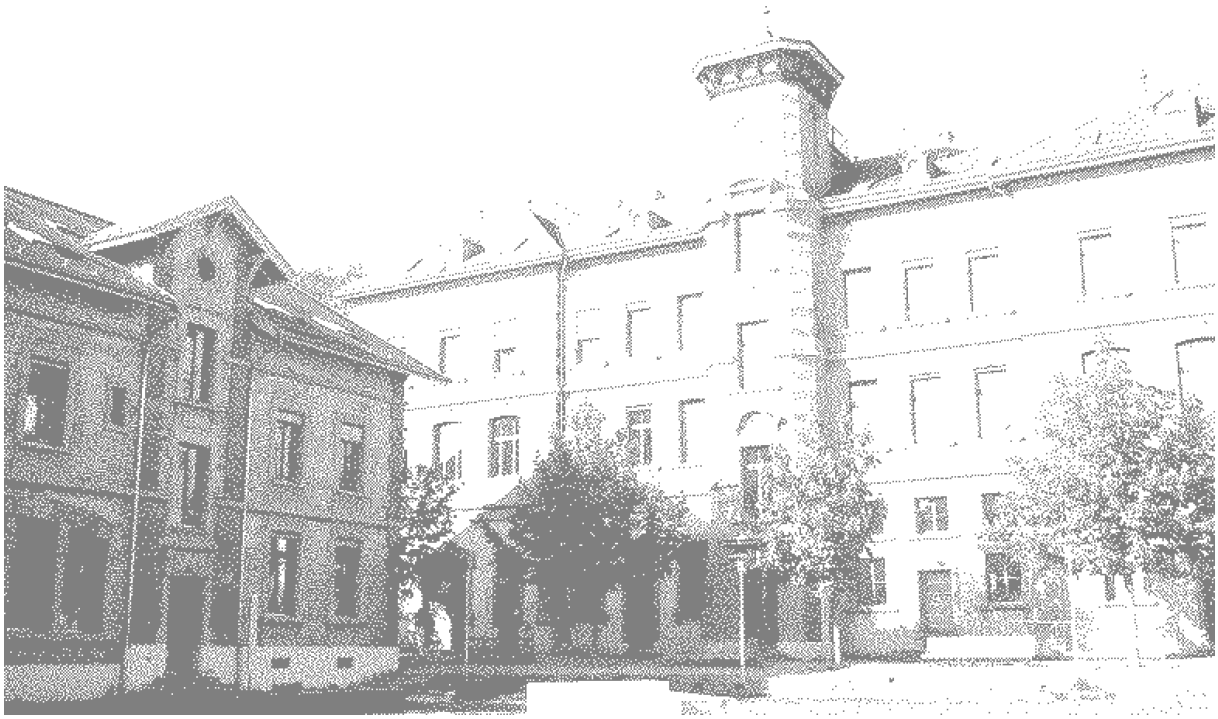
UNIVERSITÄT BERN

Self-aware, Evolving Eternal Systems

O. Nierstrasz, M. Denker, T. Gîrba, A. Kuhn, et al.

Technischer Bericht IAM-08-001 vom 31. Mai 2008

Institut für Informatik und angewandte Mathematik, www.iam.unibe.ch



Self-aware, Evolving Eternal Systems [1]

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

Technischer Bericht IAM-08-001 vom 31. Mai 2008

CR Categories and Subject Descriptors:
D.1.5 Object-oriented Programming; D.2.6 [Programming Environments]: Integrated environments

General Terms:
Languages

Additional Key Words:
self-aware, eternal systems, software-evolution

Institut für Informatik und angewandte Mathematik, Universität Bern

[1] This extended abstract was submitted to the InterLink Working Group Challenges for Software-Intensive Systems and New Computing Paradigms

Contents

1 Abtract

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. These systems also last longer than their developers might imagine — they are, in effect, eternal. 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 eternal systems, we must make them self-aware. A self-aware eternal system supports evolution 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 self-aware eternal systems and identify the research challenges to realizing this vision.

2 Eternal Systems

Software inevitably changes, but our development methods, programming languages, development environments and run-time systems provide little that acknowledges this fact. There is a general assumption behind most tools and methods 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 [1]. We claim that this vision can only be realized if software evolution is supported in a fundamental way in our platforms, run-time environments and development environments [2]. Specifically, what does this entail?

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 eternal 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. And to manage change over time, the history of the system must also be accessible and first-class (see Section 3).

Second, a self-aware eternal 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 4).

Third, to enable continuous evolution, a self-aware eternal 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 5).

Finally, concrete incentives are needed to bring research and practice closer together (see Section 6).

3 Self-aware Platforms for Eternal Systems

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. Eternal software-intensive systems, on the other hand, are better thought of as infinite games without fixed rules or boundaries [3]. Eternal 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 an eternal software system, just as we cannot stop and restart the Internet. The Internet has been up and running since 1969, although each atom that it is physically made of has been replaced 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 runtime systems that can serve as a platform for developing eternal systems.
In order to enable change at runtime, an eternal 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. Furthermore, changes to the model and the system must be explicit and manipulable, so changes must have a first-class representation. Finally, in order to reason about the impact of changes, it is essential that the history of the system must also be fully accessible and manipulable.
It has been very early understood that abstractions are needed for making programming in the large possible. But with scale, we need to think again: are existing abstractions good enough for very large software systems? One example is that as software gets larger, the idea that every part of the system is in sync with any other part is not very convincing: in huge systems, the parts will indeed be inconsistent. Evolution itself will lead to inconsistency, as the system is so large that we can never evolve the complete system at the same time. As a consequence, an eternal 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, eternal 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 [4]. Visibility of changes can then be restricted to the context in which these changes are guaranteed to be valid.
Backward compatibility is the enemy of forward-evolvability. Nevertheless, we cannot live in a world were the old is ignored. A snapshot of an old Windows machine can run on a virtual machine forever, whereas keeping an operating system compatible forever will be bound to fail. Programming languages for eternal 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 the system at any point in the past.
Eternal systems need languages that support continuous development and evolution. But there is another aspect to the language: 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, eternal software systems needs to be itself an eternal program. An eternal language must evolve to incorporate new ideas and practices while it is used. It needs to be extensible and growable from within [5].

4 Analyzing Self-aware Eternal Systems

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. However, documentation will not only be provided in form of documents, but also in various other forms like online discussions, bug reports, and description of the changes [6]. Therefore, software analysis tools for reverse engineering are central to support software evolution [7].
A self-aware system can reflect on its own specification, which is an aid to static analysis. But the run-time architecture and other emergent properties can only be monitored with the help of dynamic analysis [8]. A self-aware, eternal system must be capable of tracing and analyzing its run-time behaviour while it is online, much like garbage collectors are always active in modern virtual machines.
In eternal software systems, the changes to the static parts are directly accessible as first class entities. As such, in eternal software systems, not only the run-time is dynamic, but also the static part is dynamic when seen from a historical perspective. Treating history as a first-class entity enables analyses of the evolution of software artifacts [9].
Given the size of eternal 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 [10]. Thus, analysis will also consist of reasoning about how developers collaborate.
Yet another complicating factor is the use of different languages and media within the same system. Furthermore, some of the languages used will be either legacy languages or dialects. For this reason, post-hoc parsing of components built with these languages will be difficult and error-prone. Thus, a software eternal software can be seen as a multi-dimensional space of data that needs to be continuously analyzed.

5 Environments for Evolving Eternal Systems

The evolution, or rather continuous development, of eternal 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 eternal system as an infinite game, it becomes clear that one cannot anticipate all forms of evolution. 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. Many modern IDEs provide some automated mechanisms to change and evolve a software-intensive systems. For instance, they support automated refactorings such as renaming a class or moving a method from one class to another [11]. Eclipse, for example, is very well adopted in industry, with more than half of all Java developers using this environment in their daily work [12]. Nevertheless, automated refactorings tend to be low-level, the fact that code has been refactored is not evident in the code base, and developers obtain no guidance in identifying opportunities for refactoring.

Examples of promising research directions that would help to support the development of eternal systems include mining system histories to guide developers in implementing or adapting features [13], and guiding developers to the use and reuse of existing components [14, 15].

A further problem with modern IDEs is the difficulty of bridging 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 in a system consisting of a large number of classes. Furthermore, there is a significant gap between the static, class-oriented view of the source code and the dynamic, object-based view of the run-time system. To build a comprehensive mental map of a large system the developer needs to navigate many classes and methods, a process which is only poorly supported by Eclipse and most other modern IDEs.

Empirical studies report that a developer performing maintenance tasks on a system spends at least 35% of the time in navigating source code to get a understanding for the implementation of a specific feature [16]. 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 [17]. By monitoring the execution of common

user tasks, an eternal system can correlate features with software artifacts [18] and use this information to drive development tasks related to those features.

Static and dynamic analysis of eternal systems, as outlined in Section 4, can be tightly integrated into the development environment to help drive evolution. For instance, visualizations of the system or of a specific feature should be accessible directly in the programming environment. As the system evolves these visualizations have to evolve as well to constantly reflect the changing structure and behavior.

6 Fostering Research in Eternal Systems

Not only do we need to research new ways of dealing with software evolution, but the way to conduct the research needs to be adapted to the task. The greater the body of existing work, the more people will research the area, and the more difficult it will be for a single person to achieve a reasonable delta within a reasonable amount of time. As a consequence, it becomes essential to have an infrastructure that allows the state of the art to be readily applied.

Currently PhD students serve as the engine of much research. Typically a PhD student must ensure that the research carried out be distant enough from that of his peers that there be no overlap in terms of the new ideas. Hence, the PhD student will tend to implement and validate his work alone. As a consequence, although the new ideas may spread, there is no incentive to ensure that the implementation will survive to be used by others. The next PhD student will have to re-implement those ideas before he can build on them. This process does not scale, as the research space grows larger.

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 eternal systems. On the other hand, just like eternal 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 [19].

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 product 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 [20]. Many teams contribute to Eclipse due to its open archi-

tecture, 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 eternal software intensive systems, a 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).

References

[1] M. Wirsing and M. H. (editors), "Report of the Beyond the Horizon thematic group 6 on Software Intensive Systems," 2006.

[2] O. Nierstrasz, "Software evolution as the key to productivity," in Radical Innovations of Software and Systems Engineering in the Future (A. K. M. Wirsing and S. Balsamo, eds.), vol. 2941 of LNCS, pp. 274–282, Springer-Verlag, 2004.

[3] J. P. Carse, Finite and Infinite Games — A Vision of Life as Play and Possibility. Ballantine Books, 1987.

[4] P. Costanza and R. 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, (New York, NY, USA), pp. 1–10, ACM, Oct. 2005.

[5] G. Steele, "Growing a language," Higher-Order and Symbolic Computation, vol. 12, pp. 221–236, Oct. 1999.

[6] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in Proceedings IEEE Working Conference on Reverse Engineering (WCRE 2003), (Los Alamitos CA), pp. 90–99, IEEE Computer Society Press, Nov. 2003.

[7] O. Nierstrasz, M. Denker, T. Gîrba, and A. Lienhard, "Analyzing, capturing and taming software change," in Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06), July 2006.

[8] A. Hamou-Lhadj and T. Lethbridge, "A survey of trace exploration tools and techniques," in Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004), (Indianapolis IN), pp. 42–55, IBM Press, 2004.

[9] T. Gîrba and S. Ducasse, "Modeling history to analyze software evolution," Journal of Software Maintenance: Research and Practice (JSME), vol. 18, pp. 207–236, 2006.

[10] M. E. Conway, "How do committees invent?," Datamation, vol. 14, pp. 28–31, Apr. 1968.

[11] D. Roberts, J. Brant, and R. E. Johnson, "A refactoring tool for Smalltalk," Theory and Practice of Object Systems (TAPOS), vol. 3, no. 4, pp. 253–263, 1997.

[12] G. Goth, "Beware the march of this IDE: Eclipse is overshadowing other tool technologies," IEEE Software, vol. 22, no. 4, pp. 108–111, 2005.

[13] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in 26th International Conference on Software Engineering (ICSE 2004), (Los Alamitos CA), pp. 563–572, IEEE Computer Society Press, 2004.

[14] Y. Ye and G. Fischer, "Reuse-conducive development environments," Autom. Softw. Eng., vol. 12, no. 2, pp. 199–235, 2005.

[15] D. Cubranic and G. Murphy, "Hipikat: Recommending pertinent software development artifacts," in Proceedings 25th International Conference on Software Engineering (ICSE 2003), (New York NY), pp. 408–418, ACM Press, 2003.

[16] A. J. Ko, H. Aung, and B. 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, pp. 125–135, 2005.

[17] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, (New York, NY, USA), pp. 159–168, ACM Press, 2005.

[18] O. Greevy, S. Ducasse, and T. Gîrba, "Analyzing software evolution through feature views," Journal of Software Maintenance and Evolution: Research and Practice (JSME), vol. 18, no. 6, pp. 425–456, 2006.

[19] W. Bennis and P. W. Biederman, Organizing Genius — The Secrets of Creative Collaboration. Perseus Books, 1997.

[20] S. Holzner, Eclipse. O'Reilly, May 2004.