

A Reflective Approach to Dynamic Software Evolution

Peter Ebraert¹ and Tom Tourwe^{1,2}

¹ Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium

² Centrum voor Wiskunde en Informatica, P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands

Abstract. In this paper, we present a solution that allows systems to remain active while they are evolving. Our approach goes out from the principle of separated concerns and has two steps. In the first step, we have to make sure that the system's evolvable concerns are cleanly separated. We propose aspect mining and static refactorings for separating those concerns. In a second step, we allow every concern to evolve separately. We present a preliminary reflective framework that allows dynamic evolution of separate concerns.

1 Problem Statement

An intrinsic property of a successful software application is its need to evolve. In order to keep an existing application up to date, we continuously need to adapt it. Usually, evolving such an application requires it to be shut down, however, because updating it at runtime is generally not possible. In some cases, this is beyond the pale. The unavailability of critical systems, such as web services, telecommunication switches, banking systems, etc. could have unacceptable financial consequences for the companies and their position in the market.

Redundant systems [1] are currently the only solution available to solve this problem. Their main idea is to provide a critical system with a duplicate, that is able to take over all functions of the original system whenever this latter is not available. Although this solution has been proved to work, it still has some disadvantages. First of all, redundant systems require extra management concerning which software version is installed on which duplicate. Second, maintaining the redundant systems and switching between them can be hard and is often underestimated. What would happen for instance when the switching mechanism fails? Would we have to make a redundant switching mechanism and another switching mechanism for switching between the switching systems? Last, duplicate software and hardware devices should be present, which may involve severe financial issues.

The principle of separation of concerns [2] could provide an improved and more flexible solution to the problem. Applications developed with this principle in mind implement every concern in a separate entity. These entities can then be adapted and substituted without affecting the rest of the application. Depending on the programming paradigm used, an entity can be a function, an abstract data type, a class or a component, for example, or even an aspect if we employ aspect-oriented programming techniques. Whenever an application is decomposed into cleanly separated entities, its evolution boils down to the addition, the removal or the modification of such an entity. If such activities can be performed while the application is running, we call such evolution *dynamic software evolution*.

In practice, the principle of separation of concerns is not always that easy to achieve. As it turns out, no matter how well an application is decomposed into modular entities, some functionality always cross-cuts this modularisation. This phenomenon is known as the *tyranny of the dominant decomposition* [3]. As a consequence, such cross-cutting functionality (often called a *concern*) can not be evolved separately, as it affects all other entities in the application.

Although techniques exist for addressing the problem of the dominant decomposition [4–7], they should be considered too static for supporting dynamic evolution. In effect, they provide a model in which cross-cutting concerns are fixed in the application at compile time. To solve this issue, more dynamic techniques should be investigated. Several prototypes of those techniques do exist: [8, 9], but still lack some dynamic properties as well as practical experience.

2 Towards Separated Concerns

Most currently existing applications do not match with the principle of separation of concerns. This is a serious problem if we want to allow them to evolve dynamically. In order to cope with this problem, we should investigate techniques that are able to discover cross-cutting concerns in existing code, as well as techniques that are able to restructure such code so that it becomes well modularised.

2.1 Aspect mining

Research in the domain of *aspect mining* is concerned with the identification of cross-cutting concerns in existing applications. Although such research is still in the early stages, several prototype tools have already been developed that support developers in identifying cross-cutting code. Many of these tools are semi automatic, which means they require some form of input by the developer [10–12]. More advanced tools, that are able to identify aspects without human intervention, are appearing as well however [13–15]. Our aim is to study if and how these techniques can be used for our purposes. One particular shortcoming of these techniques we already identified is that they do not take into account the dynamic behaviour of the application under consideration. In order to evolve an application dynamically, it is important to know which of its parts are weakly or strongly connected, which communication patterns occur frequently, etc. This is impossible with current-day aspect mining techniques. We are thus considering extending one of them with *reflectional capabilities*, so that we can study and observe a running application and infer often recurring communication patterns.

2.2 Object- and aspect-oriented refactoring

Once the cross-cutting concerns have been identified, we need to restructure the application in order to make it well modularised according to these concerns. Refactoring techniques, as proposed by [16], allow us to modify the internal structure of an application while preserving its overall behaviour. Whenever possible, we will use these refactorings and the appropriate object-oriented features to separate concerns cleanly. However, since some concerns may then still be cross-cutting, our aim also is to investigate the use of refactorings for restructuring an ordinary object-oriented application into an aspect-oriented one. This may involve extending the already existing refactoring techniques, and defining completely new refactorings, specifically targeted toward aspects. Preliminary steps in this direction are taken by [12] and [17].

3 Towards Dynamic Software Evolution

Once we are dealing with well modularized applications – applications with no cross-cutting concerns – we want to allow every module to evolve separately. In this section we present a reflection-based framework that will permit that.

3.1 Reflective systems

A reflective system is able to reason about itself by the use of *metacomputations* – computations about computations. For permitting that, such a system is composed out of two levels: the *base level*, housing the base computations and the *metalevel*, housing the metacomputations. Both levels are said to be *causally connected*. This means that, from the base level point of view, the application has access to its representation at the metalevel and that, from the metalevel point of view, a change of the representation will affect ulterior base computations. Depending on which part of the representation is accessed, the part describing the structure of the program, or the part describing its behavior, reflection is said to be *structural* or *behavioral*.

Figure 1 illustrates the causal connection between base and metalevel, and shows how this can be used in order to change the behavior or the structure of a base-level application. The left part of the figure shows the architecture of a certain application that has cleanly separated entities at the base level. The metalevel houses a representation of this application. The application can use that in order to reason about itself (introspection). Through manipulations of that representation (introspection, the application could self-evolve. The center picture shows that a new entity is added in the metalevel representation of the application. The right picture shows the propagation of the metalevel change down to the base level, thus changing the application's behavior and structure. Using this approach we can update separated entities of a system without having to switch off the system, and thus allow dynamic evolution. Still there are several issues that have to be solved in order to do so.

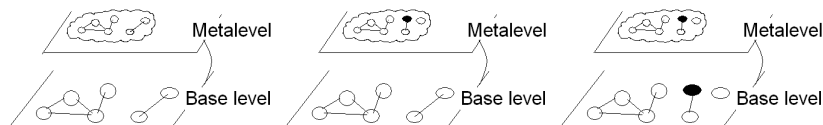


Fig. 1. Dynamically updating an entity through metalevel manipulation.

3.2 The evolution framework

We need a platform that provides both structural and behavioral reflection at runtime and that allows dynamic composition of meta-entities. As a first step, such entities will be aspects. Since we need structural reflection at runtime, we are going to experiment with Smalltalk. The behavioral reflection part will have to be added, based on the ideas of partial behavioral reflection as exposed in [18] and materialized in the Reflex platform for Java. Finally, we plan to inspire from the work on EAOP (Event-based Aspect-Oriented Programming.pdf) [9] with regards to dynamic aspect composition facilities. Although targeted to behavioral issues, Reflex and EAOP underlying ideas can be adapted to deal with structural changes. First, we definitely retain the idea of a global monitor controlling the application, and the selective introduction of hooks within base applications. As long as structural changes are intra-entity – stay locally inside a certain entity – they are straightforward to allow. If they are inter-entity changes, things will obviously get more complicated as we will have to keep track of the inter-entity dependencies. This is an issue that we will have to investigate further.

In a first version of the framework, we plan to apply a two-layered architecture to allow us to modify the behavior of a running application even when it is already running. For doing that, we instrument the running application with calls to the monitor at every point where communication between entities occurs. The monitor has to keep track of that communication in order to make it possible to substitute a certain entity. For that, it holds a representation of the application. During execution, the monitor passes control to the concerned entities (following the representation), making its presence unnoticeable. This is illustrated in figure 2. When changing a given entity, the monitor will queue all calls to the 'old' entity in order to send them to the 'new' one once in place. Our approach implies that any evolvable entity has to be referenced by the monitor, and that the monitor keeps track of entities and inter-entity relations.

3.3 The runtime API

Finally, in order to evolve the application, the user has to change the application's representation in the monitor. To that extent, a runtime API will be included so that the user can interact on-line

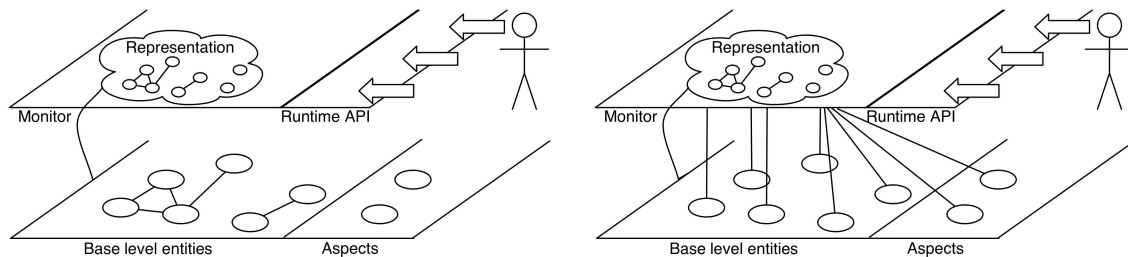


Fig. 2. Runtime evolvability by means of a two layered architecture: inter-entity communications (left) are indirected to the monitor (right).

with the monitor. The functionalities of the API have to include the addition, the removal and the modification of a system entity (aspect or functionality). Adding a new entity is done by writing its code, and by registering it in the monitor. Removing an entity is more complex, as we should make sure that no other entities are dependent of that entity before actually removing it. If that is however the case, the programmer should be warned about that. When a certain entity needs to be modified, we have to write the new entities code, and tell the monitor that it should use the new entity instead of the old one whenever the old one is referenced by an other system entity. In this case, there are also some difficulties that arise, since we should be able to transfer the state from one to another entity. Some formal definition of the before-after behavior should be established in order to avoid conflicts.

4 Validation

The opening perspective of our research is that we will allow an application to evolve dynamically by allowing its composing entities to evolve dynamically. We deliberately do not restrict ourselves to one specific entity, such as a class in the Sun JVM Hotswap API, but will consider entities of any kind. As already mentioned above, such entities can thus be functions, abstract data types, classes, aspects and so on. As this is a very ambitious perspective, we will try to get as close as possible to it, by using a step by step approach.

In a first step, we will consider applications that make use of aspect-oriented technology. Such applications are typically modularized in classes and aspects. First, we will evolve the aspects of such applications. These are the easiest entities we can make evolve, as the base application does not have any reference to those entities. After that, we will focus on the evolution of class entities. This will be a harder challenge as the application has direct knowledge and references to such entities.

After that, we should widen our field of action to other programming paradigms than the object-oriented one as the ultimate goal is to employ the same approach for all existing programming paradigms. Most of the time will be spent on the representation of the application in the monitor. Once that is done, the dynamic evolution capabilities of the program will easily follow.

5 Conclusion

This paper presented a two-step solution for allowing systems to evolve dynamically. In a first step, the application's cross-cutting concerns should be removed, so that it is well modularized. We proposed aspect mining and static refactoring techniques to detect and separate the cross-cutting concerns respectively. In a second step, the well-modularized application should be controlled at the metalevel by a monitor with full reflective capabilities. Such a monitor merged the ideas of EAOP and partial behavioral reflection with the dynamic capabilities of the Smalltalk language. As such, it allows an

application to evolve dynamically. Moreover, such an application can be of any programming style, object-oriented, aspect-oriented or any other, as long as it is well modularized.

References

1. O' Connor, P.: Practical Reliability Engineering. 4th edition edn. Wiley (2002)
2. Dijkstra, E.: The structure of THE multiprogramming system. *Communications of the ACM* 11 (1968) 341–346
3. Tarr, P., Ossher, H., Harrison, W., Stanley M. Sutton, J.: N degrees of separation: multi-dimensional separation of concerns. In: *Proceedings of the 21st international conference on Software engineering*, IEEE Computer Society Press (1999) 107–119
4. Ossher, H., Tarr, P.: Hyper/J: multi-dimensional separation of concerns for java. In: *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland (2000) 734–737
5. Szyperski, C.: *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley (1998)
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: *Proceedings of the European Conference on Object-Oriented Programming*, *Lecture Notes in Computer Science*, Volume 2072., Springer Verlag (2001) 327 – 353 <http://aspectj.org>.
7. Akşit, M., Tekinerdoğan, B.: Aspect-oriented programming using composition filters. In Demeyer, S., Bosch, J., eds.: *Object-Oriented Technology, ECOOP'98 Workshop Reader*, Springer Verlag (1998) 435
8. Popovici, A., Alonso, G., Gross, T.: Just-in-time aspects: efficient dynamic weaving for java. In: *Proceedings of the 2nd international conference on Aspect-oriented software development*, ACM Press (2003) 100–109
9. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In Batory, D., Consel, C., Taha, W., eds.: *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*. Volume 2487 of *Lecture Notes in Computer Science*, Pittsburgh, PA, USA, Springer-Verlag (2002) 173–188
10. Griswold, W.G., Kato, Y., Yuan, J.J.: Aspectbrowser: Tool support for managing dispersed aspects. Technical Report CS1999-0640, University of California, Department of Computer Science and Engineering (2000)
11. Hannemann, J., Kiczales, G.: Overcoming the prevalent decomposition in legacy code. In: *Proceedings of the ICSE Workshop on Advanced Separation of Concerns*, Toronto, Canada (2001)
12. Ettinger, R., Verbaere, M., de Moor, O.: Slicing Based Refactoring in Eclipse. Technical report, Oxford University Computing Laboratory (2004)
13. Breu, S., Krinke, J.: Aspect mining using dynamic analysis. In: *GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft für Informatik*. Volume 23., Bad Honnef, Germany (2003) 21–22
14. Sheperd, D., Gibson, E., Pollock, L.: Automated mining of desirable aspects. Technical Report 4, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716 (2004)
15. Tourwé, T., Mens, K.: Mining Aspectual Views using Formal Concept Analysis. In: *Submitted to Workshop on Source Code Analysis and Manipulation (SCAM)*. (2004)
16. Fowler, M.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley (1999)
17. Hanenberg, S., Oberschulte, C., Unland, R.: Refactoring of aspect-oriented software. In: *Net. ObjectDays 2003*, Erfurt, Germany. (2003)
18. Tanter, E., Noyé, J., Caromel, D., Cointe, P.: Partial behavioral reflection: Spatial and temporal selection of reification. In Crocker, R., Steele, Jr., G.L., eds.: *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2003)*, Anaheim, California, USA, ACM Press (2003) 27–46 *ACM SIGPLAN Notices*, 38(11).