Change-Enabled Software System

Oscar Nierstrasz, Marcus Denker, Tudor Girba, Adrian Lienhard, David Roethlisberger

Change-Enabled Software Systems*

Oscar Nierstrasz, Marcus Denker, Tudor Gîrba,
Adrian Lienhard, David Röthlisberger

Software Composition Group, University of Bern
http://scg.unibe.ch/

Abstract. Few real software systems are built completely from scratch nowadays. Instead, systems are built iteratively and incrementally, while integrating and interacting with components from many other systems. Adaptation, reconfiguration and evolution are normal, ongoing processes throughout the lifecycle of a software system. Nevertheless the platforms, tools and environments we use to develop software are still largely based on an outmoded model that presupposes that software systems are closed and will not significantly evolve after deployment. We claim that in order to enable effective and graceful evolution of modern software systems, we must make these systems more amenable to change by (i) providing explicit, first-class models of software artifacts, change, and history at the level of the platform, (ii) continuously analysing static and dynamic evolution to track emergent properties, and (iii) closing the gap between the domain model and the developers' view of the evolving system. We outline our vision of dynamic, evolving software systems and identify the research challenges to realizing this vision.

1 Introduction

Software inevitably changes, but our development methods, programming languages, development environments and run-time systems generally assume that one is building a closed, internally consistent application, which will not significantly change after deployment. Anticipated evolution can be built in to some extent, for example by applying well-known design patterns, but unanticipated changes in requirements are hard to accommodate without reengineering the system, redeploying it, and possible migrating persistent data.

The vision of an *eternal software-intensive system* is that of a system that can survive such unanticipated changes with little or no human intervention at the lowest level [59]. We claim that this vision can only be realized if *software change is enabled* in a fundamental way in our platforms, run-time environments and development environments [42]. In particular, not only software systems themselves, but their development and support environments need to be far more *dynamic* than they are today. Specifically, what does this entail?

* M. Wirsing et al. (Eds.), *Software-Intensive Systems*, LNCS 5380, pp. 64-79, 2008.
 doi:10.1007/978-3-540-89437-7_3

- First of all, we need to provide *platforms* in terms of programming languages and run-time environments that make it possible to *manipulate and operate on change as a first-class entity*. This in turn implies that an evolving software system is not only model-driven, but actually "self-aware" — it must have a first-class representation of itself available to enable change. To control the scope of change, change itself should be represented as a first-class, high-level entity. To manage change over time, the history of the system must also be accessible and first-class (see Section 2).
- Second, an evolving software system must be *capable of analyzing itself*, and in particular of recognizing *emergent properties*. This means that the evolution of the static and dynamic models must be monitored, and the resulting data be analyzed as the system is running (see Section 3).
- Third, to enable continuous evolution, a software system must *close the gap between the development and deployment views* of itself. Domain models, usage models, and features, for example, must be made explicit in the system to facilitate change (see Section 4).

We will explore these themes in some detail, in each case summarizing previous work, and establishing a research agenda for further work. We conclude with summary remarks about next steps.

2 Self-aware platforms to support change

Traditionally, the development and deployment of software are viewed as being separate in time and space: first a system is developed, then it is deployed. Indeed, in the classical view, we deal with two completely different artifacts: the source code that can be developed, debugged and understood on the one hand, and on the other hand a generated, closed, non-understandable binary program that just can be run.

Classical development plays out like a finite game with fixed rules and boundaries. Evolving software systems, on the other hand, are better thought of as *infinite games* without fixed rules or boundaries [5]. Evolving systems will not have a clear separation of development and deployment. The system will continue to evolve when it is already deployed. The systems of the future will not be developed from the outside as a finite game. Development itself will be part of the infinite game of the system. Evolution needs to happen in parts of the system, while it is running.

We cannot afford to stop and restart a continuously evolving software system, just as we cannot stop and restart the Internet. The Internet has been up and running since 1969, although many of its atoms have been changed many times since then. The software intensive systems of the future will need to learn from these loosely-coupled, long-lived systems. To support this view, we need appropriate core technologies in terms of programming languages and run-time systems that can serve as a platform for developing evolving software systems.

2.1 Previous work

In order to enable change at run-time, an evolving system must be able to fully reflect on itself, that is, it must be *self-aware*. It is not enough to be model-driven. The models must be explicit and accessible to the run-time system. A *reflective system* provides a description of itself available from within. This description can be queried (*introspection*) as well as changed (*intercession*). In the past, computational reflection has been an active area of research [13,15,41,55]. Nevertheless, languages used in industry today do not provide full reflection, and many mainstream languages have no self representation at all (for example C and C++). More recently created languages like Java and C# support limited introspection, but no intercession.

In recent years, dynamic languages have attracted much attention [43] and are increasingly being used in industry (*e.g.*, Python, Ruby, Smalltalk). Dynamic languages provide a representation that can be queried and changed at run-time and thus are reflective. But even in these dynamic languages, the support for reflection is limited. The *structural* representation of the program stops at the granularity of the method. Classes and methods are represented as objects and available to be queried and changed, but the structure of the methods themselves is not represented. *Behavioral* reflection is limited as we cannot change behavior on a fine-grained level. In addition, when applying behavioral reflection to the system (as opposed to an application), the programmer will soon run into the problem of meta-object call recursion [11].

We have extended structural reflection to model the structure of methods: sub-method reflection [8] represents the complete structure, down to the code itself. The representation can be annotated and thus can be used for integrating tools. One example is feature annotation [10]. Instead of recording full traces to analyze features, we can simply annotate the static structure of the system with feature information. Partial behavioral reflection [51,58] provides means to select where and when a meta-object is activated and allows us to define which information is passed to the meta-object. We can introduce behavioral changes at run-time which provides the basis to supporting unanticipated change to the systems. Examples range from tools like tracers or profilers [51] to changes of the language semantics, for example transactional memory [47]. The problem of meta-object call recursion is solved by representing meta-level execution as a context and by making meta-object activation context-aware [11].

It is well-established that suitable abstractions are needed to enable programming in the large [45]. But in the case of scale, we need to think again: are existing abstractions good enough for *very large* software systems? For example, as software gets larger the assumption that every part of the system must stay in sync with every other part is not very convincing because the systems of the future will be so large that we will never be able to evolve them in a single, synchronous step. As a consequence, an evolving system must be able to cope with multiple, inconsistent views of itself.

Inconsistency is only tolerable if specific and individual views appear to be locally consistent. Instead of allowing all changes to be globally visible, we need a means to control the scope of changes. That is, evolving systems must support a notion of *context* and the run-time infrastructure must be *context-aware*. Being able to dispatch on context means that we need to support a form of context-oriented programming [7,27]. Visibility of changes can then be restricted to the context in which these changes are guaranteed to be valid.

Changeboxes [9] provide a mechanism for encapsulating change as a first-class entity in a running software system. Changeboxes support multiple, concurrent and possibly inconsistent views of software artifacts within the same running system. Since changeboxes are first-class, they can be manipulated to control the scope of change in a running system. Furthermore, changeboxes capture the semantics of change. Changeboxes can be used, for example, to encapsulate refactorings, or to replay or analyze the history of changes.

2.2 Research agenda

We maintain that both *reflection* and *context* are crucial to support change and evolution. There has been much recent progress, but more research is needed. In particular, the key ideas emerging from previous research need to be consolidated and integrated into a comprehensive model.

Efficient and practical reflective languages. Sub-method structural reflection and partial behavioral reflection are improvements over conventional reflective systems. One problem with reflection, even with efficient partial reflection [51,58] is performance. With behavioral reflection, we introduce new behavior that replaces the default behavior. One example is method lookup. The default lookup is extremely optimized and realized in the virtual machine, so any reflective redefinition is often slower by an order of magnitude. This difference in performance can render a system unusable in practice. We need to integrate reflection better into the virtual machine, leveraging the dynamic code generator of modern just-in-time compilers. Another interesting question is how to resolve static typing with reflection. Type-systems reason about properties that can be guaranteed in the future, whereas reflection is about changing the future. We need a way to check reflective change before it is applied to the system.

Backward compatibility. Backward compatibility is the enemy of forward evolvability. Nevertheless, we cannot live in a world where the old is ignored. An often overlooked property of software is that new systems can simulate the old, and the recent trends in hardware virtualization have shown that simulation of the old is far easier than for the new to stay compatible. A snapshot of an old Windows machine can run on a virtual machine forever, whereas keeping an operating system compatible forever is bound to fail. Programming languages for evolving systems should provide backwards compatibility in the same way: we need a first class description of the history of all code of the system, freeing the present from being compatible with the past while at the same time providing the possibility to go back in time easily. The system should provide complete, runnable snapshots of itself at any point in the past. Our work on changeboxes forms one first step towards this goal. However, changeboxes are only concerned with code, thus an important aspect of future work is the problem of migrating data between versions.

Contextual reflection. Context appears to be deeply related to reflection. In the case of changeboxes, the currently active changebox provides a context for execution. For partial behavioral reflection, we solve the problem of meta-object call recursion by representing the execution of the meta-level as a context [11]. We think that the next step is to revisit these cases and integrate the notion of context into the reflective model of the language and the virtual machine.

Languages supporting multiple views. In general, we need to explore multi-dimensional object systems. After achieving a reflective model that is aware of context, the next step is to build language support that makes this concept available to the programmer. Some very relevant work has been done in the past, for example the work on PIE [4,22,23,24], Us [56] and more recently ContextL and ContextS [7,27]. More research is needed to explore how to combine these ideas with contextual reflection and changeboxes.

Evolving languages. Evolving systems need languages that support continuous development and evolution. But there is another aspect when considering the language itself: to think that we can envision the perfect language to realize all future systems is to treat language design like a finite game. Thus a language suited for implementing ever-evolving software systems needs to be *itself* an evolving system. An evolving language must evolve to incorporate new ideas and practices while it is used. It needs to be extensible and growable from within [57].

3 Monitoring and analyzing change

To change a system we must first understand the system and the consequences of change. Since change inevitably causes the system to drift from its initial documentation, the most reliable source of information is the system itself. A self-aware system can reflect on its own specification, which is an aid to static analysis. But emergent properties as well as program failures can only be monitored with the help of dynamic analysis [26].

Ideally, a productive system should constantly monitor and analyze itself. This would allow us to discover properties that are only visible over a longer period of time, such as performance degradation, memory leaks, shifts in how the system is used, effects of structural changes etc. Furthermore, collecting detailed data about the program execution can provide crucial information to uncover the cause of program failures.

3.1 Previous work

Program failures for large, long-lived software systems can be hard to reproduce, and hard to simulate in a test environment. As a consequence, systems need to dynamically analyze their behavior to gather execution history while they are running and allow for remote and safe debugging inside the live system. To be able to reason about the run-time behavior of a system, dynamic information has to be linked with the model of the software structure. A platform that provides an integrated, high-level model of its own structure, design, and behavior (see Section 2) would offer an appropriate foundation for building self-analyzing systems.

The challenge we face with such an architecture is that the current state of the art in dynamic analysis does not permit run-time information to be gathered below the method level in live systems due to performance reasons. The main obstacles are (i) code instrumentation requires a system to be restarted, (ii) run-time overhead can be huge (up to a factor of 100 or more for non-trivial programs [34]), and (iii) available memory to store the gathered data limits the analysis to only few and short user sessions.

Most existing dynamic analysis approaches are detached from the run-time environment, *i.e.*, virtual machine, and hence have only limited means to adapt and reconfigure the system at runtime without restarting it. Instrumentation code is weaved into the application code at compile time, for instance through bytecode manipulation. This additional code then generates data during execution, which is either stored as a trace of events in memory or in a database where it is processed *post mortem*. An evolving system that needs to be running all the time cannot be restarted to reconfigure the analyzer. Therefore, the run-time analysis of a self-aware system needs to be an integral part of its run-time environment. Like this, no hard-wired instrumentation code is required, but rather the analyzer is a self-aware component of the virtual machine. The analyzer needs to be always running and capable of adjusting its own behavior, much like garbage collectors are always active in modern virtual machines.

We have addressed some of these problems as follows [37,39,38]. We have extended the object memory model of conventional object-oriented virtual machines by representing object references as real objects on the heap. In this way we seamlessly integrate historical execution data into the object model of the virtual machine. Our approach discards unneeded historical execution data by employing the standard garbage collector of the VM. We showed that this approach can dramatically improve the data explosion problem and has much lower execution overheads compared to other back-in-time debuggers [34,46] that are not implemented at the virtual machine level.

We have extended the common dynamic analysis model with the notion of *object aliases*. That is, object references are explicitly represented in our model, which allows us to track the flow of objects in the system or to analyze how side effects are produced [40,38].

By enabling dynamic analysis on live systems, innovative debugging and analysis techniques come within reach. In a recent study, Liblit *et al.* examined bug symptoms for various programs and found that in 50% of the cases the execution stack contains essentially no information about the bug's cause [35]. Back-in-time debuggers [28,34,46] allow the developer to explore the program state at those points in time that are no longer represented on the run-time stack. We are currently exploring the development of a highly performant back-in-time debugger on top of our history-aware VM.

In evolving software systems, the changes to the static parts are directly accessible as first class entities. As such, in evolving software systems, not only the run-time is dynamic, but also the static part is dynamic when seen from an historical perspective. Treating history as a first-class entity enables analyses of the evolution of software artifacts [17]. For example, we can predict where changes are likely to occur [19], we can detect classes that are changed frequently [21], or we can identify crosscutting concerns by detecting which parts change at the same time and in the same way [18].

Given the size of evolving systems, they will not be developed by an isolated team, but rather by several teams that are physically distributed. In this context, the social aspect of the development will become increasingly important [6]. Thus, analysis should also include reasoning about how developers collaborate [2,20,25].

3.2 Research agenda

In the long-term, we expect that run-time monitoring of program execution and evolution will become not only practical but essential to the survival of long-lived software systems. To make this a reality however, further research will be needed in the following areas.

Efficient run-time analysis. Dynamic analysis will only be widely adopted if it is cheap in time and space. The emergence of multi-core architectures for off-the-shelf desktop and laptop machines suggests that parallelization should be explored as one way to reduce the execution time. Even though future hardware capacity will allow for storing more data more efficiently, more advanced models for discarding unneeded data are important, as faster running systems also produce data faster.

Detecting emergent properties. High-level run-time models are needed to reason about the system from within the system. In an object-oriented system, recording method execution events alone is not enough for certain analyses and for debugging. Also, a run-time model should be seamlessly connected with the static model, which captures structural program entities and their evolution. This allows for recognizing links between the behavior of a system and its evolution and can serve as a source of information for maintainers in their development environment. By correlating static and dynamic information over time, one should be able to detect emergent properties, such as maintenance hot spots, performance bottlenecks and other opportunities for refactoring and reengineering.

Automatic model reconstruction. Heterogeneous and legacy sources of information can be obstacles to analysis and evolution. Yet another complicating factor is the use of different programming languages and media such as mainstream compiled languages, scripting languages, domain specific languages, HTML, XML and query languages within the same system. Furthermore, some of the languages used will be either legacy languages or dialects (such as legacy dialects of C++). Post-hoc parsing of components built with these languages will be difficult and error-prone since the original language specifications may not be available. Thus, an evolving software can be seen as a multi-dimensional space of data that needs to be continuously analyzed. Techniques will be needed to automatically reconstruct high-level models and meta-models from lower-level data, without necessarily having up-to-date access to the syntax specifications.. Possible approaches include abstraction from examples [44] and formal concept analysis [1,36], amongst many others.

4 Enabling change for the developer

The evolution, or rather continuous development, of evolving systems places special demands on the development environment. To some extent systems can be designed for evolution. But if we see the development of an evolving system as an infinite game, it becomes clear that one cannot anticipate all forms of evolution.

Current IDEs focus on providing the developer only with a static view of the source code without offering any information about how the code is actually executed at run-time, about why a bug occurred, or about whether there are performance issues or memory consumption problems. Furthermore, modern IDEs do not provide means to bridge the gap between the users' view and the developers' view of the system. For instance, it is hard to locate and understand a specific feature of a large system by studying its sources code alone.

We envision a development environment in which change is enabled by bridging the static and dynamic views of the system and by bringing the results of dynamic analysis to the IDE. Ultimately, the IDE should be an active player in the development process, enabling developers to interactively manipulate and extend the system at a high level of abstraction.

4.1 Previous work

Support for refactoring, reorganizing and reengineering must be part of the evolving system. The state-of-the-art in refactoring support is still in its infancy [16]. Many modern IDEs provide some automated mechanisms to change and evolve software systems but they tend to be low level, like renaming a class or moving a method [49]. Furthermore, developers receive little guidance in identifying opportunities for refactoring [33], and the knowledge about performed refactorings is usually not kept. A promising approach is to offer a better versioning system that is able to store the high level knowledge about a change [48] and then provide this information for further analysis [12].

Empirical studies report that a developer performing maintenance tasks on a system spends at least 35% of the time in navigating source code [31]. A maintenance-oriented IDE should present the developer with a working set of source code containing all functionality for a specific maintenance task to reduce the navigational load. By monitoring the programmer's activity to get a degree-of-interest for program elements scattered across a large code base, the IDE can reveal code elements that are likely to be important for the task at hand [30]. Other proposals [50,54] focus on emphasizing relations between source artifacts by recommending related artifacts based on the past sequence of browsing, or by characterizing the kinds of changes that have taken place during development sessions [48].

Hermion is an experimental IDE that brings run-time information to the developer to better support maintenance tasks [53]. Dynamic information gathered at run-time provides the developer with the possibility to directly navigate to actual senders and implementors of messages, or to navigate to the actual concrete types to which variables have been dynamically bound. Statistical data about the run-time call graph is also integrated into the static code views, and is dynamically updated as the code under development is further exercised. Dynamic monitoring is realized by means of mechanisms in the underlying reflectivity framework (see Section 2) which support unanticipated partial behavioural reflection [51]. This allows running code to be instrumented on-the-fly at a high level of abstraction, and without modifying the underlying source code.

Visualizations can convey complex information in a condensed and effective manner. Extensions to the VisualWorks Smalltalk IDE (*e.g.*, RBCrawler) integrate various well-known visualizations such as class blueprints [14], polymetric views [32] or system complexity views [32]. They are well integrated in the static source navigation tools of this IDE, however, they do not take dynamic information into account and are thus just a starting point for further work.

One particularly useful application of visualization is to correlate static views of software components, with the features of the running systems. Feature-driven browsing exploits run-time information gathered when exercising features and presents this information as a visual map [52]. This enables us to quickly identify the components responsible for a feature and to highlight the dependencies between different features. As this feature visualization is integrated in the IDE, the developer can also better assess the impact of a source code level change on various features of the system if there is an explicit mapping between features and source artifacts available.

4.2 Research agenda

Today's IDEs are largely passive players in the development process, though there is a clear and gradual trend towards IDEs and tools that play a more active role. We believe that further research is needed here to make the IDE a more active player in enabling change. Some promising research tracks follow.

Automatic and continuous execution. Not only should the IDE provide complete information, it should also do this efficiently to avoid slowing down the development process. To achieve efficient integration of dynamic information, we envision applying the concept of continuous integration by running the system automatically and continuously in the background. This relieves the developer from the burden of having to manually trigger the execution of the system while modifying it.

Full coverage. Since manually triggered dynamic analyses normally do not cover the entire system (*i.e.*, all system's features), the IDE should assume the responsibility to ensure full coverage and cover all code still being used by the application. The IDE should analyze all execution paths through the system by running the code currently being studied by the developer. If system monitoring and analysis (see Section 3) can be made sufficiently efficient and non-intrusive, possibly by exploiting the possibilities of parallel hardware, information gathered from the execution of the live system by real users and certainly also of recorded scripts could also be fed automatically back to the IDE.

Exploiting fine-grained change histories. Software is currently developed following a checkout/change/commit life cycle. This approach hides the local events and changes from the overall development. To limit this loss of information developers are advised to commit as often as possible in the central repository. In the future we envision a central environment that is tightly integrated with the versioning system and that stores all changes performed on the system. Furthermore, instead of being snapshot-based, change logs will capture the *intent* of changes, such as common refactoring operations. This information can then be fed back into the IDE so that developers can immediately access historical information in a productive and integrated way.

Autonomous system evolution. The IDE as an active player should be able to autonomously perform many maintenance tasks. For instance, the IDE should be able to automatically detect defects, and with the help of dynamic analysis, suggest repair strategies to the developer. If a certain feature of such an evolving system is broken, the environment should be able to locate this feature in source code and determine which classes or methods need to be corrected to successfully solve the issue. Fine-grained historical information about changes to the software supporting the defective features should also be a valuable input to automatically provide focus to the developer to correct the defect.

Model-centric development. Software developers today generally edit and manipulate textual representations of the systems being developed, *i.e.*, as source code. However, text is by definition static while the resulting system is dynamic and neither knows nor cares about the textual representation from which it has initially been built. We envision a system in which the developer directly manipulates high-level models that more closely represent the conceptual entities of the software application. Such an approach would be *model-centric* rather than "model-driven", since models would directly represent the running application, rather than serving to generate it. Such a system could be realized by a visual programming environment that enables developers to develop and change directly the dynamic structure, *e.g.*, by working with visual components of a system's behavior that can be dragged, dropped, parameterized, adapted, extended and reorganized, directly modifying the behavior.

5 Concluding remarks

We have argued that traditional approaches to software development do not adequately support the inevitable change that software will undergo during its lifetime. We have summarized some current and ongoing research activities to enable change for long-lived software systems, and we have presented an agenda of critical research topics for further investigation. Briefly, we see the need for further research in the following three related areas:

– *Self-aware platforms to enable change.* In order to enable change for software applications, the underlying platform (*i.e.*, programming language and run-time environment) must support change in a deep way. Such systems must therefore be *reflective*, by means of *high-level models*. Change must be modeled as a *first-class entity*, so that changes can be manipulated and reasoned about. Finally, the platform must be *context-aware* so that the scope of change can be controlled to ensure safe evolution at run-time. We have argued that these ingredients are necessary for a future generation of self-aware, long-lived, evolving software applications. Some of these ideas have been prototyped, but much research is needed before they are ready to be adopted in mainstream platforms.
– *Monitoring and analyzing change.* A long-lived, evolving software system must provide means to monitor and analyze change, both in terms of short-term, dynamic adaptations, and long-term software evolution. Dynamic analysis today is mostly *post hoc*. For performance reasons it is generally impractical to monitor and analyze live systems. Novel techniques for monitoring, debugging and analyzing live systems are being developed, but further research is needed to make these approaches practical, for example by exploiting the possibilities of the emerging class of multi-core processors. Further research is also needed to detect emergent properties of long-lived evolving software systems, particularly in terms of correlating static and dynamic software structures. Automatic detection of maintenance hot spots, performance bottlenecks, and opportunities for refactoring are some of the areas where further research is needed.
– *Enabling change for the developer.* Modern application development environments essentially provide developers with static views of the application source code. Particularly in the case of object-oriented software development, the gap between the static and the run-time structures implies that much valuable information is simply not available to the developer. By bringing together static and dynamic views of the live system, many opportunities

present themselves to improve the productivity of the developer. For instance, dynamic information integrated into the static software views provides richer navigation possibilities. Further research is needed to make the IDE itself an active player that exercises, tests and analyzes the live system as it is being developed, actively updates the static views with the results of dynamic analysis, automatically detects defects and actively suggests repair strategies, and keeps track of changes and their intent at a finer granularity than is currently supported. As a long-term goal, we envision IDEs that are not so much focussed on editing source code, but rather support direct manipulation and transformation of *models* of the live system under development.

As a closing remark, we note that this kind of research is almost impossible to carry out effectively by isolated researchers. A certain critical mass in terms of software infrastructure is needed before research results stabilize and new ideas can be built on top of older ones. Much of current research results in prototypes that do not even begin to achieve the level of stability that is needed before other researchers can build something new on top of them.

On the one hand, we advocate that the research process will need to acknowledge and to reward the engineering effort. In the future, research and engineering must meet to face the wide space opened by evolving systems. On the other hand, just like evolving systems will not be the result of one team's work, we advocate that research will need to break the group boundary and open towards research networks [3].

Bringing together research and engineering will also bring together two worlds that are now rather separated: research and practice. Practitioners face real problems and need new ideas to solve these problems, but cannot afford the time and effort to experiment with unproven ideas. Researchers need real problems to develop new ideas, but cannot afford the effort to fully validate and mature these ideas in a practical setting. Each group is under pressure to get their products out the door with acceptable quality and minimum cost. As a consequence few new ideas get proven in practice, and real problems of practitioners tend not to propagate in the research environment. The perceived cost of collaboration is just too high.

A first step to bring these groups together and reduce the cost of collaboration is to provide an infrastructure in which new ideas can be quickly implemented, tested and adopted. The need for collaboration to build a successful infrastructure can be seen in the wide adoption of Eclipse as a platform [29]. Many teams contribute to Eclipse due to its open architecture, and many researchers are using it for implementing their vision. While Eclipse is not an academic exercise, it does facilitate software evolution research. To facilitate relevant and collaborative research into evolving software intensive systems, an analogous common infrastructure will be needed upon which both research and practice can build.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008) and that of the Hasler Foundation for the project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance".

References

1. 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.
2. Mihai Balint, Tudor Gîrba, and Radu Marinescu. How developers copy. In *Proceedings of International Conference on Program Comprehension (ICPC 2006)*, pages 56–65, 2006.
3. Warren Bennis and Patricia Ward Biederman. *Organizing Genius — The Secrets of Creative Collaboration*. Perseus Books, 1997.
4. Daniel G. Bobrow and Ira P. Goldstein. Representing design alternatives. In *Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior*, July 1980.
5. James P. Carse. *Finite and Infinite Games — A Vision of Life as Play and Possibility*. Ballantine Books, 1987.
6. Melvin E. Conway. How do committees invent? *Datamation*, 14(4):28–31, April 1968.
7. Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*, pages 1–10, New York, NY, USA, October 2005. ACM.
8. Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007.
9. Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 25–49. ACM Digital Library, 2007.
10. Marcus Denker, Orla Greevy, and Oscar Nierstrasz. Supporting feature analysis with runtime annotations. In *Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2007)*, pages 29–33. Technische Universiteit Delft, 2007.
11. Marcus Denker, Mathieu Suen, and Stéphane Ducasse. The meta in meta-object architectures. In *Proceedings of TOOLS EUROPE 2008*, volume 11 of *LNBIP*, pages 218–237, 2008.
12. Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien Nguyen. Refactoring-aware configuration management for object-oriented programs. In *International Conference on Software Engineering (ICSE 2007)*, pages 427–436, 2007.
13. Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

14. Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, January 2005.
15. Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 317–326, October 1989.
16. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
17. Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, 18:207–236, 2006.
18. Tudor Gîrba, Stéphane Ducasse, Adrian Kuhn, Radu Marinescu, and Daniel Raţiu. Using concept analysis to detect co-change patterns. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2007)*, pages 83–89. ACM Press, 2007.
19. Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, September 2004. IEEE Computer Society.
20. 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.
21. Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.
22. Ira P. Goldstein and Daniel G. Bobrow. Descriptions for a programming environment. In *Proceedings of the First Annual Conference of the National Association for Artificial Intelligence*, August 1980.
23. Ira P. Goldstein and Daniel G. Bobrow. Extending object-oriented programming in Smalltalk. In *Proceedings of the Lisp Conference*, pages 75–81, August 1980.
24. Ira P. Goldstein and Daniel G. Bobrow. A layered approach to software design. Technical Report CSL-80-5, Xerox PARC, December 1980.
25. Orla Greevy, Tudor Gîrba, and Stéphane Ducasse. How developers develop features. In *Proceedings of 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 256–274, Los Alamitos CA, 2007. IEEE Computer Society.
26. Abdelwahab Hamou-Lhadj and Timothy Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004)*, pages 42–55, Indianapolis IN, 2004. IBM Press.
27. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March 2008.
28. Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In *Proceedings of NODE'06*, volume P-88 of *Lecture Notes in Informatics*, pages 17–32. Gesellschaft für Informatik (GI), September 2006.
29. Steve Holzner. *Eclipse*. O'Reilly, May 2004.

30. 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.
31. Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 125–135, 2005.
32. 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.
33. Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.
34. Bill Lewis and Mireille Ducassé. Using events to debug Java programs backwards in time. In *OOPSLA Companion 2003*, pages 96–97, 2003.
35. Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pages 15–26, New York, NY, USA, 2005. ACM.
36. Adrian Lienhard, Stéphane Ducasse, and Gabriela Arévalo. Identifying traits with formal concept analysis. In *Proceedings of 20th Conference on Automated Software Engineering (ASE'05)*, pages 66–75. IEEE Computer Society, November 2005.
37. Adrian Lienhard, Stéphane Ducasse, and Tudor Gîrba. Taking an object-centric view on dynamic information with object flow analysis. *Journal of Computer Languages, Systems and Structures*, 2008. To appear.
38. Adrian Lienhard, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Test blueprints – exposing side effects in execution traces to support writing unit tests. In *12th European Conference on Software Maintenance and Reengineering (CSMR'08)*, pages 83–92. IEEE Computer Society Press, 2008.
39. Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.
40. Adrian Lienhard, Orla Greevy, and Oscar Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings International Conference on Program Comprehension (ICPC'07)*, pages 59–68, Washington, DC, USA, June 2007. IEEE Computer Society.
41. Pattie Maes. *Computational Reflection*. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium, January 1987.
42. Oscar Nierstrasz. Software evolution as the key to productivity. In A. Knapp M. Wirsing and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future*, volume 2941 of *LNCS*, pages 274–282. Springer-Verlag, 2004.
43. Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gaelli, and Roel Wuyts. On the revival of dynamic languages. In Thomas Gschwind and Uwe Aßmann, editors, *Proceedings of Software Composition 2005*, volume 3628, pages 1–13. LNCS 3628, 2005. Invited paper.
44. 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.

45. David L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, December 1972.
46. Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, 2007. To appear, ACM.
47. Lukas Renggli and Oscar Nierstrasz. Transactional memory for Smalltalk. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 207–221. ACM Digital Library, 2007.
48. Romain Robbes and Michele Lanza. Characterizing and understanding development sessions. In *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*, page to be published, 2007.
49. Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.
50. Martin P. Robillard and Gail C. Murphy. Feat: A tool for locating, describing, and analyzing concerns in source code. In *Proceedings of 25th International Conference on Software Engineering*, pages 822–823, May 2003.
51. David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures*, 34(2-3):46–65, July 2008.
52. David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Feature driven browsing. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 79–100. ACM Digital Library, 2007.
53. David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Exploiting runtime information in the ide. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC 2008)*, volume 0, pages 63–72, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
54. Janice Singer, Robert Elves, and Margaret-Anne Storey. Navtracks: Supporting navigation in software maintenance. In *International Conference on Software Maintenance (ICSM'05)*, pages 325–335, sep 2005.
55. Brian Cantwell Smith. Reflection and semantics in a procedural language. Technical Report TR-272, MIT, Cambridge, MA, 1982.
56. 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.
57. Guy Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, October 1999.
58. 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.
59. Martin Wirsing and Matthias Hölzl (editors). Report of the Beyond the Horizon thematic group 6 on Software Intensive Systems, 2006.