User-centric dynamic evolution

Peter Ebraert* Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussel, Belgium
{pebraert, tjdhondt}@vub.ac.be

Yves Vandewoude*, Yolande Berbers
Department of Computer Science
KULeuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
{yvesv, yolande}@cs.kuleuven.ac.be

Abstract

The domain in which we situate this research is that of the availability of critical applications while they are being updated. In this domain, the attempt is to make sure that critical applications remain active while they are updated. For doing that, only those system parts which are affected by the update, will be made temporarily unavailable. The problem is that current approaches are not user-centric, and consequently, that they cannot provide feedback concerning which features are deactivated while performing an update.

Our approach targets a four step impact analysis that allows correct feedback to be given while a software system is dynamically updated. First, the different features are identified. Second, the system entities that implement those features are identified. Third, an atomic change sequence is established for the actual update. Finally, we compare the atomic change sequence and the implementation of the features in order to establish a list of features that are affected by the update. This allows us to provide user-centric feedback in terms of features.

Keywords: dynamic software evolution, change impact analysis, features

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.

A possible solution to this problem are redundant systems [8]. 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 works in practice, 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.

Another approach to this problem is dynamic adaptation of the system. This involves adapting the system while it is active, but requires that the parts of the system – which are affected by the update – to be deactivated while the update is performed [6]. Existing systems (such as [9, 2]), operate on an abstraction level of programming constructs (e.g. components, objects or methods). Working on this level of abstraction has the benefit of easily identifying the affected system parts, as updates will be executed on the same level of abstractness. However, it has the inconvenience of not being user-centric, bringing along the difficulty of providing useful feedback to the user – e.g. which functionalities of the system will not be usable during the update.

We propose to lift the level of abstraction towards system features. We adopt the definition of Eisenbarth et al. for features [3]. They define a feature as "an observable unit of behavior of a system which can be triggered by the user". We reason about features, but maintain a link between the features and their underlying program constructs. This allows us to benefit from the two layers of abstraction. On the one

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

hand, on the programming constructs level, we can easily identify the affected system parts. On the other hand, on the feature level, we can identify the affected functionalities of the system. This opens up possibilities of providing useful feedback to both the system user and developer. At runtime, the user can be warned about temporary offline system features. At compile time, the developer can be warned about the features that will be affected by the update, allowing him to react in case some features were not suposed to be affected by the update.

2 General approach

In this research report, we look at evolving applications at the level of system features. More precisely, we want to be able to comment which features are affected by a certain update. For doing that, we first need to identify the different features the application provides. This can be done by automatic feature extraction techniques [5] or by manual code annotations.

In the second step we capture the system entities that are implementing those features. This is currently done by using static design knowledge of the system and by producing a *call graph*: a graph of the execution trace consisting of nodes (code statements) and edges (method lookups). Those call-graphs represent the link between the two levels of abstraction (the program concepts level and the system feature level).

In the third step, the application update is rewritten as a *change sequence*: a sequence of atomic changes. A change sequence captures all source code modifications that are amenable to analysis. In [2] we explained how monitoring techniques can be used to establish the atomic change sequence.

In the fourth step, a change impact analysis [10] is performed. It takes two major inputs: the call graphs from the different application features and the change sequence that was established in the previous step. The basic idea of the analysis is to compare the atomic changes with the call graphs in order to establish the *affected features list*: a list of features that are affected by the update. Note that – thanks to the first and second steps of the process – we also know the *affected entities list*: the system entities that are implementing the affected features.

From the moment that all the affected features – and the program concepts that represent those features – are captured, we can start the actual update process. In order to avoid corruption, we want to make sure that the affected system entities are in a quiscent state before they are updated [6]. Entities that are in a quiscent state do not allow incoming messages, and thus make sure that the entities state remains consistent. Thanks to the impact analysis, we know exactly which entities we should deactivate in order to avoid corruption. After deactivating the affected entities, we execute the change sequence and reactivate the affected entities, making sure the update is carried out in a safe way. Next to that, we are able to give feedback to both the user and the developer on which features will be affected by the update. The developer can be warned at compile time on which features a certain change will impact. The user can be warned at runtime on which features are temporarily unavailable. Note that this approach acts to the basic idea of dynamic updating; carrying out updates in a safe way, without shutting down the entire system.

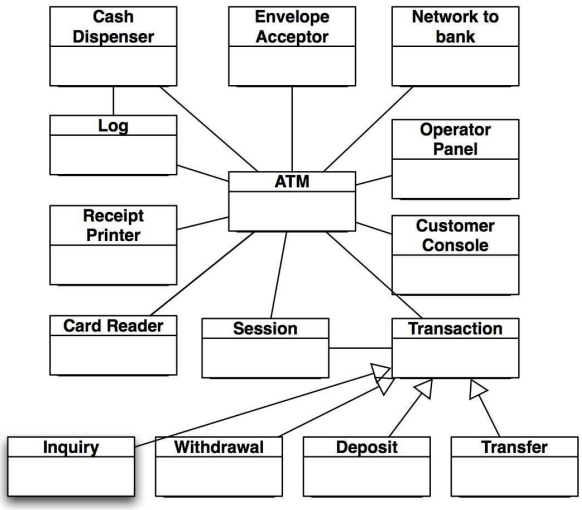

Figure 1. Class diagram of the ATM application

In order to exemplify our approach we use the example of a class-based implementation of an ATM machine. Figure 1 shows the class-diagram of the system. We see that there is a central class called ATM, which is the link between all the classes of the system. The system contains five features: logging in, making a money transfer, making a cash withdrawal, making a cash deposit and consulting the balance. While the final four features are all transactions, the first feature consists of a non-functional feature: ”the user has to be logged in before he can start one of the transactions”. Throughout this paper, we will continuously refer to this example to clarify every step of the approach.

3 Change impact analysis

In this section, we describe the four step process of detecting the impact a certain change has on the system features. The outcome is a set of features (and corresponding system entities) that will be affected by a certain update. Before we actually start the change impact analysis,

we must say that we go out from the premise that an application consists of features, and that those features can be identified in the system. The remainder of this section consists of a step-by-step description of this approach.

3.1 Identifying feature

The goal of this phase is to capture all the different features of the system. Currently, we use design information (from UML use case diagrams) for identifying the different system features. However, recent research on feature extraction techniques [5, 1, 4] has shown that automating this step is feasible.

In practice, we plan to use Starbrowser [13] to model the different features of the system. Starbrowser is a generic classification system for Smalltalk that allows the user to add, modify, delete, view and browse classifications of source code entities. We model each feature as a classifiable entity and make sure that, by clicking a feature, its call-graph is shown to the user. The way this call-graph is established, is explained in the following subsection.

ID	Feature	Explanation
F_1	Logging in	The user identification process
F_2	Money transfer	Transfer money from this account to an other one
F_3	Cash withdrawal	Withdraw cash money from this account
F_4	Cash deposit	Deposit money on the account
F_5	Balace consulting	Consult the balance of the account

Table 1. System Features

Table 1 shows the five features that we identified in the ATM case. In the rest of the paper, we are considering those four features.

3.2 Linking features with system entities

As explained in section, 1, we want to keep a link between the program construct level and the system feature level. In order to do that, we need to find the relationship between features and system entities. This is done by analysing the features and capturing their execution traces. In [7, 1, 4, 5], some techniques of dynamic analysis are presented, that capture the actual execution trace of the features. Because it is very hard to predict the actual execution trace that is used by the features, a conservative superset of the execution trace is captured.

We model each execution trace as a call-graph; a graph in which nodes represent code statements and edges represent method calls. Nodes are labeled with a $< C, M >$ tuple, where C is the class id and M the method which is called. Edges corresponding to dynamic dispatch are also labeled with a $< C, M >$ tuple, where C is the run-time type of the receiver object, and M is the method. Note that we are currently not taking into account changes to the instance variables.

In the case of the ATM example, we did not do the complex feature analysis for obtaining the call-graphs. In stead, we use design information (from collaboration and sequence diagrams) for constructing the call-graphs. We are aware that this results in a probably incomplete call-graph. This is not a problem, as the goal of the example is to explain the user-centric approach and not the details of the call-graph mining.

In practice, we could use monitoring techniques [14] for obtaining the execution trace of the features. The execution traces can directly be modeled as call-graphs, which are in there turn stored in the Starbrowser [13]. Figure 2 shows the call-graphs of all the features of the ATM example. We can see that all transaction features have a commun part in their call-graph. However we see that in that commun part, there is a slight difference concerning the runtime type which is passed when calling $complete()$ on $Transaction$. This is due to the fact that there is a dynamic dispatch in that place.

3.3 Establishing the atomic change sequence

In order to be able to start a change analysis, we first need to decompose the update that brings the system from version S to version S', and capture it into a δS (= a change sequence [2]). This change sequence captures all modifications to the source code in a list of atomic changes. We extended the model that was presented in [2] with a few extra atomic changes which capture variable modifications and method lookup [10].

Scope	Atomic change	Explanation
Class	AC	Add a Class
	DC	Delete a Class
Variable	AV	Add a Variable
	DV	Delete a Variable
Method	AM	Add a Method
	DM	Delete a Method
	CM	Change the body of a Method
	ML	Change the Method Lookup

Table 2. Atomic Changes

Table 2 summarizes the set of atomic changes. The first and most simple atomic changes incorporate added classes (AC), deleted empty classes (DC), added variables (AV), deleted variables (DV), added methods (AM), deleted methods (DM) and changed method bodies (CM). The last kind of change consists of changes in the method lookup (ML). Note that a change to a method body is captured by only one CM, even if it consists of many changes.

In practice, the developer has to use a tool which is appropriate for doing updates to the system. This tool allows all the atomic changes that were shown in table 2 and mon-

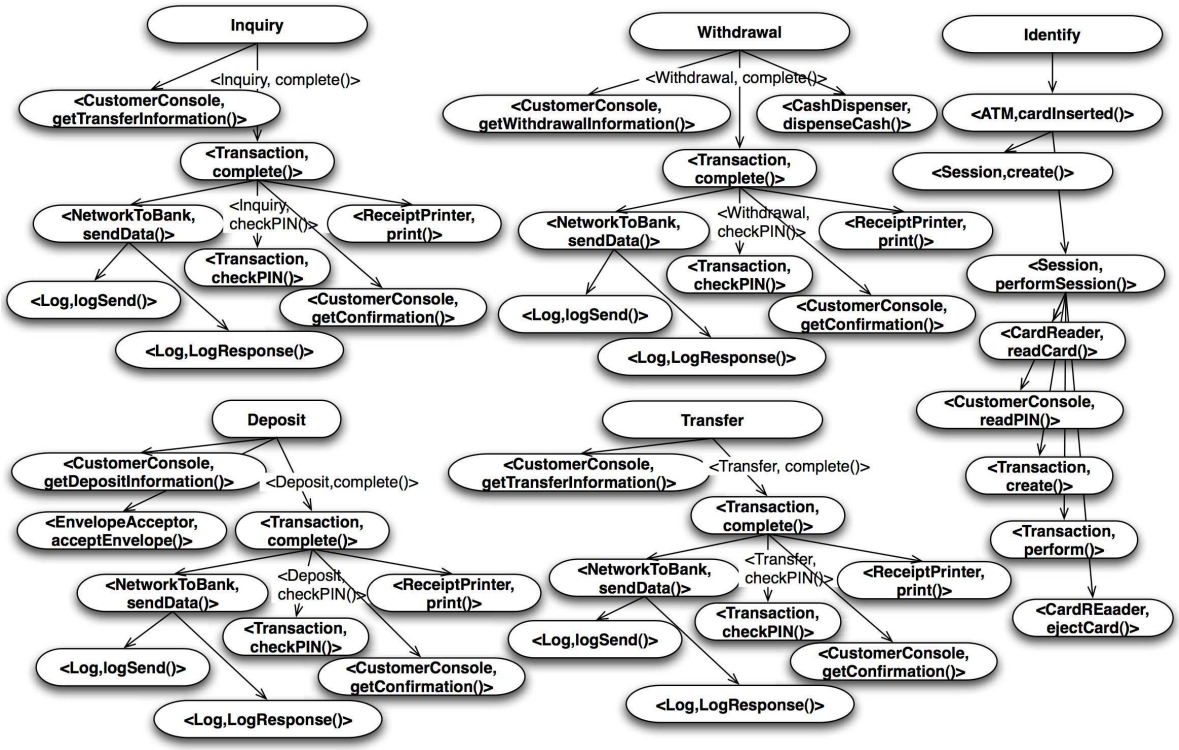

Figure 2. Call graphs of the system features

itors the changes the developer is applying. Those changes are captured and stored in the atomic change sequence.

Imagine that the gouvernement issues new bank notes of 1000 Euros. Our ATM machine will from now on have to accept cash deposits of this new bank note. Next to that, it must also be able to dispence the new bank notes. It is clear that this requirement brings along the need for both a hardware and a software update. Table 3 shows the atomic change sequence $A = \{A_1, A_2, A_3, A_4, A_5, A_6\}$ of the software update that we need to apply for obtaining the desired behavior.

ID	Type	Details
A_1	CM	<CustomerConsole, getWithdrawalInformation()>
A_2	CM	<CustomerConsole, getDepositInformation()>
A_3	CM	<CashDispenser, DispenseCash()>
A_4	CM	<Withdrawal, complete()>
A_5	CM	<EnvelopeAcceptor, acceptEnvelope()>
A_6	CM	<Deposit, complete()>

Table 3. Atomic Change Sequence of the update

3.4 Finding affected features

In this final step of the impact analysis, we want to establish the set of affected features (and their corresponding system entities). Next to that, we also want to specify which specific atomic changes are affecting the features of the system. This could be used as an extra feedback to the developer, telling him which atomic changes affect which system features. For doing that, we first need to establish the transitive closure of dependent atomic changes. An atomic change A_i is said to be dependent of an other atomic change A_j:

$$A_j \leftarrow A_i \qquad (A_i \text{ is dependent of } A_j)$$

if applying A_j without applying A_i, is conflicting (= results in a syntactical error). Taking this into account, we establish a partial order of all the atomic changes of an atomic change sequence A.

A feature F_k is determined to be affected by A_i:

$$A_i \Leftarrow F_k \qquad (F_k \text{ is affected by } A_i)$$

if its call graph contains either (i) a node that corresponds to an atomic change of type CM (changed method) or DM

(deleted method) change, or (ii) an edge that corresponds to an atomic change of type ML (lookup) change.

A feature F_k is said to be affected by an atomic change sequence A:

$$A \Leftarrow F_k \qquad (F_k \text{ is affected by } A)$$

if there is at least one atomic change A_i which is affecting F_k:

$$A \Leftarrow F_k \Leftrightarrow \exists A_i \in A \wedge A_i \Leftarrow F_k$$

In order to determine the set of atomic changes AF that are affecting a feature F_1 – $AF(F_k)$ – we say that an atomic change A_i is affecting a feature F_k if F_k is affected by A_i, or if there exists another atomic change A_j which is affecting F_k and from which A_i is dependent.

$$AF(A, F_k) \equiv \{A_i \in A \mid \wedge A_i \Leftarrow F_k\} \cup \{A_i \mid \exists A_j \in A \wedge A_i \leftarrow A_j \wedge A_j \Leftarrow F_k\}$$

We can say that a feature F_k is not affected by a software update A, if $AF(A, F_k) = \emptyset$ From the moment we have $AF(A, F_k)$ for all the features of the system, we can (i) give feedback to the developer of which parts of the atomic change sequence will be affecting which features, and (ii) we can start the dynamic updating process.

Tables 1, 2 and 3 respectively show the different system features, the call-graps, and the atomic change sequence of the ATM example. Analysing those tables and applying the technique explained above, results in:

$$\begin{aligned} AF(A, F_1) &= \emptyset \\ AF(A, F_2) &= \emptyset \\ AF(A, F_3) &= \{A_1, A_3, A_4\} \\ AF(A, F_4) &= \{A_2, A_5, A_6\} \\ AF(A, F_5) &= \emptyset \end{aligned}$$

This summarizes the findings of the analysis; the impact the atomic changes have on the different ATM features. From this moment on, we know for certain that both the features F_3 and F_4 will be affected by the update. This knowledge can be used for two kinds of feedback: (i) telling the programmer at compile-time that the update will affect those features, (ii) telling the user at runtime that those features are currently offline due to an update. Next to that, the knowledge can also be used for knowing which entities need to be deactivated before starting this update. How this is done, is explained in the following section.

4 Dynamic updating

From the moment we know which features (and their system entities) will be affected by the update, we are able to start the actual updating process. In this process, we first need to deactivate [2] those entities, in order to make sure they remain in a quiscent state [6] while the update is actually carried out. For doing this, we make use of a dynamic update framework that was previously presented in [2]. This framework uses a wrapper approach to stop incoming threads from running through the deactivated entities. Those threads are put into a waiting queue and are handled after reactivation. We also intercept the different entry points of the affected features (the first entity of the sequence or collaboration diagram) and make them display a widget that tells the user that this feature is currently offline because it is being updated, and that he should try again later.

After this, we perform the actual updates on the system. This is done by using the interceptive reflectional capabilities that are offered by a runtime API (presented in [2]). Once the update is completely carried out, we can reactivate the stopped entities and reset the entry points of the affected features. Note that the developer can also use the affected feature list as a means of feedback on the impact of his update. It is perfectly possible that it turns out some features, that were not meant to be affected by an update, would be affected anyway. If that would be the case, the developer can react, before actually carrying out the update.

5 Future work

As we have already mentioned in the paper, we only use design information for obtaining the link between features and system entities. Since there are systems that do not have this design information available or for which the design information is incorrect, a better technique is required. for this we are planning to incorporate feature extraction techniques. They could assist us for (i) identifying the different features and (ii) to discover their execution traces.

We intentionally skipped discussing the atomic changes that concern instance variables. However we already investigated how these atomic changes impact on the systems. In [11, 12] we discuss about how the state is mapped from one version to another.

Currently, we are only experimenting with the toy example we explained throughout this paper. However, we understand that it is a must to performing some real-world *case studies* for really testing this approach. Executing some *benchmark tests* on these cases would allow us to answer the question on whether real systems can be spilt up in different features, which can be deactivated separately. We

could easily imagine a system which features are so tangled so that all of them will always be affected by some update.

Currently, we are working on the implementation of the approach. While some parts are already finished, others, such as the feature classification in Starbrowser or not. However, we must say that we already see some possibilities for *optimisations* concerning speed and user friendliness. However, it is not clear yet, till what extent they can be formalized.

- Only deactivating features during critical parts of the update
- Reordering the atomic changes of the update, in order to provide shorter deactivation periods
- Determining which part of the update can maybe be postponed in order to leave some features active.

As a lot of programs are currently being written in the aspect oriented programming paradigm for disentangling the different features, we think that our approach should be extended to support this. That is why we foresee the addition of *aspectual atomic changes*; atomic changes that grasp the notion of aspect oriented programming. This way, we could allow the same approach for evolving aspect oriented programs.

6 Conclusion

In this paper, we propose a user-centric approach to dynamic evolution of applications. This approach involves two layers of abstraction. The concrete layer – the program constructs layer – consists of system entities and can be used for reasoning about source code. The abstract layer – the feature layer – consists of the different system features and is used for reasoning about system features.

Thanks to the fact that we maintain a link between the two layers, we are able to do a change impact analysis on the program constructs layer, and reason about it on the feature layer. This allows us to return feedback about the system features. It is this kind of feedback that separates this approach from already existing approaches to dynamic software evolution, as this feedback tells the user wether some feature can currently by used or not. Next to that, it can be used by the developer to see wether a certain update affects some unexpected system features.

References

[1] G. Antoniol and Y. Gueheneuc. Feature identification: A novel approach and a case study. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance)*, 2005.

[2] P. Ebraert, T. Mens, and T. D'Hondt. Enabling dynamic software evolution through automatic refactorings. In *proceedings of the Workshop on Software Evolution Transformations (SET2004)*, pages 3–7, 2004.

[3] Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. In *IEEE Computer*, volume 29(3), pages 210–224, March 2003.

[4] A. Eisenberg and K. D. Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance)*, 2005.

[5] O. Greevy, S. Ducasse, and T. Gîrba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance)*, pages 347–35, 2005.

[6] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[7] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. Separating features in source code: An exploratory study. In *Proc. 23rd Int'l Conf. Software Engineering*, pages 275–284. IEEE Computer Society, 2001.

[8] P. O´ Connor. *Practical Reliability Engineering*. Wiley, 4th edition edition, 2002.

[9] M. Oriol. *An Approach to the Dynamic Evolution of Software Systems*. PhD thesis, Université de Genève, 2004.

[10] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of java programs. In *Proceedings of the International Conference on Object Oriented Programming OOPSLA'04*, 2004.

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

[12] Y. Vandewoude and Y. Berbers. Deepcompare: Static analysis for runtime software evolution. Technical Report CW405, KULeuven, Belgium, Februari 2005.

[13] R. Wuyts. Starbrowser.

[14] R. Wuyts. Smallbrother - the big brother for smalltalk. Technical report, Université Libre de Bruxelles, 2000.