Enabling Dynamic Software Evolution through Automatic Refactoring

Peter Ebraert
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
Email: pebraert@vub.ac.be

Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
Email: tjdhondt@vub.ac.be

Tom Mens
University of Mons-Hainaut
Avenue du Champ de Mars 6
B-7000 Mons, Belgium
Email:tom.mens@umh.ac.be

***Abstract*— Many software systems must always stay operational, and cannot be shutdown in order to adapt them to new requirements. For such systems, dynamic software evolution techniques are needed. In this paper we show how we can exploit automated refactorings to improve a software the component structure of a software system while the system is running in order to facilitate future evolutions. We report on some experiments we performed in Smalltalk to achieve this goal.**

I. Introduction

People always say that you should never change a system that is working fine. However, even if a software system seems to work properly from a user's point of view, it may be difficult to maintain or adapt from a developer's point of view. As such, it may be very cumbersome to evolve the system by adding a new feature, fixing a bug or porting the system to a new environment.

In all these situations where a software system is not flexible enough to allow for a certain change, the technique of *software refactoring* can be used. According to Fowler [1], a refactoring is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour." Refactorings can be used to simplify the structure of a software system in order to prepare it for a certain evolution step.

Now suppose we have a running system, and we would like to evolve it without shutting it down. This is a much bigger challenge since there are considerably more constraints on the running system. Refactoring techniques would be very useful here too. For example, by reducing the coupling between components in a running system, we could at the same time increase the system performance (from a user point of view) and its understandibility and evolvability (from a developer point of view).

Until now, refactorings have only been investigated in the context of source code restructuring. The main contribution of this paper is to show the use and feasibility of applying *dynamic refactorings*, i.e., refactorings that modify a running system.

II. Experimental Setup

In order to be able to apply refactorings to a running system, we need to be able to dynamically modify the structure and behavior of the software that is being executed. To this extent we need a software system that has full reflective capabilities. According to Pattie Maes [2], reflection is the ability of a program to manipulate as data, something that is representing the state of the program during its own execution.

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.

Because the focus of this paper is on refactorings, we restrict ourselves to class-based object-oriented languages. Object-oriented languages like Java are excluded because of the limitations of their reflective capabilities. Smalltalk, on the other hand, is fully reflective: everything is an object, and can thus be taken apart, queried for information and possibly be modified. Even messages are objects, and can thus be monitored and modified when they are sent or received [3].

III. Proposed solution

A. Terminology

In order to provide a sound basis for starting a discussion on how we plan to do dynamic evolution, we first need to establish a common vocabulary that will be used throughout this paper.

A **software system** is assumed to consist of a set of processing components with directed connections indicating the communication that occurs between the components.

A **component** is a processing entity that can request and provide services. In class-based OO languages, components typically consist of interrelated classes, which are communicating through message passing.

A **connection** is a directed communication from one component - the *initiator* of the communication to another component - the *recipient*. This connection contains all important messages sent between the classes contained in the initiator component and the recipient component. Connections indicate

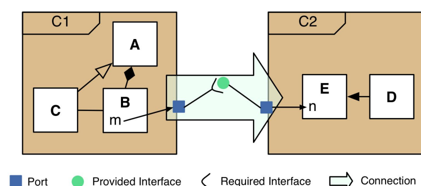

Fig. 1. A base level application architecture

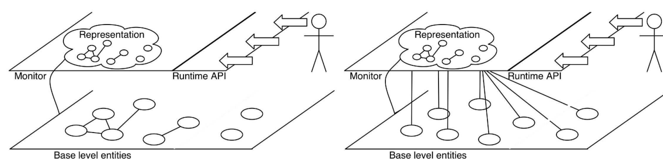

Fig. 2. Runtime evolvability by means of a two layered architecture: inter-component communications (left) are indirected to the monitor (right).

that the component might initiate transactions.

A **transaction** is an exchange of information, by means of message sends, between two and only two components, initiated by one of the components. Transactions are the means by which the state of a component is affected by other connected components in the system. Transactions consist of a sequence of one or more message exchanges between the two connected components. It is assumed that transactions complete in bounded time and that the initiator of a transaction is aware of its completion. The completion of transactions at the initiator is required to ensure correct termination of the change management protocol that will be described later.

Figure 1 introduces the notation that we use to show what the structure of a base-level software system looks like. In this notation, the system is represented as a directed graph of components and connections. The big edge in the directed graph represent a connection between two components.

B. Detecting the dynamic component structure

Most of the time, we do not know the dynamic component structure of an application described as a directed graph of nodes (components) and edges (connections), but instead we only have the source code of the application. The dynamic component structure reflects the way objects need to be grouped into components based on their communication patterns. To increase cohesion and reduce coupling between components, objects that exchange many messages between one another should be clustered into the same component.

In order to detect which objects can be clustered together, static source code analysis does not suffice. First of all, static analysis only provides an approximation of the possible runtime behaviour. Second, since we reason about object-oriented code, we need to take into account dynamic information such as late binding and polymorphism. Third, since Smalltalk is a dynamically typed programming language, we need to do at least some amount of dynamic type inferencing.

Therefore, we decided to resort to a dynamic analysis of the program. We use statistical information that we obtain by monitoring execution traces of the application at runtime. This can be achieved by using a two-level architecture. While the application is running normally at the base level, a monitoring application will be located at the metalevel. Such applications already exist [4], but do not meet all our needs. That is why we developed our own monitoring application that collects information on running applications. That information is used to cluster classes that are very related together into components. The messages that go from a class in a certain cluster to a class in another cluster, will be captured in connections.

C. The dynamic evolution framework

In [5], [6] we presented a first version of an evolution framework [7]. It shows how we apply a two-layered architecture to allow the modification of the behavior of running applications. For doing so, we instrument the base-level application with calls to a metalevel monitor at every point where communication between components occurs. During execution, the monitor passes control to the concerned components (following the metalevel representation of the application), making its presence unnoticeable. This is illustrated in figure 2.

In order to evolve the application, the user has to change the application's representation in the monitor. To that extent, a runtime API is included so that on-line interaction with the monitor becomes possible. The functionalities of the API include the addition, the removal and the modification of system components, and their relations. Each of these operations have their pre-conditions, which have to hold before the operation is actually carried out by the framework [8]. When modifying an existing component, problems could rise concerning state consistency [9].

D. Dynamic refactoring and evolution

1) Detecting possible refactorings: In order to decide which part of the code needs to be refactored to facilitate future evolutions, we use the information obtained from the monitoring application to verify at regular time intervals whether the current component structure should be modified. It this happens to be the case, the proper refactorings will be suggested to be applied with our without user interaction.

2) Change Management Protocol: An evolution or refactoring typically replaces a set of components C_1, C_2, C_3, ... by their new versions C'_1, C'_2, C'_3,... We use the notation ΔC_i to denote the difference between C_i and C'_i. In this section, we first define a set of atomic steps - *change transactions* - that can be used to compute ΔC in an automated way.

In object-oriented programs, components typically consist of some related objects, that on their turn consist of instance variables and methods. Most of the relations between the objects are caught in the methods and instance variables. This is why our meta-object protocol currently implements the following set of atomic change transactions. (In the future we intend to extend this set to cover a more realistic set of applications.)

chaName Changes the name of a class.

addMethod Adds a method to a class. As a result, all

objects that are instance of this class will automatically understand this new method thanks to Smalltalk's method lookup mechanism.

remMethod Removes a method from a class. As a result, all instances of this class may no longer understand this method. Hence, one should be very careful with this operation as it can give rise to runtime exceptions. We will explain how to deal with this later in this section.

chaMethod Modifies the implementation of a method in a class. Again, this will have an impact on all objects that are instance of this class or one of its subclasses.

addInstVar Adds an instance variable to a class. As a result, all objects that are instance of this class have a new variable they can use to store values. By default, the value will be set to `nil`.

remInstVar Remove an instance variable from a class.

chaSuper Changes the parent of a class. This will change the inheritance hierarchy, and as a result the methods that any object of this class, or of any of its children, can understand.

deactivate Deactivates a component – all the classes and instances it contains – to make sure that no transaction will occur in the component. When a component is deactivated, all transactions are put in a waiting list.

activate Allows the component – all the classes and instances it contains – to resume its execution. All waiting transactions will be processed upon activation.

By monitoring the users' actions when evolving on offline copy of C to C', we can automatically obtain ΔC. Afterwards, the operations that need to be performed, in order to comply with possible preconditions of certain actions, are inserted in order to obtain the *change transaction sequence*; the sequence of all atomic change steps. That sequence is then used to evolve the online C to C'.

The main benefits of this approach are the preservation of the state and object identity, as we will keep on working on the same (already existing) component C. Replacing an entity C would involve the creation of C', the swapping of all relations from C to C', the deletion of C and the mapping of the state from C to C'. Evolving the existing C component to C' only involves the creation of C' and the propagation of the changes on C. This implies that there will be no more relation swapping problems and less state mapping problems.

A second benefit is the possibility to test the new version of the component offline. In order to validate C', we need to perform some tests. First we need to perform *component testing* (testing the internal behavior of the component), and then we need to do *system testing* (testing the external behavior of the component - its relation with the other components).

Every one of the atomic change transactions has some specific requirements that need to comply before the transaction can be carried out. For instance, we cannot remove a certain method m that is still called somewhere. So before removing m, the methods in which m is called, must be modified so that they do not call m anymore. For including these preconditions in the rules, we have to introduce states. A method or instance variable is in a *removable* state if it is not called anymore. For modifying methods or instance variables, we need to make sure that the components that use them, are deactivated, so that no inconsistencies are introduced while changing the component. So we can say that instance variables or methods are in a *modifiable* state, if they cannot be called while the evolution is occurring. Adding a method or instance variable does not have any precondition, so we say that the precondition for adding always holds true.

When the backtracking algorithm finishes, it outputs the best set of atomic transactions that can be followed in order to bring C to C'. Note that it is guaranteed that a valid path will be found as C and C' are both composed of objects that are in their turn composed of methods and instance variables. In the worst case, all methods and instance variables of all objects in C can be removed, and the methods and instance variables of all objects in C' then added. From the moment the best path is found, the change transaction set is established.

3) Managing consistency: Informally, we can say that a consistent application state is a state from which the system can continue processing normally rather than progressing towards an error state. A system is viewed as moving from one consistent state to the next, as the transaction processing continues. In fact, application transactions modify the state of the application, and, while in progress, have transient state distributed in the system. While transactions are in progress, the internal states of nodes may be mutually inconsistent. So we should also avoid the loss of application transactions and make sure that we achieve a consistent state after carrying through a change. This consistent state requires that there is no communication in progress between the affected components nor with their environment.

For making sure that no communication is occurring while a certain component is being modified, we introduce the concept op *deactivated* components. Such components will queue all incoming transaction requests, and carry them out whenever they get *activated*. The evolution framework that we proposed in section III-C, offers the functionality of activating and deactivating system components. Note that the current implementation goes out from an asynchronous messaging system, and that transactions that are queued, will not proceed. Such an approach will make sure that consistency is preserved, but will also make the entire application stop, unless it is developed in a distributed way (with multiple threads). In class-based systems, deactivating a component means that we will deactivate all its class and all living instances of that class and bring them in a modifiable state.

4) Carrying out refactorings and evolutions: From the moment we have the change transaction sequence, we can start implementing these changes on the running system. In this phase, the atomic changes of the transaction set will be implemented in the system one by one. While most of them can be done transparently, some may require the programmer's interference. This is the case when there is a state involved, that needs to be preserved. Concretely, when an instance variable is deleted or modified, the question arises what has to

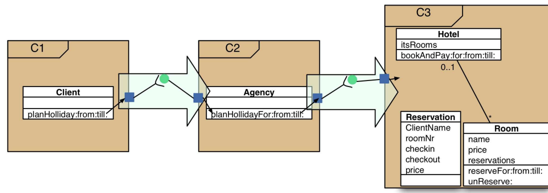

Fig. 3. The original architecture of the travel example; providing 1 interface.

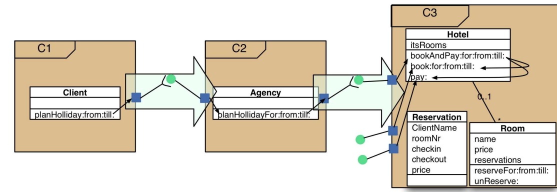

Fig. 4. The adapted architecture of the travel example; providing 3 interfaces.

happen with the value of that instance variable. Either the value can be ignored, or its is needed later in a new instance variable that will be added. Consequently, when an instance variable is added, the programmer is also requested to interfere, and to tell wether the variable should be initialized with a certain value. For example, using Euros instead of Belgian Francs in our bank accounts requires us to use the following formula: 'take the old value and multiply it by 40,3399, and use it as the new value'.

For methods in class-based systems, things are much simpler. Because methods are only referenced through the class itself, adapting them on the class level does the job.

IV. Example

We validate our approach by working out the simple example that we shortly introduced above. In that example we have a software system with clients, travel agencies and hotels, as the basic components (figure 3). When a certain client wants to go on a holiday, he contacts an agency he knows. That agency can then book a room in a hotel, by using the `bookAndPay : for : from : till :` interface that is provided by the Hotel component. This service makes sure a room is booked and payed for.

Imagine that, at a certain time, we might want to allow reservations that are not payed at the same time than making reservations. For adding that functionality to the system, we need to split up the `bookAndPay : for : from : till :` service in two different services: `book` and `pay`. For not affecting existing agencies, that still might want to use the `bookAndPay : for : from : till :` service, that service must still be provided by the Hotel component. For not having code duplication, that service will invoke the book and pay service in its time, as we can see in figure 4.

Place	Atomic change	Parameters
C3	deactivate	
C3$>Hotel$	chaMethod	"bookAndPay:for:from:till:"[meth.Body]
C3$>Hotel$	addMethod	"book:for:from:till" [meth.Body]
C3$>Hotel$	addMethod	"pay:" [met.Body]
C3	activate	

TABLE I

The atomic change set for method extraction

Table I shows the atomic changes that are needed for performing this evolution. This specification is passed to the Evolution Framework, that handles all those steps, and thus carries out the evolution. As, in this case, there is no state modification involved, the user won't have to interact with the system anymore and the evolution gets carried out totally automatic. The resulting architecture can be seen in figure 4.

V. Conclusion

In some cases, software systems can not be turned off for carrying out an evolution. This triggers the need for a framework that supports dynamic evolution. We suggested to apply dynamic refactorings to improve the runtime component structure of object-oriented software systems. The approach relies on the reflective properties of the underlying programming language in order to modify the application's behavior.

Our framework uses a monitor at meta-level that keeps track of the base-level application. This monitor detects the dynamic architecture of the application by collecting statistical information about the messages sent between objects. This information is used to determine the dynamic software architecture in terms of components and connections. Whenever this structure changes, dynamic refactorings can be triggered to evolve the component structure at runtime.

Currently, we are in the process of implementing the above framework in Smalltalk. The framework allows the addition, removal, or modification of base-level components while the system is running. The framework provides facilities to establish atomic change sequences, manage consistency, preserve object identity and state, and apply dynamic refactorings and evolutions.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] P. Maes, "Computational reflection," Ph.D. dissertation, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[3] S. L. Messick and K. L. Beck, "Active variables in smalltalk-80," in *Technical Report CR-85-09*. Computer Research Lab, Tektronix, 1985.

[4] R. Wuyts, "Smallbrother - the big brother for smalltalk," Université Libre de Bruxelles, Tech. Rep., 2000. [Online]. Available: http://homepages.ulb.ac.be/~rowuyts/SmallBrother/index.html

[5] P. Ebraert and E. Tanter, "A concern-based approach to dynamic software evolution," in *The proceedings of the Dynamic Aspects Workshop in conjunction with AOSD*, Lancaster, UK, march 2004.

[6] P. Ebraert and T. Tourwe, "A reflective approach to dynamic software evolution," in *The proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution in conjunction with ECOOP*, Oslo, Norway, June 2004.

[7] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *IEEE Computer Society*, vol. 37, no. 10, pp. 46–54, october 2004.

[8] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, November 1990.

[9] Y. Vandewoude and Y. Berbers, "Fresco: Flexible and reliable evolution system for components," in *Electronic Notes in Theoretical Computer Science*, 2004.