improving the IDE based on big data analysis
Architectural Monitoring	Develop an experimental language to specify architectural constraints Explore means to present architectural evolution
Year 3	
Meta-Tooling	Study: explore and assess effectiveness of meta-tooling Further experimentation with assessment mechanisms and tooling
Agile Modeling	Carry out extensive case studies with code and data sources Experiment with further techniques to infer structure
Large-Scale Software Analysis	Experiment with the dynamic analysis of big software data
Architectural Monitoring	Explore techniques to integrate architectural constraint checking with monitoring Study the impact of such a system in an organization or community

2.5 Importance and impact

Effective decision-making is one of the key challenges to continuous development of complex software systems. This project tackles this challenge by proposing to develop techniques to rapidly and effectively extract useful software models from diverse data sources, monitor them, and analyze them in a timely fashion. The expected long term benefits of this research are improved developer efficiency and better quality software.

We have a strong track record of publishing results in the usual academic venues (full papers in high impact journals and international, peer-reviewed conferences), and will continue to disseminate our research through these venues. We also regularly take advantage of invitations for keynote presentations at conferences to present a broader perspective of our research projects than is possible with a technical paper presentation.

We have ongoing collaborations with software developers at various Swiss companies (Crédit Suisse, Zühlke, CompuGroup, Swiss Federal Institute of Intellectual Property). In the context of this project, and particularly in the empirical studies, we plan to exploit these ties as well as seek out new partnerships.