Taking *ArchSync* to the Real World: An Analysis of Three Case-Studies

J. Andres Diaz-Pace* Alvaro Soria Guillermo Rodriguez* Marcelo R. Campo*

ISISTAN Research Institute, UNICEN University
Campus Universitario, (B7001BBO) Tandil, Buenos Aires, Argentina.
**Also CONICET-Argentina*
{adiaz, asoria, grodri, mcampo}@exa.unicen.edu.ar

Abstract. Architectural drift is a well-known phenomenon in software development, in which the architecture "as documented" and the architecture "as implemented" diverge from each other over time. To avoid this problem, periodic conformance checks should be in place, allowing architects to detect and correct differences. To this end, we developed a tool approach, called *ArchSync*, which helps architects to keep the main architectural scenarios of an application in-sync with its code. Our tool relies on heuristics that process application execution traces and correlate them with the intended architectural behavior. Despite being heuristic, *ArchSync* can bring mismatches upfront and reduce (re-)synchronization efforts. In order to validate these claims in practical settings, we have applied the tool in three medium-size case-studies for different domains. This article reports on the results of that evaluation, and also discusses insights and limitations regarding tool support for checking conformance to architectural behavior.

Keywords: conformance checking, tool support, architectural scenarios, evaluation.

1. Introduction

A software architecture is a design abstraction that helps to manage the bridge between requirements and implementation [2]. Ensuring conformance between an architecture and its implementation is a key aspect of architecture-centric development. However, the natural evolution of a software system and its environment leads to differences between the architecture "as-documented" and the architecture "as implemented" [2]. Examples include: new requirements causing an architecture re-design, with consequent implementation changes; reuse of code modules that alter the detailed design; technological decisions that trigger architectural updates; or code refactoring that the architecture must accommodate. This misalignment is known as *architectural drift* [9], and if the differences are not properly managed, the drift can seriously hinder the benefits of having a software architecture. Differences can involve structural or behavioral design aspects. An example of a structural difference is a code dependency between modules that is not allowed in the architecture. An example of a behavioral difference is an architectural interaction pattern that is not followed in the code.

Conformance checking is a common approach to tackle the problem of architectural drift and keep traceability between an architectural specification and its implementation. In practice, performing conformance checks in large systems is a time-consuming and error-prone activity [6, 8] in which architectural differences might be overlooked; therefore, tool support is vital for this activity. Existing semi-automated approaches have employed visualization, reverse engineering and consistency checking techniques [9, 5, 12, 1, 13]. A well-known technique is the Reflexion Model [10], which performs a comparison between two graphs: an intended architectural view X, and an architectural view Y that reflects the current implementation. From

this graph analysis, divergences and absences are identified. Commercial tools, such as Lattix, SonarJ, or Structure101, provide basic support for monitoring structural compliance. However, checking conformance to architectural behavior is still challenging because of the runtime characteristics of components. Detecting component interactions that deviate from the scenarios prescribed by the architecture cannot be achieved by looking (only) at structural system views.

In previous work [4], we have developed a tool approach called *ArchSync* that assists architects with conformance checks for the main architectural scenarios of an application. Our approach can be seen as an architecture-based simulation [2], and it basically tracks intended sequences of architectural responsibilities against actual system executions. In case of behavioral deviations, the tool can also suggest repairs to the architect, who must decide either to update the architectural specification or to fix the corresponding code. To do its work, *ArchSync* instruments the application code, collects traces, and then processes them via heuristics.

Although preliminary results showed the usefulness of *ArchSync* [4], a more extensive evaluation was required in order to support our claims. The fact that the approach relies on runtime information and heuristics must be also factored in the evaluation. For instance, the checks performed by *ArchSync* hint conformance issues but are not intended to detect all the problems in a system. Therefore, one can assess how good the tool works under particular assumptions, rather than whether it produces the "correct" results for a given conformance problem. The main assumptions for *ArchSync* are: i) an initial, consistent mapping between the architecture specification and the code being checked; ii) application traces chosen in such a way they expose behavioral differences; and iii) synchronizations between architecture and code that happen on a regular basis. Along this line, one should analyze the effect that variations in the tool inputs might have on the tool outputs. In light of these observations, this article describes a series of experiments applying *ArchSync* in three medium-size case-studies for different domains. As the main contribution of this work, we corroborate that the *ArchSync* assistance reduces architect's efforts to keep the architecture aligned with their implementation. Furthermore, we discuss lessons learned, practical limitations and research challenges regarding tool support for checking behavioral conformance to architectural scenarios.

The rest of the article is organized into 8 sections. Section 2 briefly describes the *ArchSync* approach and the experimental setup. Sections 3, 4, and 5 are devoted to the individual case-studies. Section 6 discusses a sensitivity analysis of the tool. Section 7 presents the findings of our evaluation. Finally, Section 8 concludes the article and comments on future work.

2. Overview of *ArchSync*

The main components of the *ArchSync* tool are shown in Figure 1. We consider two roles in a conformance checking process: the architect and the reviewer. The architect specifies the architecture and their code mappings, while the reviewer inspects the code for adherence to the architectural rules. The approach involves four activities (denoted as zero, A, B and C for our experiments). The first and fourth activities are carried out by the architect, while the second and third ones are carried out by the reviewer. Activity zero is the preparation for the remaining three activities, in which the tool assistance takes place. The activities are summarized below.

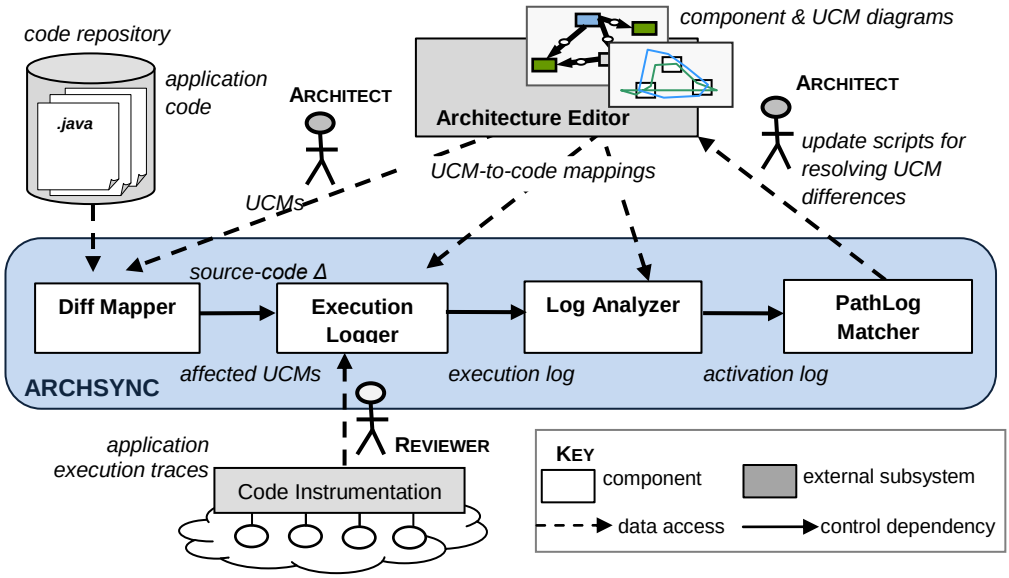
- Activity zero: The architect specifies the intended architecture of the application with Use-Case Maps (UCMs), a notation for modeling both high-level structure and behavior [3]. The architect also defines mappings from the architectural components and responsibilities of the UCMs to an object-oriented implementation. Specifically, a UCM models a set of scenarios by means of causal paths that cut across design structures. The basic UCM

elements are: responsibilities, paths, and components. A responsibility is a function that a component has to deliver. A component is a unit of computation and state, and also a container of responsibilities. A path shows a progression of causes and effects among responsibilities. UCMs are usually derived from informal requirements or from use cases.

Initially, we assume a Java codebase that is consistent with the UCMs. As developers make different changes, this code gets out-of-sync with the UCMs. Let's call *source-code delta* the set of changes between two consecutive code versions. After some period of development, the conformance checking itself takes place.

- Activity A: The reviewer feeds a pair of consecutive code versions into *ArchSync* and let the tool compute the delta with respect to original UCMs. Based on the resulting delta, the reviewer identifies the UCMs affected by changes and chooses test cases for them.
- Activity B: The reviewer exercises the test cases on the current implementation. In background, *ArchSync* instruments the application code and monitors its behavior, producing a set of execution traces. These traces are stored in a log, which is subsequently processed by the tool heuristics. Traces are converted into "responsibility activations".
- Activity C: The tool analyzes the differences between the UCMs and what was observed in the log, and presents the inconsistencies to the user. On this basis, the architect must decide the right actions to restore conformance. In general, four repairing actions are possible, namely: i) change the code as prescribed by the architecture; ii) refine the architecture to accommodate the current state of the code; iii) update the mappings between architecture and code; or iv) tolerate the code discrepancy as an exception to the architecture. *ArchSync* is able to recommend certain repairing actions, called *update scripts*, which synchronize the affected UCMs with the actual code. If the architecture must be updated, the architect picks a suitable script for the design problem and applies it on the UCMs.

Figure 1. Main actors and components of *ArchSync* assistance.



ArchSync is designed as an Eclipse plugin that interacts with a code repository, a logging system, and a graphical architecture editor. The core is a pipeline with four filters (see rounded

box in Figure 1). Filter *DiffMapper* implements activity A above, filters *ExecutionLogger* and *LogAnalyzer* both implement activity B, and filter *PathLogMatcher* supports activity C. For a more detailed description of *ArchSync* and examples, please refer to [4].

2.1. Experimental Setup

Our objective was to evaluate whether *ArchSync* is useful for checking conformance to UCM scenarios, given a series of implementation changes in the system. To do so, we performed a retrospective conformance analysis of three software projects (see Table 1) and exercised the *ArchSync* tool. For each case-study, we looked at historical changes in the project versions (issue tracker and configuration management system), and selected changes that were architecturally significant. By architecturally significant, we mean changes that have impact in the architecture, and thus, are visible in terms of UCMs. Once we sketched a sequence of versions, a conformance process was enacted by replaying changes from one version to the next one. This process was structured around the four activities discussed above. We carried out two types of experiments: one supported by *ArchSync*, and another one without the tool. The participants in these experiments were developers with a degree of involvement in the projects.

Table 1. Main characteristics of the case-studies.

Project Name	Project Type	Domain	Implementation	Architecture
Universidad3D	academia	education, 3D game	Java, ~370 classes, ~25 KLOC	multi-tier client-server,3D engine
InQuality	industry	process quality control	Java, ~1096 classes ~67 KLOC	Web client-server, event-based system
Government organization	government	management of social security	Java /C#, 1324 classes ~89 KLOC	service-oriented architecture

Two evaluation criteria were defined: i) the effort to perform activities A, B and C in the two types of experiments, and ii) the precision achieved by *ArchSync* when recommending update scripts. As effort indicators, we measured the time spent per activity by a participant, and the savings when she explored UCMs and Java classes for inconsistencies (relative to browsing all relevant cases). In the manual checks, these savings often come from the participant's domain knowledge, which helps her to avoid browsing unnecessary components or classes.
In preparation for the experiments (activity zero), the participants responsible for each case-study reviewed the project documentation (e.g., reports, UML diagrams, release notes, and ultimately Java code) and made the necessary adjustments to build an initial version of the UCMs and mappings to code. This pre-processing of documentation varied depending on the project and the available assets. Activities A, B and C happened next. For the experiments supported by *ArchSync*, the participants directed followed these three activities using the tool as needed. In background, we configured the *Mylyn* plugin[1] (as part of the Eclipse environment) in order to monitor the classes browsed and time spent by the participant on each activity. The analysis was complemented with the *Metrics* plugin[2] to collect code metrics about the deltas. For the experiments without *ArchSync* (also referred to as "manual" experiments), participants conducted activities A, B and C according to the following guidelines. First, the participant was asked to interpret the diff file between consecutive versions in order to figure out the main

[1] Mylyn homepage. http://eclipse.org/mylyn This plugin keeps track of user's activity within the context of a task, such as browsing of classes/methods, as well as time spent on tasks.
[2] Metrics homepage. http://metrics.sourceforge.net/

changes (i.e., the source-code delta). She also had to manually tag the UCMs related to those changes (activity A). Second, she was asked to browse the UCMs and code to find deviations (activity B). Third, she had to decide on the best fix for these deviations (activity C). We again gathered measures about time and browsed classes using the *Mylyn* and *Metrics* plugins.
In both types of experiments, we considered that *ArchSync* was successful for the delta under analysis, if any of the update scripts outputted by the tool was a satisfactory solution to the UCMs (according to the participant). Otherwise, we discarded the results. The average precision of the assistance was computed in terms of the satisfactory scripts for all versions exercised with the tool (irrespective of the existence of conformance problems).
To make comparisons across case-studies, we need a consistent ratio of "coverage" of UCMs over code. For a given version, this coverage is estimated as the classes receiving mappings from architectural components divided by the total classes in that version. We set an average coverage of 50% for the three case-studies. Also, we managed to have a time of 7-10 days between consecutive versions, ensuring that the UCMs were checked at regular time periods.

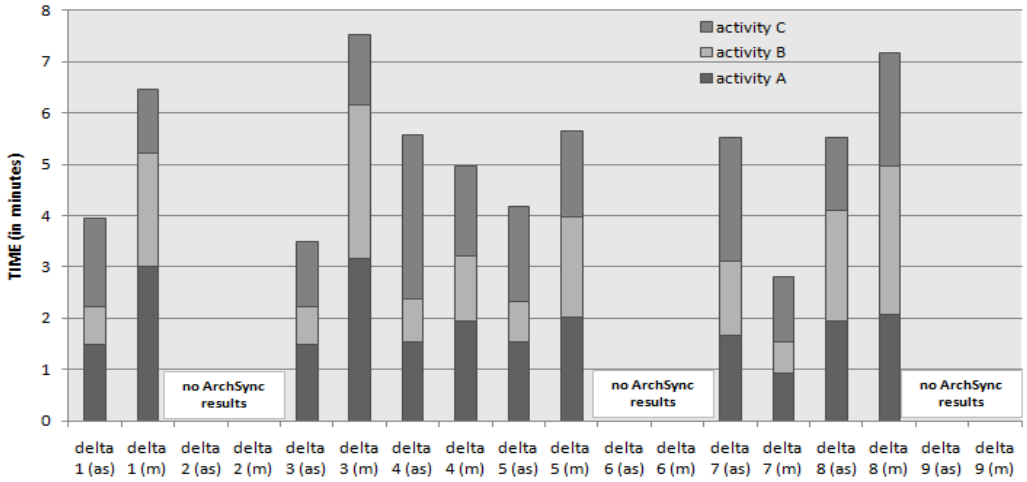
3. Case-study 1: *Universidad3D*

Universidad3D[3] is a 3D educative game developed by UNICEN University that simulates a virtual campus, allowing students to navigate the campus facilities and interactively learn about academic offerings. The core of the system is a Java 3D engine, which also incorporates chat, e-mail and forum mechanisms for communication between players. Furthermore, *Universidad3D* has features for constructing student profiles, with the goal of helping students to choose offerings that match their interests. One of the authors (Rodriguez) was the lead architect of this project, and the main participant that carried out the conformance process.
Universidad3D is designed as a multi-tiered client-server architecture. We scoped our analysis to the architectural design of the server, where most of the gaming functionality resides. Being an academic project, this case-study included already a baseline of UCMs as part of the architectural documentation. Building these UCMs required around 2 hours (activity zero), and the architecture modeling consisted of 7 components and 24 responsibilities. The final server implementation had 190 Java classes (approximately 13 KLOC). The participant selected 10 versions for the server, and then processed the 9 deltas between pairs of consecutive versions.
Figure 2 shows time efforts per activity for all deltas, in both the manual and *ArchSync*-supported experiments. We have blanks for deltas 2, 6 and 9. Although there were conformance issues in these three deltas (which were detected in the manual experiments), *ArchSync* failed to suggest valid update scripts for the UCMs[4]. We should note that the UCM synchronization might trigger (re-)mappings of responsibilities, which have to be set by the participant in both types of experiments. Thus, the time of activity C is always influenced by this manual effort.
Overall, the participant using *ArchSync* spent less (total) time for the UCM checks than performing the same checks manually. The most significant gain when using *ArchSync* was in activity B. There were also minor time reductions in activity A, when compared to the manual mode. Nonetheless, two exceptions to these trends were deltas 4 and 7, in which the time consumed by *ArchSync* exceeded the time spent without the tool. In delta 4 using *ArchSync*, the first two activities took a short time but fixing the UCMs required a long time. Specifically, a script generated by the tool touched a complex component with several responsibilities (component *GameLogic* in delta 4), and this suggestion made the participant analyze the whole component along with its mappings to code. Only a subset of the responsibilities in *GameLogic*

[3] Universidad3D homepage. http://isistan.exa.unicen.edu.ar/u3d/index.htm
[4] This problem was caused by an implementation bug in some rules of the correlation heuristic.

was actually affected by the change, but neither the tool nor the participant knew this fact a-priori. In the manual experiment for delta 4, the participant followed instead a more pragmatic strategy. During activities A and B, she acquired knowledge about the components and classes involved in the change, and later used that knowledge to inspect the right responsibilities in *GameLogic*. This fact exemplifies the role of domain knowledge in the conformance process.

Figure 2. Comparison of time efforts for the three conformance activities in case-study #1.



In delta 7, the manual experiment did much better than the experiment with *ArchSync* for each of the three activities. This happened because the change under analysis was actually very simple from an architectural perspective. We should note that, despite the nature of the change, the tool always makes a diff between the two code versions. This processing overhead is a limitation of *ArchSync*. Thus, performing a manual check turned out more economical. Activity C also took longer with *ArchSync* due to the mapping problem discussed for delta 4.

For the three activities, we computed the percentage of Java classes inspected by the participant while dealing with a conformance problem, relative to all relevant classes in the version. The effort savings were 35% and 65% in average for the manual and *ArchSync*-supported experiments respectively. These results show that, although *ArchSync* lacks domain knowledge, it can still reduce the user's search for conformance-related classes. The number of scripts proposed for a delta was around 11-17 scripts per UCM. The average precision for the *University3D* server was 0.81 (without counting blank deltas), showing that *ArchSync* recommendations are somehow affected by false positives.

4. Case-study 2: *InQuality*

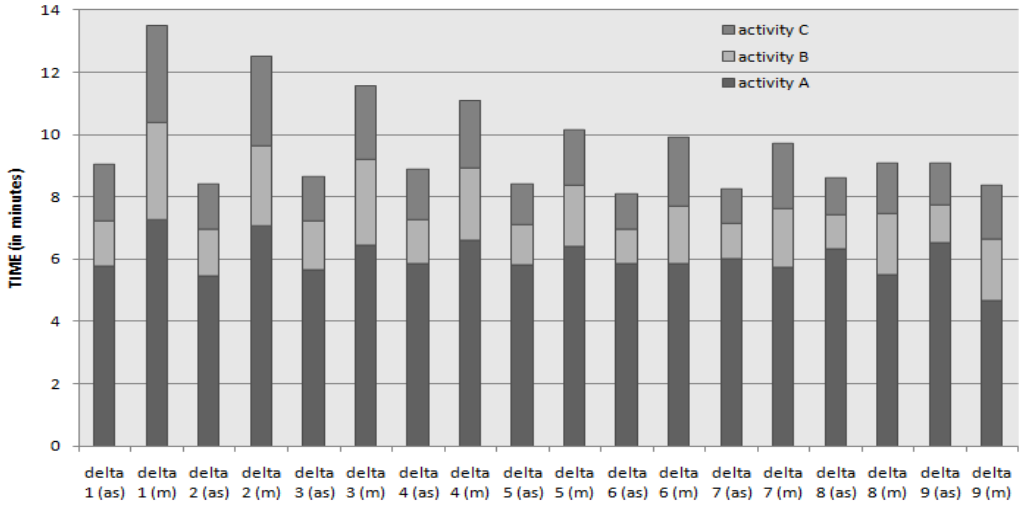
InQuality[5] is a commercial Web-based platform for Enterprise Quality Management systems developed by the Analyte company. Basically, *InQuality* provides a framework for managing and keeping consistent the corpus of documents and contents of an enterprise. The system is designed as a client-server architecture with a Web interface acting as client and an event-based system playing the server role. For our experiments, we focused on the server-side of the

[5] Analyte homepage & InQuality information. http://analyte.com/pdf/inQuality.pdf

framework. We modeled a sub-system that deals with the organizational structure and the edition of documents. The existing documentation relied on UML sequence diagrams rather than on UCMs. One of the authors (Soria), who worked as a consultant for Analyte, translated those sequence diagrams to UCMs suitable for the experiments. He also helped the participants specify the UCM mappings to Java classes. This initial effort (activity zero) required around 1.5 hours, and it consisted of 9 components and 23 responsibilities. The final implementation of this subsystem had 311 Java classes (approximately 12 KLOC). The participants in the experiments were two semi-senior developers at Analyte, performing one type of experiment each. We selected 10 system versions and processed the corresponding 9 deltas.

The time efforts for both the manual and *ArchSync*-supported experiments are shown in Figure 3. Here, the amount of code changed in every delta was higher than in case-study #1. Activity A consumed a reasonable time (~ 6 minutes) when using *ArchSync* in the present case-study, because the tool had more code to process when checking for changes. Across deltas, the experiments with *ArchSync* performed better than the manual experiments. The main time reduction brought by *ArchSync* happened in activity B. However, we observed that the time differences between the two types of experiments decreased towards the last deltas. In delta 9, the participant with *ArchSync* needed more time than the participant without the tool. We argue that this effect comes from two related factors: the changes were of the same extent (i.e., they affected similar sets of components and classes), and the participant gradually became proficient in dealing with those changes. Actually, we see that activity A slightly improved over time in the manual experiments. The situation is favored by the implementation guidelines of the *InQuality* framework, and also by the repetition of experiments on the same system.

Figure 3. Comparison of time efforts for the three conformance activities in case-study #2.

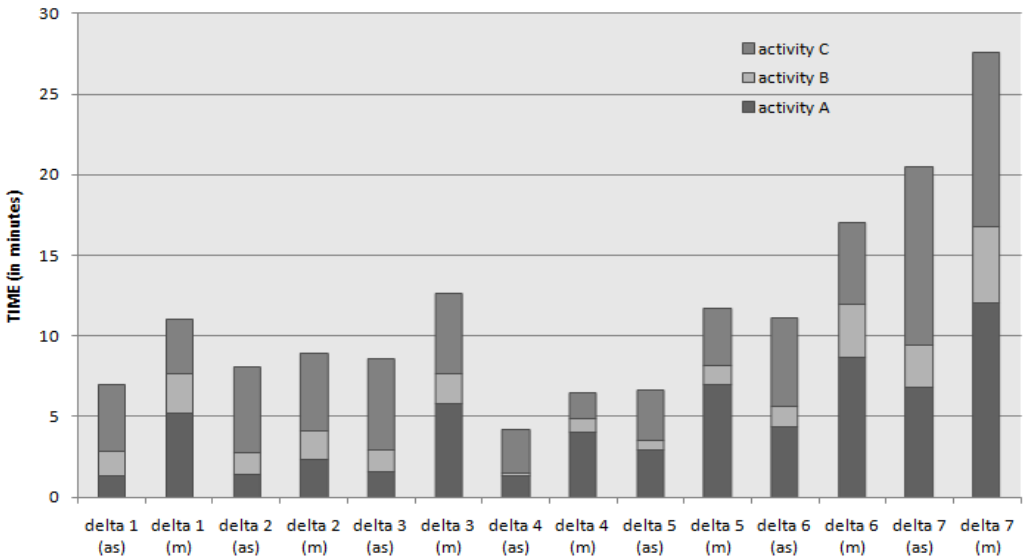


As regards effort savings, 42% and 66% of the relevant classes were explored by the participants in average, for the manual and *ArchSync*-supported experiments respectively. The average precision for delta was 0.80, with around 14-12 scripts proposed by the tool per UCM. Overall, these results are consistent with those of case-study #1.

5. Case-study 3: A Government Organization

This project[6] involved the migration of a legacy system to a Web services platform for an Argentine government organization (ANSES). The platform is intended to support several applications handling social security and employment data. The system is designed as a service-oriented architecture (SOA) with C# and Java implementations. For our experiments, we selected 3 Web services, each supporting several operations. These Web services were implemented with 60 C# classes and 12 Java classes (approximately 20 KLOC). The initial UCMs had to be created from scratch (activity zero). Two project developers along with one of the authors (Soria) were responsible for specifying these UCMs and their mappings. This architecture modeling required around 40 minutes, and it consisted of 3 components and 18 responsibilities. A technological aspect was that many artifacts were XML documents or configuration scripts rather than source code. In fact, several classes were automatically generated from XML specifications. Currently, the *ArchSync* tool does not support such artifacts at the implementation level, so the participants had trouble when mapping architectural elements. Because of this tooling issue, not all changes could be properly captured and processed by the tool. Two project developers were the participants in the experiments. Each participant was assigned to a separate type of experiment. We selected 8 system versions and processed the corresponding 7 deltas. Like in case-study #2, the team had guidelines for the development of services (these guidelines were established at delta 3).

Figure 4. Comparison of time efforts for the three conformance activities in case-study #3.



The time efforts for both the manual and *ArchSync*-supported experiments are shown in Figure 4. Across all deltas, the participant using the tool achieved smaller total times than the participant performing the checks manually. In the *ArchSync* experiments, we noticed speedups in activities A and B. In activity A, the delta processing times gradually increased for both types of experiments (remember that the tool processes all the code of the versions, regardless of the changes). In activity C, we had almost the same times for every delta, although the values increased along the versions. Here, we argue that the increase is due to the incremental

[6] Specific project details cannot be disclosed, due to confidentiality agreements.

development of services (from delta 5 on). In the process followed by the team, deltas 1-4 correspond to exploratory and design iterations, while deltas 5-7 are implementation iterations. The average precision for delta was 0.72, and the tool proposed around 11-16 scripts per UCM to the user. In the manual and *ArchSync*-supported experiments, we had average effort savings of 43% and 60% respectively. The figures for precision and relevant browsed classes were both in agreement with the results of previous case-studies.

6. Sensitivity Analysis for Synchronization Time and Mappings

Since *ArchSync* is based on heuristics, we analyzed whether the outputs of the tool are affected by variations in its inputs. Three input factors were mainly investigated: i) time interval between UCM conformance checks (for a pair of versions), ii) granularity in the architecture-code mappings, and iii) inconsistencies in those mappings.

Initially, the synchronization interval was just the delta between a version and the next one (remember that the reference interval is 7-10 days in the case-studies). As variations of this configuration, we delayed the time between UCM synchronizations, considering that a reviewer performs checks every 2 versions and every 3 versions. We conjectured that this separation should degrade the *ArchSync* performance, i.e., the number of update scripts generated by the tool and its precision. Regarding the mapping policies, we started with a set of coarse-grained, consistent mappings for the UCMs. In the coarse-grained variant the mappings are viewed from an abstract perspective, that is, the mappings for components and responsibilities are generally one-to-many. On the contrary, in the fine-grained variant the mappings are very detailed, that is, most mappings between components and Java classes and between responsibilities and Java methods are one-to-one. The mapping choice affects the generation of the application traces and their translation to responsibility activations. We were interested in two variations here: making the mappings more fine-grained, and having erroneous responsibility mappings. These modeling variations can be attributed to architects with different background. We measured the effects of the mapping variations in terms of tool precision.

We performed three analyses on case-study #1 and case-study #2, modifying one factor at a time. First, the mappings were as given (in the case-study), and we tried out configurations with 2 and 3 time intervals for UCM synchronizations. Second, the time interval was kept as in the initial setting, and we used consistent but fine-grained mappings for the existing UCMs. Third, we kept the time interval and initial coarse-grained mappings, but we introduced inconsistent mappings. Table 2 shows the original precision for case-study #1 and case-study #2 (grey column), and also the precision results as we explored different *ArchSync* configurations.

Table 2. Variations in tool precision for different configurations (the gray column shows reference values; the underlined text indicates the factor being changed in each experiment)

delta →	**interval 1**	**interval 2**	**interval 3**	**interval 1**	**interval 1**
mappings →	**coarse granularity**			**fine granularity**	**coarse granularity**
	consistent			**consistent**	**inconsistent**
Universidad3D	0.81	0.75	0.50	0.62	0.38
InQuality	0.80	0.71	0.63	0.72	0.46

From the experiments with incremental intervals between UCM synchronizations, we noticed a drop in tool precision for both case-studies. We also observed a lower number of (relevant) update scripts for each UCM. In case-study #1, for instance, the tool suggested 14 scripts with

the reference configuration, and this number went down to 9 and 5 scripts as we expanded the synchronization interval. These results indicate that the accumulation of changes over time has a negative effect on the *ArchSync* heuristics, degrading the recommendations for the UCMs.

When it comes to the granularity of the mappings, we again noticed that the tool precision decreased for both case-studies when using fine-grained mappings. Interestingly, the loss of precision in case-study #2 was not as much as in case-study #1. This situation can be explained in terms of the context information provided by the application traces. If the granularity is very detailed, the execution of the test cases might not cover all the methods that are needed to expose an inconsistency at a given responsibility. As the granularity level gets higher, we have more mappings per responsibility, and thus, there is a better chance of covering methods for that responsibility in the execution traces. In case-study #2, the reason for the precision difference was that the fine-grained mappings covered a considerable part of the same architectural behavior covered by the coarse-grained mappings. A contributing factor here was the framework that supports *InQuality*, as this framework implements many architectural concepts directly and makes their mappings quite straightforward. That is, the gap between the application architectural design and its implementation is smaller than the gap in case-study #1.

At last, inconsistent UCM mappings were evaluated for the two case-studies. To do so, we randomly created instances in which some components or responsibilities were mapped to wrong Java classes or methods. We observed different responses of the tool, namely: UCMs mistakenly tagged by *ArchSync* as affected by a change, no recommendation of update scripts, or a high number of scripts but with many false positives. This "confusion" was mostly caused by awkward responsibility matchings in the correlation heuristic. Consequently, the experiments showed a considerable loss in tool precision for both case-studies.

All the evidence above reinforced our view of the UCM specification as a key asset in the *ArchSync* approach. Creating the UCMs and their mappings is not a low-effort activity for the architect, but it pays off when the tool assists her with the conformance checks.

7. Lessons learned and Limitations

The analysis of the experiments shows that *ArchSync* does facilitate the synchronization of UCMs with code. The tool helped participants by reducing their time and browsing efforts during conformance tasks, as well as pointing UCM mismatches to these participants. Regarding global time spent in a conformance exercise, the results of the *ArchSync*-supported experiments generally outperformed those of the manual experiments. In particular, the tool sped up activity B, meaning that the participants quickly discovered mismatches between a UCM and its corresponding implementation. The tool also contributed to a small improvement in activity A related to detection of source-code deltas. Nonetheless, some participants were equally able to identify these deltas on their own, reporting similar (or even lower) times as with tool assistance. The fact that *ArchSync* has to process the whole code of the versions being analyzed leads to a cost (proportional to the version sizes) that adds up to the global time. In case of simple changes, manual conformance checks are more economical than tool-supported ones, because the participants can manually inspect the diff files and rely on their application knowledge. At last, activity C consumed almost the same time in both types of experiments. This result was expected, as all participants had to assess the differences and decide on fixes for the UCMs or the code. So, these fixing efforts are independent from the *ArchSync* capabilities.

From the analysis of the browsing efforts (i.e., participants inspecting only relevant code elements) with and without *ArchSync*, we had positive findings. For instance, the average effort reduction was 40% for the manual experiments. This initial value is justified by the developer's

knowledge of the application. Moreover, the effort reduction increased to 63% in average for the *ArchSync*-supported experiments. The participants said that they perceived this improvement because the tool "focused" their conformance revisions usually on a single component, whereas in the manual revisions the participants tended to look across several components. The more the inspected components, the more the classes mapped from them, leading to higher browsing efforts. Furthermore, there is a relationship between the mappings of an application and its underlying framework. A framework often prescribes "implementation patterns" for the application components (like in case-study #2 and case-study #3), and thereof conditions the kind of changes undergone by the component implementations. These factors simplify conformance tasks, as we noticed mainly in case-study #2. Another finding was that, after repeating similar conformance checks, the participants (without *ArchSync*) might "counteract" the tool improvements and achieve a comparable performance.

The precision of the tool was acceptable, as long as the UCM checks are frequent and the UCM mappings are consistent. However, we still found problems in the generation of scripts for certain deltas. We had experiences of erroneous mappings or test cases that caused deflections in the responsibility activations, which in turn affected the UCM correlations. These problems can be traced to the implementation of the heuristics. For instance, if a delta contains a sequence of responsibilities (in the code) that are missing in a UCM path, the heuristic can only detect the first responsibility but not the remaining ones. A related issue is that of changes that are "out of context" of the code already mapped to UCM responsibilities. This situation happens when a new architectural feature, with no relationships to existing features, is only added to the code. The tool did not recognize these changes as relevant to the current UCMs.

Overall, we believe that the *ArchSync* approach is applicable to other case-studies. Anwyay, there were some threats to validity in our evaluation. A threat to construct validity is the retrospective nature of the experiments, which replayed changes from the project repositories. The main threats to internal validity were: i) the four-activity conformance process that facilitated comparisons during the experiments; ii) the choice of versions for each case-study and the input artifacts fed into *ArchSync*; and iii) the impact of code instrumentation on the application behavior. These factors might have biased the conformance problems under analysis. The main threats to external validity were: i) the alternative "modeling options" of UCMs for representing the same architectural functionality; and ii) the participants of the experiments being familiar with their respective projects. Other architectural specifications or participants with different background might have led to different conformance efforts.

8. Conclusions

In this article, we have presented the findings of applying the *ArchSync* tool in three real-world systems. In the experiments, different participants performed periodic conformance checks between architectural scenarios and code, across several project versions. The goal of *ArchSync* is to help architects prevent the drift of architectural behavior. The approach relies on a UCM-based representation with predefined mappings from responsibilities to code.

The present evaluation was mainly motivated by the heuristic nature of *ArchSync* assistance. The results from the analysis of the case-studies were encouraging. We noticed that *ArchSync* reduces the architect's efforts to keep the UCMs aligned with their implementation over time. A perceived benefit of the tool is its ability to bring relevant changes upfront, so that developers are not overwhelmed by many suspicious elements (i.e., UCM paths or classes). The checks performed by the tool hint conformance issues but are not intended to detect all the problems in a system. Nonetheless, we found the UCM deviations and scripts proposed by *ArchSynch* very

practical, if compared to tedious manual revisions of documentation and related code, especially when code changes happen at a faster pace than architectural reviews.
Anyway, the approach has still room for improvements. For example, experiments to determine the tool scalability in large projects are still pending. There are also several lines of future work. On the modeling side, we are creating guidelines for developers to specify UCMs and map them to code. A candidate technique to explore is incremental mapping [7]. In order to facilitate the generation of application traces, we plan to link the UCMs with test suites for the architectural responsibilities. To make the tool more effective, we will use structural checks to complement the behavioral checks currently implemented by the heuristics. It is possible to detect whether a UCM responsibility executed with errors and then incorporate this information into the correlation algorithm. Another interesting topic is to extend *ArchSync* to work with architectural paths annotated with quality-attribute properties. Also, it is desirable to strike a balance between the events logged by the tool and accuracy of its outputs. We will explore model-checking and configuration management techniques [11, 14] to achieve this balance.

References

1. Abi-Antoun, M. and Aldrich, J. Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. ACM, 2009. OOPSLA'09. Vol. 44, pp. 321-340.
2. Bass, L., Clements, P. and Kazman, R. Software architecture in practice. 2nd Edition: Addison-Wesley, 2003.
3. Buhr, R. J.A. Use case maps as architectural entities for complex systems. 12, s.l. : IEEE Computer Society, 1998, IEEE Transactions on Software Engineering, Vol. 24, pp. 1131-1155.
4. Diaz-Pace, J. A., Carlino, J.P., Blech M., Soria A., Campo, M.R. Assisting the synchronization of UCM-based architectural documentation with implementation. Cambridge, UK. IEEE Computer Society, 2009. WICSA/ECSA 2009. pp. 151-160.
5. Kazman, R. and Carriere, S. Playing detective: Reconstructing software architecture from available evidence. Springer, 1999, Automated Software Engineering, Vol. 6, pp. 107-139.
6. Knodel, J., Muthing, D., Haury, U., and Meier, G. Architecture Compliance Checking-Experiences from Successful Technology Transfer to Industry. Athens, Greece. IEEE Computer Society, 2008. 12th European Conference on Software Maintenance and Reengineering, CSMR '08. pp. 43-52
7. Koschke, R. Incremental Reflexion Analysis. Madrid, Spain. IEEE Computer Society, 2010. 14th European Conference on Software Maintenance and Reengineering, CSMR'10.
8. Lilienthal, C. Architectural Complexity of Large-Scale Software Systems. Kaiserslautern, Germany. IEEE Computer Society, 2009. 13th European Conference on Software Maintenance and Reengineering. CSMR '09. Pp. 17-26.
9. Medvidovic, N., Egyed, A. and Gruenbacher. Stemming architectural erosion by coupling architectural discovery and recovery. P. Oregon, USA: s.n., 2003. STRAW'03, Software Requirements to Architectures Workshop at ICSE'03.
10. Murphy, G., Notkin, D. and Sullivan, K. Software reflexion models: Bridging the gap between design and implementation. IEEE, 2001, IEEE Transactions on Software Engineering, Vol. 27, pp. 364-380.
11. Murta, L. P. G., van der Hoek, A. and Werner, C. M. L. ArchTrace: Policy-based support for managing evolving architecture-to-implementation traceability links. IEEE Computer Society, 2006. 21st International Conference on Automated Software Engineering, ASE'06. pp. 135-144.
12. Nentwich, C., Emmerich, W. and Finkelstein, A. Consistency management with repair actions. Portland, Oregon, USA: IEEE Computer Society, 2003. ICSE 2003. pp. 455-464.
13. Postma, A. A method for module architecture verification and its application on a large component-based system. Elsevier, 2003, Information and Software Technology, Vol. 45, pp. 171-194.
14. Sousa, F., Mendoca, N., Uchitel S., and Kramer J. Detecting implied scenarios from execution traces. IEEE Comp. Society, 2007. 14th Working Conference on Reverse Engineering, WCRE'07. pp. 50-59.