Position Paper for DAW 2004

A Concern-based Approach to Dynamic Software Evolution

Peter Ebraert[1] and Eric Tanter[2,3]

[1] Vrije Universiteit Brussel, PROG Pleinlaan 2, Brussel Belgium
[2] University of Chile, DCC/CWR Avenida Blanco Encalada 2120, Santiago, Chile
[3] Ecole des Mines de Nantes, 4, Rue Alfred Kastler, Nantes, France

Abstract. The problem we are trying to tackle in this research is that of the availability of critical applications while they are being updated. We are investigating how dynamic aspects can be used to evolve some concerns of a running application. Our approach targets a three steps process: first, an application is statically refactored so that evolvable concerns are cleanly separated, second, concerns are handled by a reflective infrastructure, and third, a runtime API to this infrastructure makes it possible to update the separated concerns one at a time.

1 Introduction

An intrinsical property of a successful software application is its need for evolution. In order to keep an application up to date, we continuously need to adapt it. Usually, applications have to be shut down before they can updated, in order to avoid data corruption for example, but mostly because it is generally not possible to update an application at runtime. In some cases, this is beyond the pale, for example in critical systems such as web services, telecommunication switches, banking systems, etc. Unavailability of software systems could have unacceptable financial consequences for the companies and their position in the market.

Currently this problem is being solved using redundant systems [1]. Every critical system is provided with a double that can take over all the functions of the original one, whenever it is not available. This solution has already proved it is working well but it still has some disadvantages. Firstly, the financial side of the story: every piece of hardware and software has to be purchased multiple times. Also, redundant systems have their own problems as the management of the different versions gets harder. Also the maintenance and the switching between the redundant systems are sometimes underestimated.

We are investigating a better, more flexible, solution to this problem, based on the development of applications with separated concerns [2]. In such an application, every adressed concern – crosscutting or not – exists as a separate entity that can be adapted and substituted without affecting the rest of the system. What we call an *entity* in this paper is a part of the program that copes with a certain concern. Depending on which programming paradigm is used, the entity can be of different types. In AOP for example, an entity refers to a set of objects that handles one function or aspect, but an entity can also refer to a component, a set of functions, a set of procedures, a set of abstract data types, etc., depending on the programming paradigm.

If the application is cleanly split up in separate entities, we can say that the evolution of that software system falls down to the removal, the addition or the modification of a certain separated system entity. From the moment that this can be done while the application is running, we can talk about *dynamic software evolution.*

In real life, it turns out that the principle of separated concerns is not always that easy to achieve. Even though some experimental techniques already exist to that matter, they are not exerted all the way through. The major part of today's applications is only a set of strongly woven concerns. Next to that, most existing techniques in separation of concerns are still too static to support dynamic maintenance in real time, because they provide a model in which concerns are fixed in the application [3–6]. This hinders their modification, at execution time. More dynamic techniques are to be investigated to solve that issue. Several prototypes of those techniques do exist [7, 8], but still lack some dynamic properties as well as practical experience.

2 A Concern-based Approach to Dynamic Software Evolution

In order to solve the problems stated above, there are two fundamental issues we have to cope with. On the one hand, we should make sure that the systems match the principle of separated concerns. On the other hand, we should find a technique that allows cleanly developed systems to evolve dynamically.

2.1 General approach

The opening perspective of our research is that we will allow all kinds of entities to evolve dynamically. This is a very ambitious perspective, so we will try to get as close to it as possible by using a step by step approach.

As a start, we will test this approach on the separated aspects of an aspect-oriented application. In such an application, evolvable concerns (aspects) are intrinsically encapsulated out of the base application (this is the obliviousness property). This makes it a lot easier to consider their evolution, compared to evolving pieces of the rest of the base program, which will always be interacting with the other modules.

In a second phase, we want to evolve cleanly separated entities of an ordinary object oriented application which are still well modularized, but not implemented as an aspect. This will be a harder challenge as the application has direct knowledge and references to such modules.

As the ultimate goal is to allow the same approach for functional, procedural, and programming paradigms, we should finally widen our field of action.

2.2 Matching the principle of separated concerns

Most of existing applications do not match with the principle of separated concerns. Nevertheless, they should match with it, if we want to allow them to evolve dynamically. For that, we want to investigate how aspects can be detected in existing applications. Research in that domain, *aspect mining*, only started recently. Although abstract results seem promising, concrete results are not yet available.

We need to investigate if and how we can detect different concerns by observing the dynamic behavior of the application and by deriving which parts are weakly or strongly connected, which communication patterns occur frequently, etc. We plan to use *reflection* – calculations on calculations – in order to do so. In [9] a reflection based runtime monitor is presented, which is able to observe a running application. Extending that monitor with some domain knowledge on communication patterns would already allow the detection of some concerns. The use of a logic metalanguage is also a envisionned possibility.

Once we have identified the different concerns, we need to restructure the application in order to make the concerns explicit. For this, we use refactoring techniques [10], which permit the modification of the internal structure of an application without influencing its semantics. We may need to extend existing refactoring techniques for our purpose.

2.3 From runtime reflection to dynamic evolution

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. Using dynamic structural and behavioral intercession, the application could self-evolve through metalevel manipulations. 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.

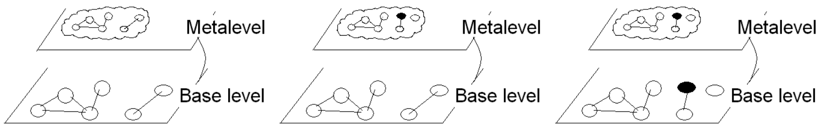

Fig. 1. Dynamically updating an entity through metalevel manipulation.

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 [11] and materialized in the Reflex platform for Java. Finally, we plan to inspire from the work on EAOP (Event-based Aspect-Oriented Programming) [8] 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. During execution, the monitor passes control to the concerned entities, making its presence unnoticeable. 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. This is illustrated in Fig. 2. 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.

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 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 dependant 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

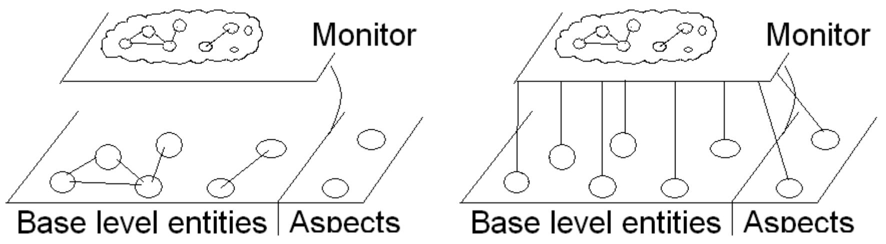

Fig. 2. Runtime evolvability by means of a two layered architecture: inter-entity communications (left) are indirected to the monitor (right).

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.

3 Conclusion

In this paper, we started to investigate the issue of dynamic evolution of applications. We have sketched a three-step process based on cleanly separating evolvable concerns in an application, controlled at the metalevel by a monitor with full reflective capabilities. Such a monitor merges the ideas of EAOP and partial behavioral reflection with the great dynamic capabilities of a language like Smalltalk, to provide dynamic evolution of object-oriented and aspect-oriented applications.

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. 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
4. Szyperski, C.: Component Software : Beyond Object-Oriented Programming. Addison-Wesley (1998)
5. Kiczales, G., Hilsdale, E., Hugunun, J., Kersen, M., Palm, J., Grisworld, 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.
6. 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

7. Popovici, A., Gross, T., Alonso, G.: Dynamic weaving for aspect oriented programming. In: Proceedings of the 1st International Conference on Aspect-Oriented Software Development. (2002)
8. 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
9. Tanter, E., Ebraert, P.: A portable yet flexible approach to interactive runtime inspection. In: ECOOP Workshop on Advancing the State-of-the-Art in Runtime Inspection (ASARTI 2003). (2003) Darmstadt, Germany.
10. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
11. 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 programing, systems, languages, and applications (OOPSLA 2003), Anaheim, California, USA, ACM Press (2003) 27–46 ACM SIGPLAN Notices, 38(11).