Position Paper for RAM-SE 2005

Pitfalls in unanticipated dynamic software evolution

Peter Ebraert[1⋆], Yves Vandewoude[2⋆], Theo D'Hondt[1] and Yolande Berbers[2]

[1] Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussel, Belgium
[2] KULeuven Department of Computer Science, Celestijnenlaan 200A, B-3001 Leuven, Belgium

Abstract. The authors of this paper have all developed a framework that allows runtime adaptation of software systems. Based on our experiences, we wish to summarize common pitfalls concerning dynamic software evolution. Systems for dynamic adaptation typically follow a certain process which is used as a starting point in this paper. The problems that occur in the different steps of this evolution process are given and a suggestion is made on how these problems can be tackled. The reader will notice that the solution to most of the pitfalls lies in the use of reflection, meta-data and meta-object protocols. We conclude that reflection or meta-object protocol manipulations are indispensable in the process of dynamic software evolution.

1 Problem Statement

Lehman [1] defines software evolution as the collection of all programming activities intended to generate a new version from an older and operational version. The problem of software evolution occurs after the initial delivery of the software and typically deals with bugfixes and the addition, change or removal of functionality to the system. Different sources estimate that evolution is responsible for 50% [2] to 90% [3] of the total cost of software. The following quote by Keith Bennet [4] perfectly describes the real difficulty of software evolution: its unanticipated nature. *"The fundamental problem, supported by 40 years of hard experience, is that many changes actually required are those that the original designers cannot even conceive of."* Some systems can not be shut down due to safety or financial reasons. Well known examples are web services, telecommunication switches, banking systems, airport traffic control systems or military systems. Adapting such systems without halting them is a challenging operation that encompasses many different problems. In this short paper, we try to summarize the main difficulties and show that in order to resolve them, both reflection and meta-data are two indispensible tools.

2 Unanticipated dynamic software evolution

In his PhD report, Oriol states three main reasons that make unanticipated dynamic software evolution such a hard undertaking: coping with state transfer, active threads and uncertainty. In this section we see why that is the case by going over the most common process of performing dynamic software evolution. Step by step, we discuss all the phases of this process.

1. **Offline activities.** Before a dynamic update is deployed, it must first be implemented. Offline activities start by locating all structures or entities that are affected by the changes. This problem is often referred to as *dependency finding*. The new code is then implemented according to the renewed specifications of the system. In most cases, the design and source code of the old version are present, since it is likely that new versions are implemented by the owner of the old version. If this is not the case, an attempt must be made to recover these from the software artefacts

⋆ Authors funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen)"

that are available (such as the running system) using *reengineering techniques*. Finally, the correct behaviour of the new version must be verified using either formal proofs or extensive testing. In some cases, the deployment of the new version itself is also tested by deploying it on a duplicate copy of the running system.

2. **Addition of new code to the running system.** The complexity of *introducing new code* into the running system strongly depends on the programming language and environment used. It is easy for languages as SmallTalk or CLOS, harder for statically typed languages as Java or C# (since code can not easily be reloaded), and very hard for languages that are compiled to native code such as C or Assembly.
3. **Deactivation of affected entities.** Dynamic adaptation of an active system entity can result in inconsistencies that may lead the application to an erroneous state. *Program consistency* can be preserved by deactivating all entities or structures within the application that are affected by the change.
4. **Transformation of affected entities.** This phase consists of the actual transformation from the old version to the new version and is composed of transforming behavior and porting state. In class based languages, behavior is captured in method definitions and in the inheritance hierarchy. As such, behavior transformations boil down to class based modifications. The most difficult part of this phase however, is the transformation of runtime state that is contained in variables throughout the system to preserve *state consistency*.
5. **Online verification of new code.** Once the transformation has completed, we wish to *verify* a number of conditions to guarantuee its correctness. This is achieved by evaluating a number of conditions and invariants on the new code version. If these checks fail, a *rollback* mechanism must make sure that the previous state is restored.
6. **(Re)activation of the halted entities.** The last step consists of reactivating all the entities that were deactivated earlier in the process.

Relating to the issues identified by oriol, steps 3 and 6 are present to cope with active threads, step 4 deals with state transfer and the tests in steps 1 and 5 are present to lower incertainty. In the next section, we discuss common pitfalls that occur in this process and suggest possible solutions.

3 Pitfalls

3.1 Dependency finding and reengineering

In order to implement a new version of a software system, it is crucial to obtain its architecture and design if not already present. This information is required for the identification of all the entities that have some relation with the evolving part of the software. In addition, we need it for providing new source code that will fit in the existing system.

Recovering the design of an application has to be done by using both static and dynamic information. Static information describes the structure of the software as it is written in the source code, while dynamic information describes what is really happening at runtime. It can be perfectly possible that structural information shows that two classes are just a bit related, as there is only one method call from one class to the other. However, it is possible that dynamic information shows that the same call occurs continuously when running the application, making the two classes very related. This explains why both dynamic and static analysis result in more realistic design recovery.

Static information can be obtained by looking at the application's implementation. Practically this can be done in two different ways; by looking at the source code or by using introspection (= reflective capabilities of observing the application).

Dynamic information can be gathered by monitoring the behavior of the running application. This is typically done by using a layered approach. Implementations of this approach include the adaptation of the metaobject protocol in such a way that all baselevel executions are intercepted and monitored. Another implementation consists of the instrumentation of base-level entities with calls to a monitor by means of intercession (= reflective capabilities of modifying the application).

3.2 Introduction of new code.

Whereas safely introducing new code to a running system is a non-trivial accomplishment by itself for languages such as C or assembly [5, 6], modern languages such as Java, Smalltalk or C# allow a programmer to add new code in a safe and extremely convenient manner.

However, the ability to add new code to a running system is by itself not sufficient for dynamic adaptation. Unanticipated software evolution almost always includes modifications to existing code. Although this is not a problem for purely dynamically typed languages such as Smalltalk, it is a much harder problem for statically typed languages such as Java or C# (for a extensive discussion on the influence of a programming language and its type system on runtime evolution we refer to [7]). For statically typed languages, unloading or modifying code which is already loaded is prohibited due to safety restrictions that are enforced by the language model. For instance, Java ensures that all methods or fields that are used also exist. Such a guarantee can not be given if class definitions can change at runtime unless extensive and frequent runtime type checks are added which would degrade runtime performance significantly. This is a sacrifice designers of statically typed languages are not prepared to make.

In Java, the problem is partially circumvented using the classloader mechanism. Since classes loaded by different classloaders are considered to be distinct types, the classloader architecture also allows different versions of a class to be used simultaneously. Such techniques are used by component runtime environments to (un)load different components independently. While this approach is sufficient for loading modified code into the system, it causes some important problems related to dynamic adaptation: an object of version $n + 1$ can not be assigned to a variable of version n. This problem is called the version barrier [8], and is especially relevant for state transfer between different versions of an application. As we will see in the section 3.4, reflection and meta-data will play a vital role in solving these problems.

3.3 Program consistency and deactivation.

It is clear that for a dynamic update to succeed, arbitrary changes to the software can not be allowed. For example, online software replacement may not be feasible if the new version of the program is an entirely different program. It is vital that a runtime change preserves program consistency. An informal definition of a consistent application state is a state from which the program continues execution in a correct manner rather than progressing towards an error state. An application can be seen as moving from one consistent state to the next. Since state is distributed throughout the system, different state structures can be temporarily inconsistent with oneanother. It is vital that the application is in a consistent state before runtime modifications are performed. Kramer and Magee introduced the concept of quiescence in [9]. While their work was originally in the field of distributed systems consisting of a set of distinct nodes, it can be applied to dynamic adaptations of object-oriented or component-oriented applications as well.

In order to ensure that no communication or method calls are active during the modification of a certain entity, the entity must be deactivated. Any deactivated entity will queue all incoming transaction request and postpone their execution until the entity is reactivated. Different implementations are possible to achieve this goal. In [10], a wrapper based approach is used. A wrapper is added to each system entity that adds additional functionality for activation or deactivation. *Futures* are returned to the caller as a return value. These futures will be resolved when the entity is reactivated. Messages to futures result in futures themselves. This chain of futures continues until a side-effect occurs, after which the application is halted until the entities are reactivated. In [11], communication between components is asynchronous and realised by sending messages through *connectors* that are capable of queuing messages until the component is reactivated. Since there are no return values, the concept of futures is not required.

In the end, the implementation of a deactivation scheme strongly depends on reflection: communication is reified into messages that can be queued until further notice. The advantage of reflection is that it allows the addition deactivation logic without modifying the components themselves.

3.4 State transfer and consistency.

Although deactivation is essential for dynamic software adaptation, it is not sufficient by itself. True dynamic evolution requires that the state from the previous version is imported in the new version of the software. The assumption of quiescence ensures that all state is contained in the instance variables of the different objects that make up the application or component (assuming an object oriented paradigm). Two possible approaches exist to achieve state transfer between different versions:

Indirect: The old version exports its state in some *abstract form* which is later interpreted by the new version. In some cases, the exported state can be written to disk.
Direct: The new version *directly* interprets the state from the old version.

Although the first version seems more convenient at first, it has some major disadvantages. First of all, in order for the exported state to be in an *abstract* form, a generally accepted ontology must exist so that all parties can agree on the semantics of this abstract state. Such an ontology only exists for certain domains, severely limiting the practical approach of this technique. In addition, this requires that each entity implements a state export function, even if it may never be used. This lays a huge additional burdon on the programmer. The second technique does not suffer from these restrictions. State is extracted using either getters/setters, or, more likely, using reflection.

For statically typed languages, the presence of different application domains or classloaders (see section 3.2) further complicates the adaptation, since communication between different versions is strictly limited to known common types (eliminating the ability to extract state using getter-methods and increasing the dependency on reflection). The presence of such an architecture also results in a *cascading effect of changes*, which eliminates the possibility of preserving large (unchanged) portions of the application or component. Indeed, a type A which has not changed by itself, but that contains a reference to a changed type B will not be able to use the new version of B due to the version barrier. Transforming objects of A to refer to the new version of B causes a cascading effect, since all types referring to A would require changes as well. A solution to this problem was proposed in [8], in which Sato and Chiba introduce Negligent Classloaders, which relax the version barrier under certain circomstances. An alternative technique would be to change the classloader of unmodified types from the old version to the classloader of the new version, allowing them to be integrated in the object tree of the new version. Both solutions require virtual machine adaptations.

It is unlikely that a generic solution can be developped without strongly depending on both reflection and meta-data. Regardless of how the actual state transition logic is generated, transferring state between different versions requires information which is not always present in the sources of the different versions, and therefore would need to be added using meta-data. Both the extraction of the state from the old version, and its insertion in the new version require reflective operations (the adaptation is unanticipated and its likely that the running version does not have the required extraction functionality).

3.5 Verification and rollback.

Deepak Gupta has proven ([12]) that full automatic verification of the correctness of an update is computationally undecideable. Therefore, the designer of the update must include a number of checks with the new version. These checks can either be executed before the transition, verifying that certain unwanted states are not present, or after, to insure that the update was indeed succesfull. An example of the a pre-condition that is commonly used for dynamic adaptations is ensuring that a component is not involved in a transaction [13, 9]. As long as the precondition is not satisfied, the update is delayed.

After the update, additional sanity checks can be executed on new version before it is reactivated. Postconditions are also commonly used to verify the result of dynamic aspect weaving (for example, in [14] the authors verify their aspect compositions using postconditions). Reflective mechanisms are necessary, not only to extract these conditions from the new version, but also to evaluate them. Indeed, it is likely that these conditions will use state from the new version that can not be accessed without reflection.

If one of the tests fails, a rollback is required to restore the original system. Following the process that was described in section 2, the rollback will only have to be applied on deactivated entities. This is achieved by maintaining of a copy of the original entities that are to be restored during rollback. This copy can be retrieved using introspection and restored using intercession.

4 Conclusion

We start this paper by giving an overview of common difficulties that come with dynamic software evolution. In particular these are related to state transfer, active threads and uncertainty. A typical process of dynamic software evolution is then presented that copes with these difficulties. However, in this process a number of pitfalls show up. An explanation is given for each of these pitfalls and a conceptual solution is suggested. Our conclusions confirm that both reflection and meta-object protocol manipulations are indispensable in the field of dynamic software maintenance.

References

1. Lehman, M., Ramil, J.: Towards a theory of software evolution - and its practical impact. Invited Lecture, Proc. Intl. Symp. on Principles of Software Evolution (2000) 2–11
2. Lientz, B., Swanson, E.: Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations. Addison-Wesley (1980)
3. Erlikh, L.: Leveraging legacy system dollars for e-business. IEEE IT Pro (2000) 17–23
4. Bennet, K.H., Rajlich, V.: Software maintenance and evolution: A roadmap. Future of Software Engineering. (2000)
5. Segal, M.E., Frieder, O.: On-the-fly program modification: Systems for a dynamic updating. IEEE Software **10** (1993) 53–65
6. Hicks, M.: Dynamic Software Updating. PhD thesis, Department of Computer and Information Science, University of Pennsylvania (2001)
7. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Influence of type systems on dynamic software evolution. Technical Report CW410, KULeuven, Belgium (2005)
8. Sato, Y., Chiba, S.: Negligent class loaders for software evolution. In Cazzola, W., Chiba, S., Saake, G., eds.: ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04), Oslo, Norway, Fakultät für Informatik, Universität Magdeburg (2004)
9. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering **16** (1990) 1293–1306
10. Ebraert, P., Mens, T., D'Hondt, T.: Enabling dynamic software evolution through automatic refactorings. In: Proceedings of the Workshop on Software Evolution Transformations (SET2004), Delft, Netherlands (2004)
11. Vandewoude, Y., Rigole, P., Urting, D., Berbers, Y.: Draco : An adaptive runtime environment for components. Technical Report CW372, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (2003)
12. Gupta, D.: On-line Software Version Change. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur (1994)
13. Janssens, N., Michiels, S., Mahieu, T., Verbaeten, P.: Towards Hot-Swappable System Software: The DIPS/CuPS Component Framework. In: Proceedings of the Seventh International Workshop on Component-Oriented Programming, Malaga, Spain (2002)
14. Klaeren, H., Pulvermüller, E., Rashid, A., Speck, A.: Aspect composition applying the design by contract principle. In: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering, Erfurt, Germany (2000) 57–69