Keeping Design Documentation Updated through Synchronization of Use-Case Maps with Implementation

Martín Blech, Juan P. Carlino, J. Andrés Díaz Pace*, Alvaro Soria*
{mblech, jcarlino, adiaz, asoria}@exa.unicen.edu.ar

ISISTAN Research Institute, Faculty of Sciences, UNICEN University
Campus Universitario, (B7001BBO) Tandil, Bs. As., Argentina
*Also CONICET

Abstract. A key aspect for the success of architecture-centric development is the traceability of design documentation. This means that the developers should always be aware of the relationships between the architectural model as documented and the derived implementation. However, these two artifacts are likely to diverge from each other, due to new requirements or maintenance tasks. In the case of complex systems, the provision of tool support to deal with these matters becomes really necessary. Along this line, we present a tool approach called *ArchSynch* that can assist the developers to conciliate architectural documentation expressed through UCMs with implementation, as changes in the source code appear over time. The main advantage of *ArchSynch* is that detects, and eventually repairs, inconsistencies with respect to the architectural prescriptions.

1 Introduction

Architecture-centric development is being widely adopted as a software engineering practice. More and more, researchers and practitioners recognize that software architectures are very convenient models to do software design [4], because these models allow developers manage the bridge between requirements and implementation at a high level of abstraction. The value of having an architecture model resides on that it documents the principal design concerns early in the development process, embodies the overall structure of the system being built, and therefore, impacts the final quality of the products. Along this line, developers usually depart from an architectural description of the system and progressively refine this description until deriving a concrete implementation. During this activity, they also specify the relationships between the architectural model *as documented* and the architectural model *as implemented* [4, 8], in order to ensure some degree of consistency between the models. Several notations and profiles are nowadays available for this purpose [8].

Unfortunately, due to the natural evolution of the system, the architecture and implementation are likely to lose consistency. Once the architectural design is

ready, all the efforts typically focus on the implementation, and this progressively makes design documentation to get out-of-date. For instance, new requirements may cause a re-design of the architecture, with consequent changes in some parts of the implementation; and conversely, maintenance tasks may produce changes in the code that should be accommodated by the architecture. In these cases, the relationships of components (at the architectural level) with classes and methods (at the implementation level) no longer hold, and the developers have to manually restore consistency. This phenomenon is known as *architecture-implementation drift* [15], and if not properly managed, it may actually spoils the benefits of architecture-centric development. So far, the topic has been tackled through some approaches based on reverse engineering, with uneven results. The lack of more suitable tool support to relate architectural specifications with code is still a problem for many software projects.

In this work, we propose a tool approach called *ArchSynch* to aid developers to conciliate architectural documentation with implementation as the latter changes over time. This documentation only covers the behavior of components and some system structure. The *ArchSynch* approach has been developed in the context of the *FLABot* project[1]. In particular, we assume that some architectural documentation exists prior to the codification of the system. The architectural models are represented with Use-case Maps (UCMs), a practical notation to specify both components and flows of responsibilities. These responsibilities are materialized by specific classes and methods in the code. When something changes in the code and breaks what is prescribed by the architecture, *ArchSynch* is able to trace these changes back to the architectural specification. To do so, we have to instrument the application code and log information about its execution. For its analysis, *ArchSynch* compares these logs with the architectural specification, using number of "re-construction filters", and then detects which UCMs are inconsistent with the code. Moreover, *ArchSynch* can provide a list of possible repairs for the UCMs, so that the developer can execute them to update the architecture if necessary. This way, the developer is always aware of the "state of the traceability" between the architectural specification and the code.

The rest of the article is organized into 4 sections. Section 2 provides some core concepts regarding architectural documentation and use-case maps, and also explains the goals of the *FLABot* project. Section 3 presents the details of the *ArchSync* approach through a motivating example, and describes the implementation of the tool. Section 4 discusses some related work. Finally, section 5 summarizes the conclusions of this work and comments on future lines of research.

2 Background

A software architecture can be seen as a model of software organization, which serves to capture the most relevant design decisions regarding quality-attribute

[1] FLABot Homepage `http://www.exa.unicen.edu.ar/isistan/flabot/`

issues. Examples of architectural decisions include: the selection of a publisher-subscriber style, the allocation of responsibilities to a virtual-machine style, the choice of security checkpoints in a three-tier style, or the refactoring of some services accessed by many modules into a single module with an abstract interface, among others. Given the importance of these decisions for both stakeholders and developers, the documentation of software architectures is a central concern[8]. An architectural description gives basically a high-level decomposition of the system in terms of: design elements, coarse-grained pieces of functionality, plus interactions and constraints among them.

In practice, architectural models are described with box-and-line notations through a number of different views [8], such as modules, logic, processes, deployment, etc. In order to understand how the system works, these views should cover two aspects of the architecture: structure and behavior. On one side, the structure tells us from which components and connectors the systems are built, which patterns guide their composition in larger fragments, and what kind of constraints they should follow. On the other side, the behavior serves us to establish the correspondence between the system requirements and the responsibilities assigned to the different design elements. In particular, a useful view for specifying structure and behavior are *Use-case Maps (UCMs)* [6], which is the main architectural notation used by our approach.

The basic concept behind UCMs is to model functional scenarios by means of causal paths that cut across organizational structures. The core of UCMs has four elements, namely: responsibilities, paths, components, and couplings among paths. The *responsibilities* express the functions each component is responsible for. The *paths* trace the progression of causes and effects among responsibilities. The *components* act as containers for responsibilities. The *couplings* serve to connect paths together and form larger patterns (e.g., fork and join nodes, preconditions, stubs, etc.).

Figure 1 shows an example of UCMs, where we have a simple *"set user name" scenario* for a user account management system. The system is organized around a Model-View-Controller style [7], in which the *Model* component and its *View* components are decoupled by means of a *Controller* component. The responsibility *receiveWidgetEvent* in *Controller* translates the event produced by a *View* to the responsibility *setUserName* in *Model*, which then activates the responsibility *notifyModelObservers* also in *Model*. When a change notification is sent from *Model* to *Controller*, this causes the activation of the responsibility *handleModelChangeEvent*, which finally triggers the update of *View* by executing the responsibility *updateView*.

2.1 The FLABot project

Our interest in UCMs comes actually from a joint research project with Intel Corp., called *FLABot*[20], about tool support for fault-localization and debugging. As part of this project, we have developed a tool approach that is able to take the architectural model of an application and use this information to assist the developer in approximating the areas of the code where the errors are

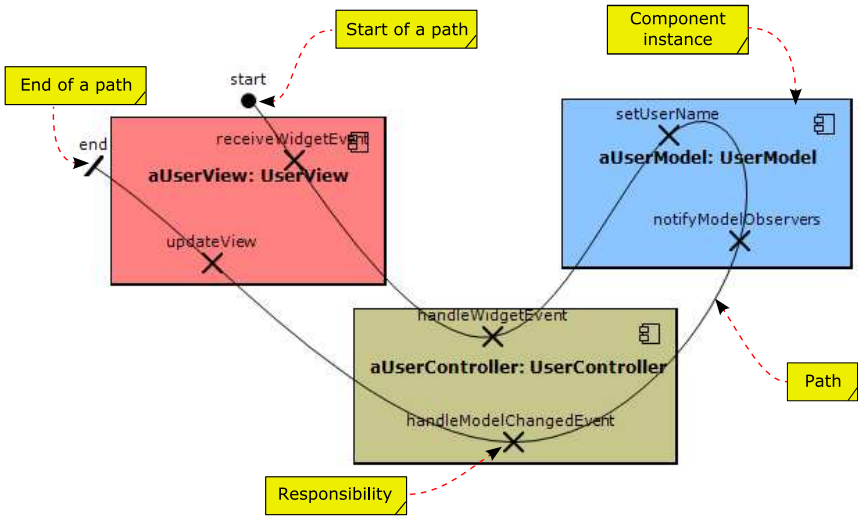


Fig. 1. A Use Case Map that shows a typical MVC application's control flow

most likely originated. Here, we decided to use UCMs for the specification of architectural models, because the notation fits well with the idea of exploring cause-effect paths as in traditional debugging approaches.

Besides, the issue of architectural drift is also relevant to the project. In order to do an effective fault localization, *FLABot* requires the input application to be implemented according to its architectural design. This is accomplished by linking each responsibility of the architectural UCMs with a collection of Java classes and methods. Although we can initially ensure that the UCMs are somehow in agreement with the implementation, maintaining traceability between the architecture and the code is not straightforward. In fact, when testing the fault-localization tool with different applications, we faced various architectural documentation situations. Specifically, as the faults detected by the tool were repaired by the programmers, they hardly updated the architecture to reflect their modifications. Many times, these programmers had little knowledge of the global design, so they could not rearrange the UCMs and their mappings to fixed code.

Motivated by these experiences, we started to develop a complementary tool able to help developers to keep architectural UCMs and Java code in-sync while using *FLABot*. A more general view of this issue led to the *ArchSync* approach, which is the topic of the following section.

3 ArchSync Approach

The objective of *ArchSync* is to find differences between a UCM description and a piece of source code given as input, trying to generate a new version of the

UCM where these differences are resolved as output. To achieve this functionality, we take advantage of a version control system and some execution logs of the application. The version control system provides information about those regions of the code that have been changed since the last update of the UCM. A general view of *ArchSync* is presented in Figure 2. Here, the *"source code Δ"* refers to the changes in the code, the *"UCM Φ"* is the original architectural description, and the *"UCM Φ+Δ"* is the architectural description generated after synchronization.

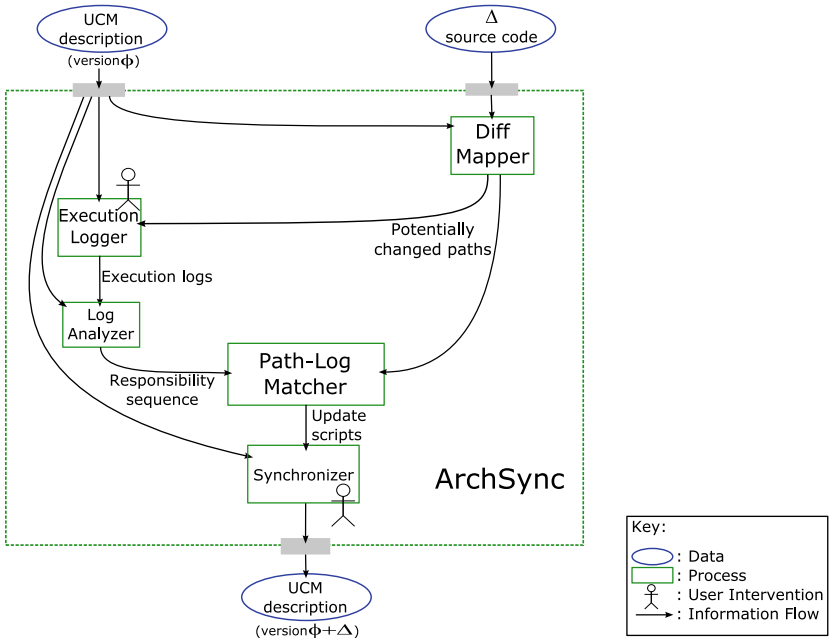


Fig. 2. Internal view of ArchSync

Internally, *ArchSynch* is equipped with a network of filters that implement different types of processing. The operation of some of these filters requires user intervention. As mentioned in the previous section, the *UCM Φ* comes already with a set of mappings to code, but many of mappings might be out-of-date in the *source code Δ*. Based on this information, the filter *DiffMapper* is in charge of detecting all the potential UCM paths that might have been affected by changes in the source code. Once these paths are identified, the developer is responsible of exercising the system on the modified portions of code, to gather data about its execution. Assuming that the code is somehow instrumented, the filter *ExecutionLogger* constructs a number of execution logs, which give traces of the actual system behavior with respect to the potentially-changed paths. The

instrumentation involves capturing a variety of low-level events such as: method calls, object creation, threads and exceptions, etc.

The mappings of *UCM* Φ are taken by the filter *LogAnalyzer*, so as to translate the low-level events of the logs into a sequence of responsibility activations. Each sequence of responsibilities is matched against the corresponding outdated path by the filter *PathLogMatcher*, which then exposes the mismatches between the *UCM* Φ and the *source code* Δ by creating a collection of *update scripts*. At this point, the filter *Synchronizer* may apply the user-selected *update script* on the *UCM* Φ to produce finally the *UCM* $\Phi + \Delta$. This way, UCMs can be put in agreement with the changes in the source code.

The *PathLogMatcher* filter takes as input a potentially changed path and a responsibility activation log (i.e., architectural-level execution trace). The algorithm in Figure 3 correlates both inputs, performing the necessary changes over the out-of-date path to conciliate it with the log. The algorithm works around a set of *cursors*, which keep the last points in the analyzed path that were successfully correlated and corrected. Every cursor points to some responsibility in the path, which is not yet correlated with some input in the log; this implies that all previous responsibilities in the causal sequence were correlated or corrected.

The set of update scripts begins empty, while the set of cursors starts with only a single element that points to the first responsibility of the path. After initialization, log entries are taken one by one. On each step, the algorithm selects a cursor to match against the current log entry. If the matching fails, that is, some unanticipated responsibility was activated, the path is changed to match the log correctly at that point. After that, the cursor is advanced to the next responsibility in the path and the algorithm processes the next log entry, until the entire log is consumed.

In principle, note that there are many ways to represent the same system behavior with UCMs. Because of that, the *PathLogMatcher* produces a set of alternative scripts rather than a single one. An update script is a sequence of atomic actions, such as the ones listed in Table 1, that allow us to transform UCMs. These scripts are ranked by the amount of changes on the old UCM that each script carries on, as the smallest scripts will more likely resemble the architect's style of specification. Then, the developer is free to select the script that (s)he finds most appropriate for the specific case of architectural design, and ask the *Synchronizer* to apply the script automatically.

3.1 A Sample Example

Let's go back to the UCM of Figure 1, assuming that it has been correctly designed and implemented. In a later development stage (e.g. during maintenance), a new requirement arrives, and now all the operations performed by the users must be shown in a log view. Let's consider that a developer that is not aware of the architectural design is assigned to the requirement, and (s)he inadvertently implements a call to the log view inside the user model. Figure 4 gives a possible Java implementation of the requirement. This clearly breaks the rules of

Script action	Description
`remove(targetNode)`	It removes *targetNode* from the path
`insertBetween(node1, node2, newNode)`	It inserts *newNode* between *node1* and *node2*
`insertAfter(targetNode, newNode)`	It inserts *newNode* after *targetNode* in the path
`transform(targetNode, newType)`	It transforms *targetNode*'s type to *newType*

Table 1. UCM script operations

```
path-log-match(path: Path, log: Log)
  cursors := { path.initialNode }
  script := Ø
  while log ≠ Ø
    cursor := selectCursor(cursors)
    logEntry := log.first
    if cursor.responsibility ∉ logEntry.responsibilities
      correctiveAction := selectAction(path, cursor, logEntry)
      script := script + correctiveAction
      path := correctPath(path, correctiveAction)
      cursors := correctCursors(path, cursors, correctiveAction)
    else
      cursors := advanceCursor(cursor, cursors)
      log := removeFirst(log)
    end if
  end while
  if cursors ≠ Ø
    script := script + removeAllFrom(cursors)
  end if
  return script
end path-log-match
```

Fig. 3. Path-Log Matcher Algorithm (non-deterministic points in ***italic***)

the MVC style, but unless there is a strict code review, the change (and similar ones) would remain unnoticed to the rest of the developers.

```
public class User {
  ...
  public void setName(String newName) {
    this.name = newName;
    // begin change
    logView.appendLogEntry('User name modified');
    // end change
    notifyModelObservers(...);
  }
  ...
}
```

Fig. 4. Souce code-level MVC violation

For dealing with this case of design documentation out-of-date, the *ArchSync* tool works as follows. First, the tool is notified of a change in the file *User.java*, which is further analyzed to detect that the method *setName()* was actually modified by some programmer. Since this method had a mapping to the responsibility *setUserName*, *DiffMapper* marks the path presented in Figure 1 as potentially changed. The developer runs the test-case on the current implementation of the system, which produces an execution log. This execution log is, in turn, processed by *LogAnalyzer* to generate a sequence of responsibility activations like the one shown in Figure 5. As a result of the comparison of this sequence with the original UCM, the *PathLogMatcher* suggests a list of update scripts. Some update scripts applicable to the outdated UCM are shown in Figure 6.

```
1. UserView.receiveWidgetEvent
2. UserController.handleWidgetEvent
3. UserModel.setUserName
4. LogView.addLogEntry
5. UserModel.notifyModelObservers
6. UserController.handleModelChangedEvent
7. UserView.updateView
```

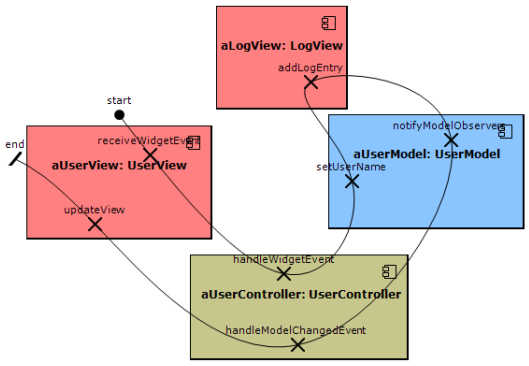
Fig. 5. Responsibility execution sequence

At this point, we should remark that the two UCM solutions proposed by *ArchSynch* match correctly with the implementation. In general, the variations in the solutions are due to the fact that UCMs are a quite abstract notation,

Update script A:

insertBetween(setUserName, \\
notifyModelObserver, addLogEntry)

↓



Update script B:

transform(setUserName, andFork)
insertAfter(setUserName, addLogEntry)

↓

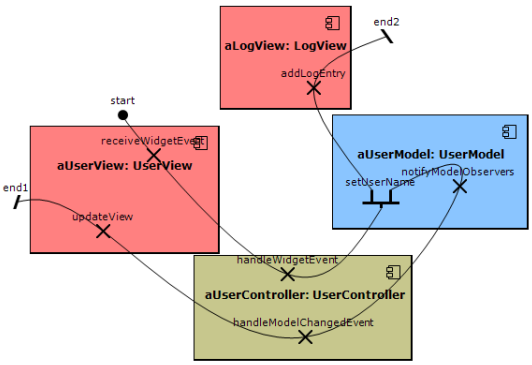


Fig. 6. UCM update alternatives

and thus, it gives a palette of options for implementing the same conceptual use-case. Likewise, as causal paths do not imply a strict flow of execution, a single execution trace might correspond to many different paths. This is why, even when all the solutions generated by *ArchSync* are considered correct, the architect is responsible for choosing the solution that is most appropriate for the actual design context. A related remark is that the solutions often suggest changes on the UCMs that compromise the integrity of the architectural model. For instance, the update scripts shown in Figure 6 reveal a style of implementation that violates the MVC architectural pattern. In these cases, the architect's criterion plays again a fundamental role. The architect can either proceed to apply the solution on his/her own, or decide that the requirement that originated the inconsistency must be re-implemented so as to conform the architecture.

3.2 Implementation Issues

The *ArchSync* prototype has been implemented as an Eclipse plugin, allowing the tool to be integrated within the development process. At its current state, the *source code* Δ is built by using the Eclipse platform's local history infrastructure, but integration with Concurrent Versioning System (CVS) and other configuration management technologies is underway. Additionally, the UCM editor and the instrumentation mechanisms have been actually borrowed from some of the FLABot project's subsystems. The *PathLogMatcher* component was implemented in Prolog using JavaLog [3] to take advantage of its Java-Prolog integration features.

The tool has been exercised with examples containing small source code changes between synchronizations, and also with examples with broader changes between synchronizations. As a result of evaluating the first group of examples, we found that the update scripts proposed by the tool were correct and generated new UCMs consistent with both the actual example implementation and the architect's specification style. Nonetheless, the update scripts proposed by tool for the second group of examples were considerably complex. That is, the amount of different alternatives for updating the UCMs made the problem impractical for being handled by the architect. Such a complexity was somehow expected and understandable for us, and it is because of this kind of situations that we imposed some restrictions for the approach. These restrictions are the following: (a) the initial UCM must be correct and (b) the updates must be performed regularly. This way, the gap that the tool must conciliate is likely to be of a manageable size.

4 Related Work

The problem of *architecture-implementation drift* has been tackled in different ways. Basically, we can recognize two groups of approaches: one focused on the recovery of the architecture through reverse engineering techniques [16, 18, 19, 5,

10] [12, 21], and another one using architectural description languages (ADLs) as support [1, 13].

In the approaches presented in [16, 18, 19], there are rules that specify the architect's intent, defining the permitted and forbidden relationships among the software components. These rules can then be used at any time to perform static and/or dynamic analysis over the source code to detect places where the architectural prescriptions are not followed. For example, Sangal et.al. [18] use a fixed matrix of components and relationships based on Java packages and their dependencies. Richner [16] and Sefika et.al. [19] provide more flexibility in that respect, by letting the user define both the clustering policy for components and the analysis strategies for different kinds of relationships via Prolog rules. As the main drawback, we see that these rules are not always intuitive to write or understand for common developers.

Other approaches to architectural recovery such as [5, 10, 12] attempt to extract the system architecture directly from its source code. Perhaps the most complete approach is the one discussed by Medvidovic et.al. [12], where the extraction activities are not only based on source code but rather complemented with requirements and architectural styles. More recently, Yan et.al. [21] have developed a tool called DiscoTect that maps low-level system events to abstract architectural operations, which are interpreted by a special engine to build the runtime structure of the analyzed system. This idea of monitoring a running application is close to the analysis of execution logs performed by *ArchSync*. The authors of *DiscoTect* have reported two successful case-studies for legacy systems, recovering a pipe-and-filter and a client-server style. The tool still needs to be evaluated with more complex architectural styles, where the mappings to code are not necessarily direct. Besides, none of these approaches have considered the extraction of paths of functionality, such as UCMs. Finally, Egyed [9] presents an automated approach to generate and validate trace dependencies between diverse software development artifacts. Such artifacts must be mapped to the source code of the implemented system. This approach also requires the designer to supply a set of test scenarios for the software system described, and a few hypothesized traces that link development artifacts. Test scenarios are executed on the running system to observe the lines of code they use. Execution traces are then combined to identify overlaps between them and interpreted to detect and analyze trace dependencies. This approach can generate new traces between model elements and scenarios, but the tool does not define the semantic meaning of those traces. This is not the case of ArchSync, where traces must be interpreted in the context UCMs that express causal order between significative responsibilities. The approach also depends on the systematic manual updating of the test cases to detect new dependencies on software changes.

On the other hand, Abi-Antoun et.al. [1] proposes an approach where an architectural specification in the Acme ADL [11] is kept in-sync with an implementation counterpart in ArchJava[2], by taking advantage of the fact that both Acme and ArchJava are languages where components and connectors are first-class entities. Both views, i.e. the architectural specification and the imple-

mentation, are transformed into instances of a common tree structure, which is then compared to detect changes such as node insertions, deletions and renames. Unfortunately, this synchronization does not address neither architecture dynamism nor behavioral conformance between architecture and implementation. Besides, it should be noted that although ArchJava is very similar to Java, it is not strictly an object-oriented programming language.

An early work of Medvidovic et. al. [13] describes a component-based environment targeted to the C2 style that enables architecture modeling, analysis and evolution based on an ADL that was specifically designed for these tasks. The environment assists the developer in the generation of an implementation for an evolutionary change that was performed on the architectural model. This is basically a top-down approach, well-suited for the ideal case where all changes are first applied to the architecture. Unfortunately, the common scenario is usually the opposite: the focus gears towards the implementation, and documentation is left out-of-date.

Regarding software configuration management (SCM) oriented approaches, we can find some approaches like [14, 17] that provide tool support for versioning the architecture-implementation relationships throughout design, implementation and deployment. For example, the tool presented by Nistor et.al. [14] imposes a systematic editing process for source code and architecture in order to guarantee that both models are kept in-sync. However, the propagation of changes from one model to the other is always performed manually by the developer. On the contrary, *ArchSynch* implements a more flexible model of edition, which allows the developer to have the control of when and how to synchronize the UCMs with the implementation.

5 Conclusions and Future Work

In this paper, we have introduced a tool approach to deal with the inconsistencies between architectural documentation and implementation. The contributions of *ArchSync* to prevent the architecture-implementation drift are two-fold. It can reveal violations to behavioral rules imposed by the architectural model. Moreover, this analysis can be used by the architect either to update the architectural UCMs with respect to the implementation, or to re-codify some parts of the implementation according to the base architectural model.

Currently, the *ArchSync* tool is still at the level of prototype, which has been mainly used within the *FLABot* project. Thus, there are some assumptions and limitations. First, when a change in the source code occurs, we assume that it is possible to know the previous mappings between the UCMs and implementation. This means that the old implementation is consistent with the scenarios of the UCMs. These mappings should be specified when the architect designs the UCMs for the first time and then proceeds to implement them. Second, we rely on that the synchronization between UCMs and implementation is carried out frequently on small increments (e.g., daily), so that the delta among source code versions presents only small variations. If not, the recognition of activa-

tion of responsibilities based on the events of the execution logs may become computationally unmanageable.

The main drawback of the approach is the gap between a given UCM and the many possible implementations for the paths of responsibilities. So far, the tool tries to reconstruct the paths of responsibilities using a bi-directional analysis, which combines reverse information coming from the execution logs with forward information coming from the existent UCM mappings. Even though, this processing is far from being automatic, because it requires considerable semantic knowledge that is actually supplied by the architect. The architect interacts with the tool at two points: to provide the right execution logs and to select the update scripts if necessary. Besides, we cannot identify yet new responsibilities introduced from the code or changes in the grouping of components. We are planning to incorporate here more intelligent strategies, based for example on machine learning and case-based reasoning techniques.

There are other interesting lines of work that can improve the performance of *ArchSync*. It is possible to integrate *ArchSync* with techniques for the synchronization of structural aspects to support more drastic evolutionary changes, such as refactorings. It is also possible to provide a suite of test-cases as a complement of the execution logs. Besides, the low-level information contained in the traces could be further analyzed to prune some operations in the generation of all the possible update scripts, and moreover, to detect source code changes that have been missed by the test cases. Finally, we believe the practical perspective taken in the design of *ArchSync* enables us to apply this tool as support for traceability in architectural design methods.

References

1. Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley Schmerl, Nagi Nahas, and Tony Tseng. Improving system dependability by enforcing architectural intent. In *Proceedings of the 2005 Workshop on Architecting Dependable Systems (WADS 2005)*, St. Louis, MS., USA, May 2005.
2. Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.
3. A. Amandi, M. Campo, and A. Zunino. JavaLog: A framework-based integration of Java and Prolog for agent-oriented programming. *Computer Languages, Systems and Structures*, 31, 2004. ISSN 0096-0551.
4. L. Bass, P. Clement, and R. Kazman. *Software Architecture in Practice. 2ed.* Addison-Wesley, 2003. ISBN 0-321-15495-9.
5. I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *ICSE'99*, Los Angeles, CA, USA, 1999.
6. R.J.A. Buhr. Use Case Maps: A New Model to Bridge the Gap Between Requirements and Design. Austin, TX., USA, October 1995. Contribution to the OOPSLA 95 Use Case Map Workshop.
7. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

8. Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.
9. A. Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, 29(2):116–132, 2003.
10. H. Gall, R. Klosch, and R. Mittermeir. Object-oriented re-architecting. In *ESEC-5*, Berlin, Germany, 1995.
11. D. Garlan, R. Monroe, and D. Wile. *ACME: Architectural Description of Component-Based Systems*. Cambridge University Press, 2000.
12. Nenad Medvidovic, Alexander Egyed, and Paul Gruenbacher. Stemming architectural erosion by coupling architectural discovery and recovery. In *STRAW'03: Second International SofTware Requirements to Architectures Workshop at ICSE 2003*, Portland, Oregon, USA, 2003.
13. Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 44–53, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
14. Eugen Nistor, Justin R. Erenkrantz, Scott A. Hendrickson, and André van der Hoek. Archevol: Versioning architectural-implementation relationships. In *Proceedings of the 12th International Workshop on Software Configuration Management*, Lisbon, Portugal, September 2005.
15. Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.
16. Tamar Richner. Using recovered views to track architectural evolution. In *ECOOP Workshops*, pages 74–75, 1999.
17. Roshanak Roshandel, Andre Van Der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. Mae-a system model and environment for managing architectural evolution. *ACM Transactions on Software Engineering and Methodology*, 13:240–276, 2004.
18. Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 167–176, New York, NY, USA, 2005. ACM Press.
19. Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high level design models. In *Proceedings of the 18th International Conference on Software Engineering*, pages 387–397. IEEE Computer Society Press, 1996.
20. Alvaro Soria. Architecture-driven fault localization of implicit invocation systems. In *Proceedings ASSE 2004, dentro de las 33 Jornadas Argentinas de Informática e Investigación Operativa (JAIIO)*, Córdoba, Argentina, September 2004.
21. H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. Discovering architectures from running systems using colored petri nets. http://www.cs.cmu.edu/afs/cs/project/able/ftp/discotect-jp05/DiscoTect.pdf.