Information and Software Technology 54 (2012) 448–466

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Assisting conformance checks between architectural scenarios and implementation

J.A. Díaz-Pace [a,b], Álvaro Soria [a], Guillermo Rodríguez [a,b,*], Marcelo R. Campo [a,b]

[a] ISISTAN Research Institute, UNICEN University Campus Universitario, (B7001BBO) Tandil, Buenos Aires, Argentina
[b] CONICET, National Council for Scientific and Technical Research, Argentina

ARTICLE INFO

Article history:
Received 19 July 2011
Received in revised form 15 December 2011
Accepted 18 December 2011
Available online 28 December 2011

Keywords:
Software architecture
Conformance checking
Behavioral scenarios
Tool support
UCMs

ABSTRACT

Context: Conformance between architecture and implementation is a key aspect of architecture-centric development. Unfortunately, the architecture "as documented" and the architecture "as implemented" tend to diverge from each other over time. As this gap gets wider, the architects' reliance on architecture-level analyses is compromised. Thus, conformance checks should be run periodically on the system in order to detect and correct differences. In practice, tool support is very beneficial for these checks.
Objective: Despite having a structural conformance analysis, assessing whether the main scenarios describing the architectural behavior are faithfully implemented in the code is still challenging. Checking conformance to architectural scenarios is usually a time-consuming and error-prone activity. In this article, we describe a tool approach called *ArchSync* that helps architects to reconcile a scenario-based architectural description with its source code, as changes are being made on the code.
Method: The architecture is specified with Use-Case Maps (UCMs), a notation for modeling both high-level structure and behavior. *ArchSync* applies heuristics that incrementally detect code deviations with respect to predetermined UCMs, based on the analysis of system execution traces for those UCMs. Also, *ArchSync* provides suggestions for re-synchronizing the UCMs with the code.
Results: We have evaluated a prototype of *ArchSync* in three medium-size case-studies, involving developers with moderate architecture experience. We compared time consumed, code browsed and suggestions for re-synchronizing the UCMs by these developers, with and without the support of *ArchSync*.
Conclusion: The results from case-studies and lessons learned have shown that the *ArchSync* approach is practical and reduces conformance checking efforts.

1. Introduction

Architecture-centric practices are being more and more adopted in the software development community. A software architecture is a design abstraction that helps to manage the bridge between requirements and implementation [1]. Creating a software architecture has several benefits. First, the architecture captures the main decisions to satisfy the quality-attribute concerns (e.g., modifiability, performance, security, etc.) derived from the stakeholders' business goals. Second, the architecture supports analyses in early development stages (e.g., change impact analysis). Third, the architecture prescribes how those quality-attribute concerns should be addressed in the system implementation. In order to realize these benefits, an architecture is normally documented via views showing different system structures, and via behavioral specifications for the key architectural scenarios [2]. However, having a good architectural documentation is not sufficient to ensure a compliant system implementation. The natural evolution of a software system and its environment leads to differences between the architecture "as-documented" and the architecture "as implemented" [1]. Examples include: new requirements causing an architecture re-design, with consequent changes in the implementation; reuse of code modules and libraries that alter the detailed design; technological decisions that trigger architectural updates; or code refactoring that should be accommodated by the architecture. This misalignment is known as *architectural drift* [3]. As a result, the architectural documentation gets progressively out-of-date, and if the drift is not properly managed, it can seriously hinder the benefits of architecture-centric development.

A common approach to check architecture conformance is code reviews [4,5]. Conformance checks rely on mappings between architectural elements (e.g., modules, components, layers, responsibilities) and implementation elements (e.g., packages, classes, methods in the object-oriented paradigm). Based on these mappings, a reviewer walks through the implementation and searches for *differences* with respect to the architecture, either by hand or with tool assistance. The differences can involve structural or

* Corresponding author. Address: ISISTAN Research Institute, UNICEN University Campus Universitario, National Council for Scientific and Technical Research, (B7001BBO) Tandil, Buenos Aires, Argentina.
E-mail addresses: adiaz@exa.unicen.edu.ar (J.A. Díaz-Pace), asoria@exa.unicen.edu.ar (Á. Soria), grodri@exa.unicen.edu.ar (G. Rodríguez), mcampo@exa.unicen.edu.ar (M.R. Campo).

0950-5849/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2011.12.005

behavioral aspects of the design. An example of a structural difference is a code dependency between modules that is not allowed in the architecture. An example of a behavioral deviation is an interaction pattern between architectural responsibilities that is not followed strictly in the code. Once differences are identified, the architect must decide on the actions to restore conformance.

Checking conformance in large systems is a time-consuming and error-prone activity in which architectural differences might be overlooked; therefore, tool support is vital for this activity. A first step is to check for structural differences regarding architectural views. Existing semi-automated approaches have tackled this problem through reverse engineering or static dependency analyses [6–10]. These approaches are good at exposing divergence and absence relationships between architectural and code elements. A second step is to check for behavioral differences, that is, to find system interactions that deviate from the *behavioral scenarios* prescribed by the architecture. The relationships in architectural scenarios tend to be more complex than the relationships in structural views. A scenario will typically describe how architectural elements may affect one another based on time-related or state-dependent interactions. In this context, the approaches above fall short, because behavioral conformance is difficult to analyze by looking only at (structural) architectural views. A relatively simple approach to this problem is to run test cases on the code for each architectural scenario. Nonetheless, by the time these tests are executed, several behavioral deviations might be already spread over the code. An alternative approach is that the reviewer plays out the main scenarios against the architectural views and compares the results with the expected behavior – this technique is called architecture-based simulation [2]. It is possible to (partially) automate this kind of reviews, if we actually track intended sequences of architectural responsibilities against actual executions of the system. *ArchSync* is a tool approach that implements these ideas and assists architects to perform conformance checks for architecturally-significant scenarios.

In this work, we provide an in-depth description of the *ArchSync* tool infrastructure. The building blocks of the approach were outlined in previous work [11]. Basically, *ArchSync* takes three inputs: (i) the intended architecture specified with Use-Case Maps (UCMs) [12]; (ii) a Java code base implementing the architecture; and (iii) a set of system execution traces that exercise UCM scenarios. Then, *ArchSync* applies heuristics to correlate the UCMs with information about responsibilities extracted from the traces. In case of behavioral deviations, the tool searches for local repairs to the input UCMs that make them consistent with the collected traces. As output, *ArchSync* suggests those repairs to the architect, who is responsible for making informed decisions either to update the UCMs or fix the corresponding code. The contributions of this article are twofold. First, we elaborate on the heuristics used by the tool for processing the execution traces and for suggesting UCM repairs. Second, we report on the results of three case-studies using the tool. The findings have corroborated our initial results [11], showing that *ArchSync* can reduce the architect's efforts spent on keeping the architecture and implementation in-sync.

The rest of the article is organized into five sections. Section 2 gives background information about architecture conformance, and explains the *ArchSync* assistance through a motivating example. Section 3 describes the components and heuristics of the *ArchSync* infrastructure. Section 4 discusses the case-studies. Section 5 covers related work. Finally, Section 6 presents the conclusions and comments on future lines of work.

2. Conformance technical approach

The software architecture of a system is captured by the architecture documentation, which describes the main design elements (e.g., modules, components, deployment units, etc.), coarse-grained functions, interactions among them and design constraints. It is customary to organize the documentation into a set of views [2] such as: module views, runtime views, and allocation views. Normally, the implementation is not part of the architectural documentation but it is derived from it. Therefore, there must be a correspondence (or mappings) between the architectural views and their implementation in code. For instance, we can decide to map an architectural module to a set of Java classes, and further map a responsibility (allocated to that module) to methods in some of those classes.

For an implementation to comply with the intended architecture, it should be divided into the elements prescribed by the views, these elements should interact with each other according to the relations prescribed in the views, and each element should fulfill its responsibilities to the others as dictated by the views [1]. From this perspective, checking conformance is a correlation between two graphs: an intended architectural view X, and an architectural view Y that reflects the current implementation. Notice that the mappings are what informs the (re-)construction of view Y from the implementation.[1] This idea was originally proposed in the Reflexion Model [7] and has been extended by several authors since [13–15]. From the graph comparison, we obtain a list of *architectural differences* in the form of absences or divergences with respect to the intended view. This structural comparison can be more or less complex, depending on the view type and semantics of the elements and relationships permitted by the view. For instance, checking a module view against code is relatively straightforward, while checking a component-and-connector (C&C) view against code is harder due to the runtime characteristics of components and connectors [16]. Once differences are detected, several *repairing actions* can be taken in order to restore conformance 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. Even using tools, the process of ensuring architecture conformance requires periodic (code) checks, discipline, and active involvement of architects and developers.

The assessment of differences in architectural behavior has been less explored. A behavioral difference appears when the code deviates from the interaction patterns specified for the architectural components. For instance, the Model-View-Controller (MVC) pattern [17] comes with specific interaction rules for models, views and controllers regarding change notification; but a programmer might not obey these rules when implementing an instance of MVC. More specifically, let's consider that each pattern (or rule) is expressed as a behavioral scenario that consists of a sequence of (expected) architectural functions. Conformance here means that any implementation trace for that scenario should match the functions in the sequence. If not, we have a behavioral difference with respect to the architecture. A sequence mismatch can be due to required architectural functions that are missing in the trace (i.e., absences), or functions detected in the trace that do not belong to the sequence (i.e., divergences).

We did experience situations that required behavioral conformance checks in a research project called *FLABot* [18]. *FLABot* is a tool that performs root cause analysis at the architectural level in order to guide developers to identify possible faults in Java code. *FLABot* represents the architecture with Use-Case Maps (UCMs) [12], a notation for modeling both high-level structure and behavior, which fits well with the cause–effect scenarios necessary

[1] The graph difference can be alternatively performed at the implementation level, using the mapping function to project view X onto the code. The premise for the comparison is that the graphs must use the "same language".

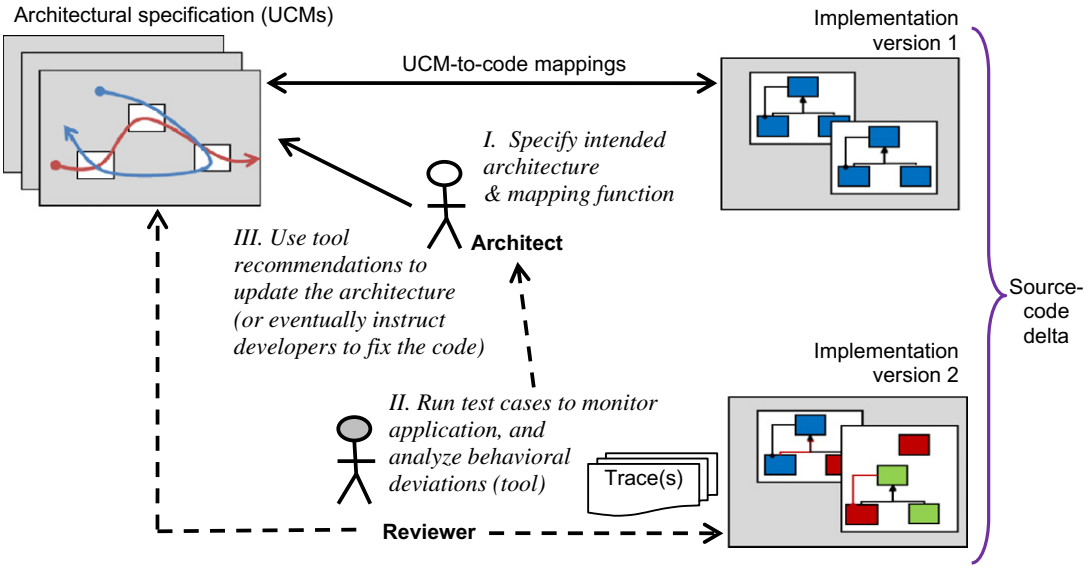

Fig. 1. Conformance phases in *ArchSync*.

for reasoning about faults. The validity of the analysis depends on having up-to-date mappings between UCMs and code. However, when developers make code changes to fix faults (pointed out by *FLABot*), the new code version gets out-of-sync with the UCMs. Let's call *source-code delta* the set of changes between two consecutive code versions. In the project, we noticed that these deltas usually led to updates of small UCM portions (e.g., specific responsibilities representing functions of the system). That is, the deltas exposed behavioral deviations between the architectural responsibilities and the actual code implementing the responsibilities. Based on the overlapping information between consecutive code versions (i.e., code elements that did not change) plus the mappings of the first version to the (intended) UCMs, we were actually able to review the UCMs and decide how to resolve the deviations perceived in the analysis. Nonetheless, reviewing the UCMs and related code required considerable manual efforts. The *ArchSync* tool was built out of these observations in order to facilitate (re-)synchronizations between architectural scenarios and code. Interestingly, *ArchSync* became a helpful conformance tool on its own [11].

2.1. Overview of ArchSync

We consider two basic roles in a conformance process: the architect and the reviewer. The architect specifies the architecture and updates it (if necessary), while the reviewer inspects the code for adherence to the architectural rules and communicates the results to the architect.

Fig. 1 shows these roles in the context of the *ArchSync* approach.[2] We distinguish three phases:

I. *architecture specification*, carried out by the architect;
II. *conformance checking*, carried out by the reviewer; and
III. *architecture repair*, carried out by the architect.

Phase I is the preparation for phase II, in which the tool assistance takes place. In phase I, the architect documents the intended architecture of the application with UCMs (see Section 2.1.1). Note that a Use-Case Map is not a use case but rather a description of the way components (or classes and methods) are meant to interact in response to use cases. The architect also defines mappings from the architectural components and responsibilities of the UCMs to an object-oriented implementation (tagged as version 1 in Fig. 1). The UCMs provide thus graph-based descriptions of expected system executions.

After some period of time, in which the implementation received modifications, the conformance checking begins. In phase II, the reviewer chooses test cases for a given UCM, and then exercises the current implementation (tagged as version 2 in the figure) with those test cases. In the background, *ArchSync* instruments the application code so that its behavior can be monitored. Thus, the test cases produce a set of application execution traces. The tool uses these traces to compute differences with the UCM under test (see Section 2.1.2), which are analyzed by the reviewer. Phase III starts when the reviewer passes on the differences to the architect. *ArchSync* can provide her with recommendations for (re-)synchronizing the UCMs with the code (see Section 2.1.3). On this basis, the architect will decide the right actions to restore conformance.

The conformance process using *ArchSync* is mainly intended for situations in which a stable software architecture exists (and has been documented), and the detailed design and implementation of that architecture are in progress. In this context, we envision the tool to be applied regularly (e.g., every 2 weeks), in order to get feedback on design violations (in the code) and as well as on the architecture documentation status (with respect to the code). In the former case, junior or semi-senior developers can take the results of *ArchSync* as development guidelines; while in the latter case, architects can use the *ArchSync* outputs to reflect on the "needs" for the current architecture (e.g., does the architecture need a major refactoring in the near term?). *ArchSync* is not suitable for situations in which the architecture is in constant evolution (i.e., it has not been stabilized yet) or for re-engineering projects in which the architecture is being "discovered".

In the following sub-sections, a simple User Account Management (UAM) system is used to exemplify how the *ArchSync* tool supports the three phases. Also, we describe more formally the behavioral checks supported by the approach.

2.1.1. Specifying intended architectural scenarios

First of all, *ArchSync* requires a description of the architectural structure, main functions and behavioral scenarios. The UML notation appears to be a natural selection for this matter. Nonetheless,

[2] Developers can play the role of helpers to the architect (e.g., definition of mappings), or to reviewer (e.g., execution of test cases on the application).

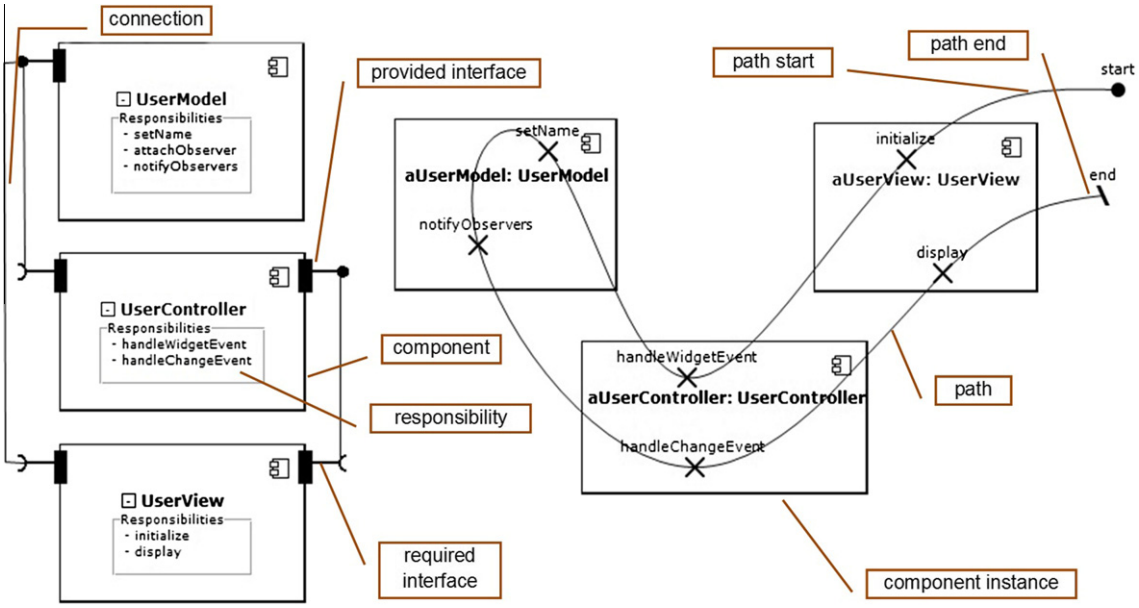

Fig. 2. Example of component diagram (left) and UCM diagram (right).

it has been argued [19,20] that the UML behavioral diagrams (e.g., Sequence, Collaboration or Activity diagrams) are suitable for describing detailed interactions (such as messages exchanged between objects) that do not always show the architectural responsibilities and components involved in a high-level scenario. Unlike UML, UCMs provide mechanisms to describe the system behavior at higher level of abstraction than messages between components [12]. We leverage on this feature to implement our conformance approach.

A UCM models a set of scenarios by means of responsibility paths that cut across design structures. The core elements of the notation are: responsibilities, paths, components, and couplings among paths. A responsibility is a function, activity or action that a component has to perform. A component is a unit of computation and state, and also a container of responsibilities. A path is a progression of responsibilities that are connected by means of causal relationships. Causal means that these relationships link causes to effects between responsibilities. A coupling connects paths according to specific patterns (e.g., fork/join nodes, preconditions, stubs, etc.). UCMs are derived from informal requirements or from use cases. Responsibilities need to be inferred from these requirements. For more details about the UCM notation, please refer to [21].

The UCMs are specified using an Eclipse-based editor that is part of *ArchSync*. This editor supports both UCM diagrams and UML2 component diagrams.[3] The architect creates component diagrams to have a structural picture of the component types involved in the design. She allocates functionality by adding responsibilities to components. These components and responsibilities are later used in UCM diagrams, as the architect specifies the behavioral scenarios of the system.

Fig. 2 shows an architectural specification fragment for our UAM system. The system is organized around an MVC pattern [17], in which the *UserModel* component and its *UserView* components are decoupled by means of a *UserController* component. This structure is depicted in the component diagram on the left of the figure. A UCM scenario is shown on the right. This scenario starts with responsibility *initialize* (in *UserView*). Responsibility *handleWidgetEvent* translates the events from *UserView* instances to *setName* (in *UserModel*), which subsequently activates *notifyObservers*. When a notification is sent from *UserModel* to *UserController*, responsibility *handleChangeEvent* gets activated, and this finally triggers the refresh of *UserView* by executing *display*. Note that a UCM scenario is more abstract than a UML sequence diagram, in the sense that a responsibility path is oblivious of how the components actually implement the path via message exchanges. For example, the UML sequence chart in Fig. 3 shows an implementation of the UCM given in Fig. 2. Note that a UCM responsibility is realized by methods or functions at the code level. Alternative implementations for the same UCM are possible, as long as the causal flow of responsibilities is ensured.

ArchSync supports two kinds of relationships between architecture and code, namely: (i) mappings of components to Java classes, and (ii) mappings of UCM responsibilities to Java methods.[4] Normally, the architect has knowledge about these mappings, so she will specify them using the editor. A variety of mapping relationships between architecture and code are possible. In practice, it is a good idea to exploit "regularities" in the mappings, such as coding policies [8,22]. For example, a policy can be that each component always maps to one principal Java class (or Java interface), which may rely on other subsidiary classes to realize the responsibilities of the component. When coding policies are used judiciously to reduce complexity, many mappings end up in one-to-one or one-to-many cases. A set of MVC mappings is depicted in Fig. 4 (top-left side). For instance, the *UserModel* component maps to classes *UserModel* and *Observable* with methods for responsibilities *setName, attachObserver* and *notifyObservers*.

2.1.2. Tracking responsibilities in execution traces

Let's assume that the MVC design has been correctly documented and coded. Then, a new requirement arrives stating that all the user's operations should be printed to a console view. Let us consider that a developer inadvertently implements this feature as a direct call from class *UserModel* to a *LogView* class. This code deviation breaks the MVC architectural rules, because *LogView* should have been instead implemented as an observer of *UserModel*. Since the

[3] http://www.agilemodeling.com/artifacts/componentDiagram.htm.

[4] For a responsibility mapping to a method to be valid, the component the responsibility is allocated to should have been previously mapped to a class defining the corresponding method.

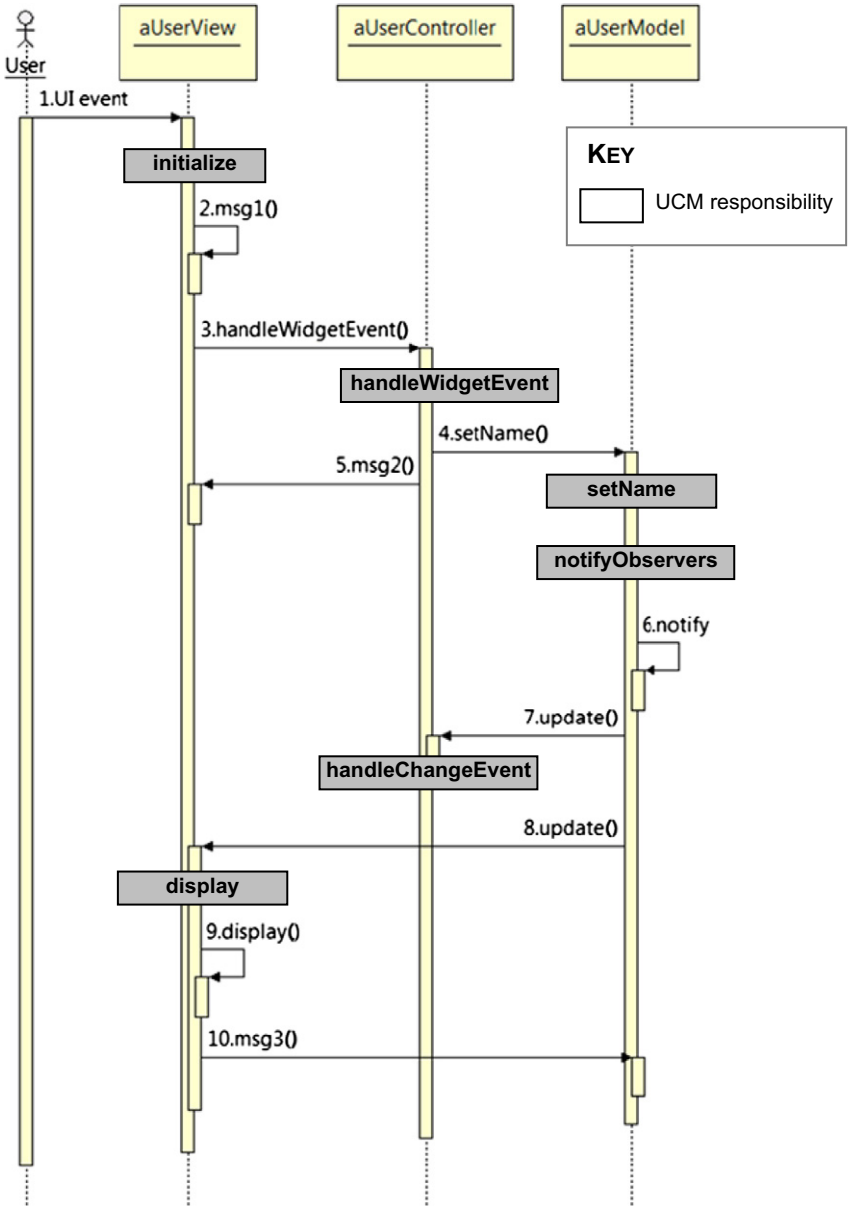

Fig. 3. Possible Message Sequence Chart realizing the UCM of Fig. 2.

change is not reflected back in the architecture, the UCM of Fig. 2 becomes inconsistent. At some point, a reviewer wants to check conformance to the MVC design (as captured by the intended UCMs), and *ArchSync* assistance comes into action.

The reviewer asks *ArchSync* to obtain the source-code delta since the last conformant version. The tool detects a change in file *UserModel.java*, and reports that method *setName()* was changed. As this method had a mapping to responsibility *setName* (in component *UserModel*), the tool marks the UCM of Fig. 2 as potentially affected by the change. The reviewer then launches an instrumented version of the UAM application and runs test cases for that UCM. In this exercise, *ArchSync* produces a log of execution traces that contain runtime events emitted by all the classes/methods with mappings from the UCM responsibilities. The tool applies a *reconstruction procedure* on this log, which transforms the traces into a sequence of high-level events called *responsibility activations*. Fig. 4 (bottom-left side) shows possible responsibility activations extracted from the log for our MVC scenario. A responsibility activation means that, according to the logged evidence, the code for that responsibility has executed.

2.1.3. Assessing differences between the UCMs and log

Once the log is processed, the reviewer asks *ArchSync* to run a *correlation procedure* that checks whether the responsibilities of the target UCM match the responsibilities inferred from the log. From a conformance perspective, this procedure implements the differencing between the intended and actual responsibility graphs. In our UAM example, when the tool traverses the path of responsibilities in the UCM of Fig. 2 against the list of responsibility activations in Fig. 4, it will find a mismatch at responsibility *LogView.addLogEntry*. This unexpected responsibility indicates that some piece of code, after responsibility *UserModel.setName*, has deviated from the intended MVC scenario. To help decision-making, *ArchSync* will search for possible repairs for the target UCM. These repairs, so-called *update scripts*, are operations to re-arrange problematic UCM paths in such a way the differences can be resolved. Fig. 5 shows two scripts outputted by the tool for the UAM example. The scripts A and B propose extensions of the problematic UCM path with a new component and responsibility. To restore conformance, the architect picks a suitable script for the problem and applies it on the UCM.

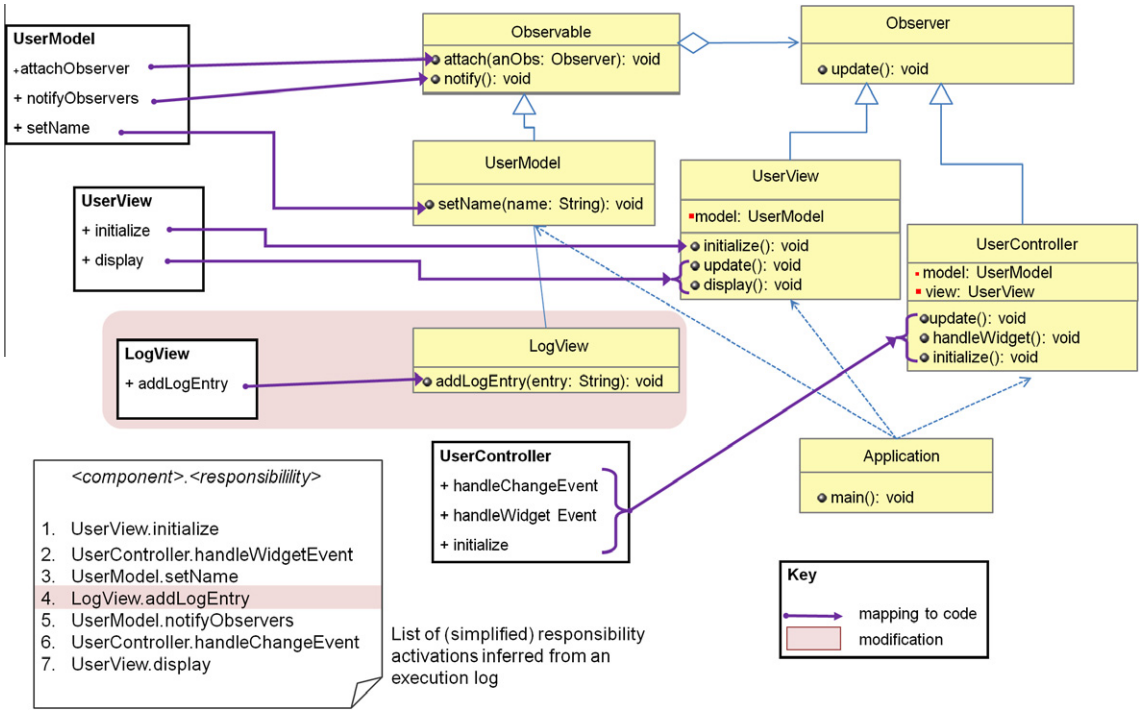

Fig. 4. Mappings from MVC to code (top), and example of responsibility activations (bottom-left).

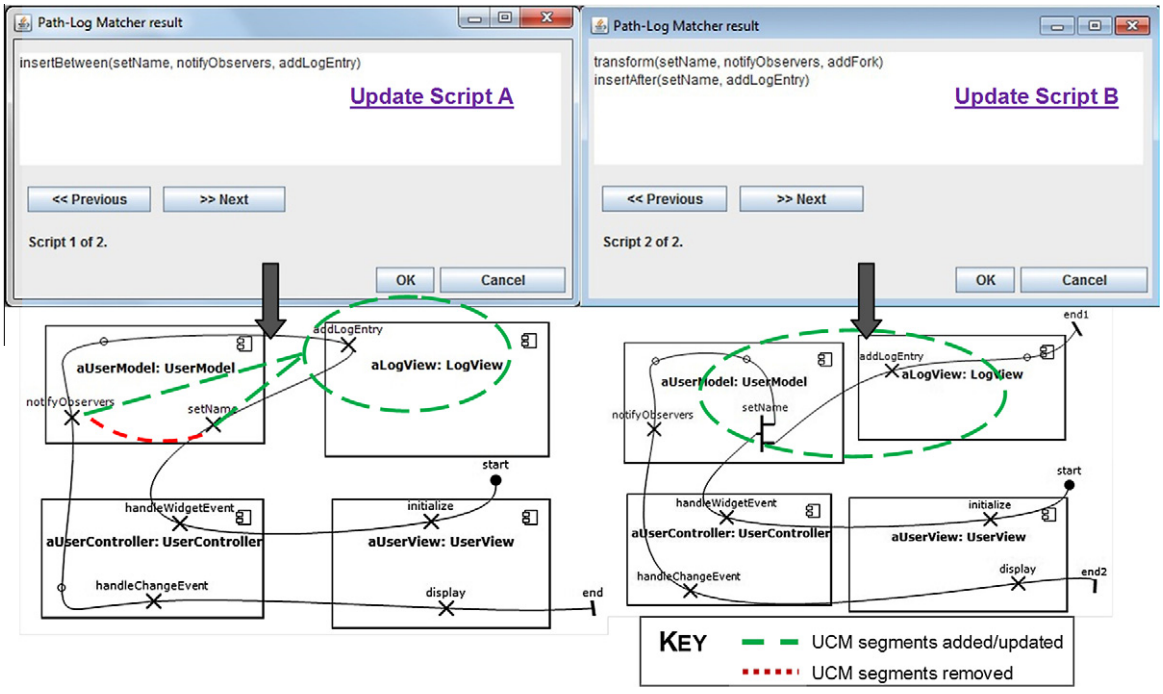

Fig. 5. Two alternative update scripts for repairing an inconsistent UCM.

Note that a given sequence of responsibility activations might correspond to alternative UCM paths at the architectural level. In Fig. 5, the two UCMs resulting from scripts A and B are both matches for the observed execution trace. The architect can execute either script A or script B to accommodate the code change, considering that the developer's implementation was motivated by performance issues; or she can turn down the *ArchSync* suggestions and ask a developer to fix the code so as to meet the MVC architecture. In the general case, *ArchSync* cannot detect all the differences between the UCMs and the log, nor can it produce all alternatives scripts for fixing the (detected) mismatches. Furthermore, a proposed script might compromise the architectural integrity (e.g., the solution for a given mismatch in one part of the MVC is different from the solution for another, similar mismatch) or some quality-attribute properties (e.g., the addition or re-connection of responsibilities changes the performance characteristics of the design). For

these reasons, the mismatches and scripts produced by *ArchSync* are informative, but the architect must act upon them according to the design context.

2.2. Formalizing conformance checks for UCMs

From a conformance perspective, a given UCM scenario represents the intended architectural behavior, while a list of responsibility activations (derived from execution traces) represents an instance of the implemented behavior that must match the UCM scenario. A more formal discussion of these concepts is provided below.

Definition 1 (*Architecture*). An *Architecture* is a tuple <*ArchCompTypes*, *ArchResps*, *defined-by*> that provides a static system view, where:

- *ArchCompTypes* is a set of component types for the system,
- *ArchResps* is a set of responsibilities, and
- *defined-by* is a function that relates a responsibility to its defining component type.

For simplicity, the component types and responsibilities are assumed to have unique names.

Definition 2 (*UCM scenario*). A *UCMScenario* is a tuple <*ArchComps*, *RespAllocations*, *instance-of*> that provides a behavioral architecture view, where:

- *ArchComps* is a set of component instances (i.e., the components touched by the UCM),
- *RespAllocations* is a graph with nodes of the form (*componentInstance*, *responsibility*) and edges between nodes that model the relationship "must execute before" (i.e., a precedence relationship along the UCM path), and
- *instance-of* is a function that relates a component instance to a component type (i.e., its parent).

A UCM scenario has special nodes, such as: one start node, one or more end nodes, and it might include fork/join nodes with specific semantics associated to those nodes. Note that the behavior (to be checked) is determined by the whole path of responsibilities, rather than by the individual responsibilities. The component types that result from applying function *instance-of* as well as the responsibilities in the graph are all included in *Architecture* (see Definition 1).

Definition 3 (*Java implementation*). An *Implementation* is a tuple <*Classes*, *Methods*, *part-of*> modeling the code view, where:

- *Classes* is a set of Java classes of the system (with unique names),
- *Methods* is a set of Java methods for those classes, and
- *part-of* is a function that relates Java methods with its defining classes.

Definition 4 (*Architecture-implementation mapping relation*). The mappings are formalized as a partial relation between implementa tion elements and architectural elements. An implementation element is a pair (*class*, *method*), while an architectural element is a pair (*componentType*, *responsibility*). Thus, *abstractionMapping* is a relation (*Classes* × *Methods*) → (*ArchCompTypes* × *ArchResps*), where:

- sets *Classes* and *Methods* are those in *Implementation* (see Definition 3), and
- sets *ArchCompTypes* and *ArchResps* are those in *Architecture* (see Definition 1).

We say that *abstractionMapping* is partial because for some implementation elements there is not always a correspondence with an architectural element. Conversely, an architectural element can relate to one or more implementation elements. Note that the mapping relation is based on a static view of both the architecture and the implementation.

Definition 5 (*Execution trace*). An *ExecutionTrace* is a sequence of *RuntimeEvent,* where *RuntimeEvent* is a (runtime) event with the structure (*class*, *object*, *methodCall*). For each method call, the event holds information about the object executing the method (i.e., the receiver) and the class for that object. All the classes and methods mentioned in an *ExecutionTrace* belong to the *Implementation* (see Definition 3).

Definition 6 (*Responsibility-activation trace*). A *RespActivationTrace* is a sequence of *ArchEvent*, where *ArchEvent* is an (architectural) event with the structure (*componentType*, *responsibility*). All the component types and responsibilities mentioned in *RespActivationTrace* belong to *Architecture* (see Definition 1).

Definition 7 (*Reconstruction procedure*). The reconstruction procedure is a function *reconstruction*: *ExecutionTrace* → *RespActivationTrace* that bridges between runtime events and architectural responsibilities. The function relies on the mappings in order to transform an execution trace into a list of responsibility activations. That is, each *archEvent* in *RespActivationTrace* is constructed by applying *abstractionMapping* on some *runtimeEvent* in *ExecutionTrace*.

For example, let (*class1*, *obj1*, *methodcall1*) be the first event of an *ExecutionTrace*, then reconstruction (*class1*, *obj1*, *methodcall1*) = (*compType1*, *resp1*) if we can find a pair ((*class1*, *method1*), (*compType1*, *resp1*)) in relation *abstractionMapping* and it also happens that *methodcall1* is the invocation of *method1*.

Note that the reconstruction procedure does not recognize component instances but component types from the execution trace.

Definition 8 (*Correlation procedure*). Let *US* be a given UCM scenario and *RA* be a responsibility-activation trace, the correlation procedure seeks to determine whether *RA* matches the behavior in *UC* (assuming that *RA* is the result of the reconstruction procedure on some execution trace). More formally, let *sequences(UC)* be the set of all possible end-to-end sequences for the UCM scenario. That is, each sequence begins at a start node, and traverses the UCM until reaching any of its end nodes, while also satisfying the constraints of fork/join nodes (if applicable). Let *sequences_sub(UC)* be the same set of sequences, in which every occurrence of a component instance is substituted by the corresponding component type (applying function *instance-of*, from Definition 2). Then, the conformance criterion is that *RA matches UC* if *RA* ∈ *sequences_sub(UC).* In case of several responsibility-activation traces for the same UCM scenario, we test the criterion iteratively on each of traces.

Definition 9 (*Update script*). If a responsibility-activation trace does not match the target UCM scenario, we can identify a set of points (i.e., responsibilities) of the UCM that led to the mismatch. This is where the notion of update script comes into play, as a transformation on these mismatching points that produces a (modified) UCM path that satisfies the activation trace. More formally, let *P* be a set of mismatching responsibilities within a

UCM scenario *US*. If *US* fails to correlate with an activation trace *RA* (in the terms of Definition 8), we search for an update script *T* such that $T(P, US) = US'$ and *RA* matches *US'*. In general, alternative update scripts for the same mismatch exist.[5]

Some mismatches are straightforward to deal with, such as an unexpected responsibility that can be fixed with a script that simply adds (or removes) the responsibility in the UCM path. In case of simple mismatches, the update scripts are actually "local" transformations to the UCM. Other mismatches can be complex, such as detecting a different responsibility ordering between the UCM and the activation trace. In the latter case, transformations affecting the whole UCM need to be considered.

2.3. Challenges and assumptions

In *ArchSync*, we have made several assumptions that respond to challenges for checking behavioral conformance in practice. First of all, our approach requires: (i) a consistent UCM specification of the application's main scenarios, (ii) the availability of source code implementing that application, and (iii) an initial degree of conformance between the UCMs and the implementation. A challenge here is the representation gap between modeling architectural behavior as scenarios and implementing these scenarios in a programming language. The UCMs specify partial snapshots of behavior rather than all the possible system behaviors. When implementing the UCMs by means of object-oriented constructs, a variety of mappings exist but no mapping relation is "perfect". It is the architect's job to find a mapping relation with adequate granularity (e.g., determining the right Java methods a responsibility should map to) and completeness (e.g., all responsibilities should map to, at least, one method) for a given application. The *ArchSync* editor can check basic consistency rules between UCMs and component diagrams, as well as rules for their mappings to code. Nonetheless, it should be kept in mind that the UCM scenarios do not imply strict execution flows in the mapped code; neither do these scenarios cover all the possible interactions between (Java) objects.

A second set of assumptions concerns the analysis of responsibility activations in execution traces. We assume that the implementation is executable, and that the test cases will provide us with runtime information for exposing UCM deviations. It is also assumed that, given a UCM scenario, the reviewer knows how to test application functions that correspond to that UCM. As several authors have pointed out [8,16,23], relating low-level runtime events to "architectural states" is a challenge. A first issue is that the same event (e.g., a method call) can provide hints for several architectural elements (e.g., responsibilities). A second issue is that events collected via code instrumentation vary between executions. A related problem is that the mappings might erode over time [25]. To alleviate this problem, we assume that the synchronization between UCMs and code happens periodically, so that the source-code delta contains few design changes. Finally, detecting all possible mismatches for a UCM-based scenario or exploring the whole space of architectural fixes for it (as formalized in Section 2.2) is not computationally feasible for *ArchSync*. The rationale for having a search for local UCM repairs follows from the above assumptions about mappings, execution traces, and frequent (re-)synchronizations.

3. Design of the tool

The *ArchSync* tool is designed as an Eclipse plugin that interacts with a code repository, a logging system, and a graphical UCM editor. Fig. 6 depicts the *ArchSync* infrastructure that follows a Pipe-and-Filter architectural pattern [17]. One key goal when designing *ArchSync* was the ability to combine different strategies (e.g., code instrumentation, detection of responsibility activations, diff between UCM and log representations, inference of UCM fixes) for the purpose of conformance. A clean separation of these functionalities was the main driver for choosing the Pipe-and-Filter pattern. Filters *DiffMapper* and *LogAnalyzer* can execute without user's intervention, whereas filters *ExecutionLogger* and *PathLogMatcher* involve user's intervention to do their processing. In the former filter, the user is responsible for launching the instrumented application with the test cases. In the latter, the user is expected to act on the UCM differences and update scripts proposed by the tool.

The first filter in the pipeline is *DiffMapper*. This filter receives a set of UCM diagrams along with the last synchronization date from the editor. *DiffMapper* also obtains from the repository (via the Eclipse local history) the code sections that have changed since the last UCM synchronization. This information is used to determine the source-code delta that contains the modified Java classes and methods. The filter tags the UCM paths that could be affected by changes within the delta. These UCM paths come from examining the original UCM mappings against the elements of the delta, as exemplified in Section 2.1.2. Both the affected UCM paths and the source-code delta are passed onto the *ExecutionLogger* filter.

Filter *ExecutionLogger* performs a selective instrumentation of the Java methods in the source-code delta. The instrumentation is based on an infrastructure of probes, as described in [23]. We implemented this infrastructure with the *Javassist*[6] toolkit that permits Java bytecode manipulation at load-time. Each UCM responsibility is associated to a probe that monitors specific method executions, based on the mappings of that responsibility to methods. As the test cases for the affected UCM paths are exercised, the probes collect runtime events of interest such as method calls, object creation, threads, and exceptions. In the end, the *ExecutionLogger* produces execution traces that summarize all these events. We refer to these traces as the *execution log*. The working of the *ExecutionLogger* (along with the *LogAnalyzer*) bears similarities with the scenario-based dependency analysis discussed in [25].

The remaining filters, *LogAnalyzer* and *PathLogMatcher*, are the core of the conformance approach. Basically, the UCMs are checked for conformance by (i) observing the execution of the source code and (ii) reasoning as to whether the expectation (as depicted in the UCMs) matches the actual system. These two functions are implemented by the reconstruction and correlation strategies respectively, which are explained next.

3.1. The reconstruction procedure

The *LogAnalyzer* filter is an implementation of the reconstruction procedure, as specified in Section 2.2 (Definition 7). The filter is equipped with a heuristic for inferring responsibility activations (like those in Fig. 4) from logged application behavior related to a given UCM. The heuristic used by the *LogAnalyzer* is described in Fig. 7. Since *ArchSync* does not have mapping information relating component instances to Java objects, the reconstruction performs a conservative approximation that links the objects to the component types enclosing their Java classes. Note also that responsibility activations are approximate because the logged events depend on particular application executions.

The *log-analysis()* algorithm processes the log entries one at a time sequentially. For each entry, i.e., a method call, the algorithm considers the candidate UCM responsibilities mapped to that method (line 5) and can create a responsibility activation (as

[5] The architect/developer might even decide to apply no scripts on the UCM and fix instead the implementation (that is, the source code that generates the activation traces).

[6] Javassist homepage: http://www.csg.is.titech.ac.jp/~chiba/javassist.

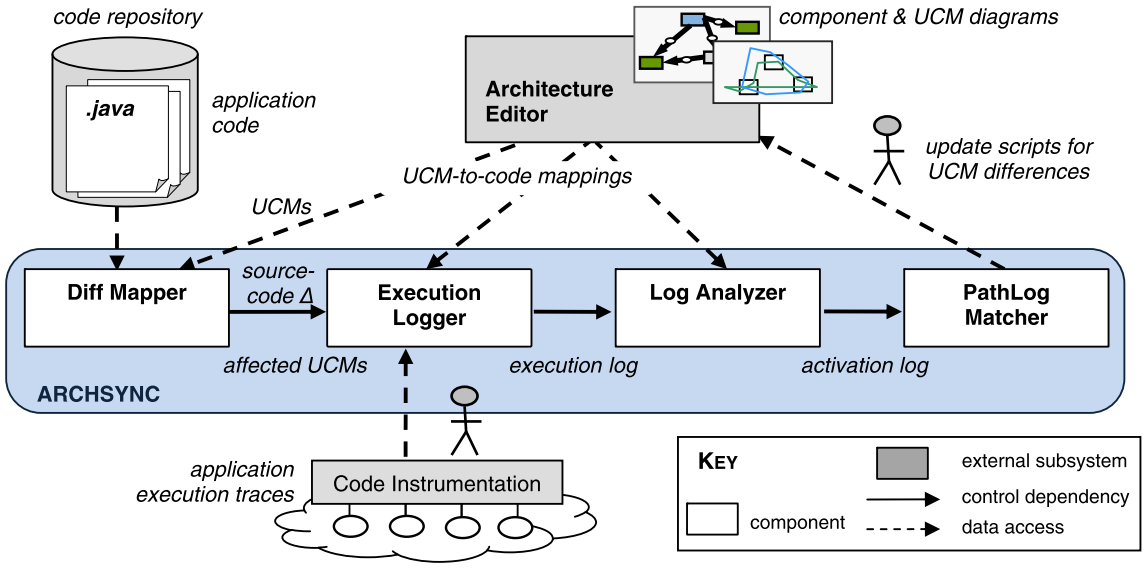

Fig. 6. The *ArchSync* architecture.

```
1.  Algorithm: log-analysis(executionLog: Seq LogEntry, mappings: UCMMappings) → Seq Responsibility
2.      activationsSequence := Ø // output
3.      lastActivation := null
4.      foreach entry in executionLog
5.          candidateResponsibility := mappings.getParents(entry) // Architecture reconstruction
6.          if (lastActivation == null) or (candidateResponsibility • lastActivation)
7.              lastActivation := candidateResponsibility  //  We have a new responsibility activation
8.              // The activation is appended to the output
9.              activationsSequence := activationsSequence + candidateResponsibility
10.         end-if
11.         // If the responsibility is the same that the last one appended, it is not considered for the output
12.     end-foreach
13. return (activationsSequence)
14. end log-analysis
```

Fig. 7. Pseudo-code of the reconstruction procedure.

introduced in Section 2.1.2). Initially, the current activation is the responsibility obtained from the first log entry (lines 6–7). This activation is appended to the output (line 9). If consecutive log entries receive mappings from the responsibility just processed, those entries are discarded because they do not contribute new activations (line 11). Otherwise, when a different responsibility is found, the current activation is updated and appended to the output. When all log entries have been consumed, the algorithm outputs a sequence of responsibility activations, or *activation log*. Unlike the log generated by the *ExecutionLogger*, the activation log is an architecture-level trace representation. This representation enables us to correlate the responsibilities just extracted and the responsibilities in the intended UCM. The *LogAnalyzer* filter yields control to the *Path-LogMatcher* filter that performs this correlation.

3.2. The correlation procedure

The *Path-LogMatcher* filter implements the *correlation procedure*, as specified in Section 2.2 (Definition 8). The idea is to progressively match a target UCM against an activation log, in order to identify deviations in the UCM paths. Furthermore, we try to patch a deviation as soon as it is discovered. That is, if a log entry cannot be matched with any UCM responsibility, we search for UCM modifications that will make the UCM path "adapt" to the activation log. The correlation heuristic used by the *Path-LogMatcher* is described in Fig. 8.

The *path-log-matching()* algorithm traverses the input UCM using a set of cursors that keep track of the responsibilities being correlated with entries (i.e., responsibility activations) in the activation log. The UCM is seen as a Petri Net, based on techniques developed elsewhere [8,26]. Thus, a cursor is a token that moves through the nodes and segments of a UCM according to specific rules. In our algorithm, a cursor also has a pointer to a log entry. In general, a cursor is allowed to move forward only if the UCM responsibility the cursor is at matches the log entry pointed by that cursor. This ensures that all the responsibilities that precede a given cursor have been already reconciled with log entries. The algorithm starts with one active cursor positioned at the first responsibility of the UCM path (line 2). This cursor points at the first log entry.

Within the loop, the algorithm selects a *currentCursor* from the active cursors and seeks to match the responsibility of that cursor against the responsibility in *currentEntry* (lines 6–8). Function *selectCursor()* supports different choices for the *currentCursor*, and the algorithm can backtrack to this step if the correlation reaches a dead-end. If the matching succeeds (lines 8–9, in if branch), *currentCursor* is advanced to the successor node in the UCM path, and

```
1.  Algorithm: path-log-matching(path: UCMPath, activationLog: Seq Responsibility) → Seq UpdateScript
2.      cursors := { <path.initialNode, activationLog.first> }
3.      recommendedRepairs:= Ø   // Output scripts that will contain the UCM modifications to restore conformance
4.      while  not (allCursorsAtEnd(UCMPath) and allCursorsPointingAtEnd(activationLog)) begin
5.          // Note that functions involving backtracking are in bold below
6.          currentCursor := selectCursor(cursors) // backtracking point
7.          currentEntry = currentCursor.^pointerToLog;
8.          if matches(currentCursor.responsibility, currentEntry) // successful matching
9.              cursors := advanceCursor(path, currentCursor, cursors)
10.         else // matching failed, so a path repair is needed
11.             script := selectAction(path,currentCursor,activationLog)  // backtracking point
12.             path := refinePath(path, script)
13.             cursors := updateCursors(path, cursors, script)
14.             recommendedRepairs = recommendedRepairs + script
15.         end-if-else
16.     end-while
17.     return (recommendedRepairs)
18. end path-log-matching
```

Fig. 8. Pseudo-code of the correlation procedure.

Table 1
Rules for moving cursors in UCM nodes.

UCM node type	Rules
Plain responsibility	Move the cursor forward to the next node
AND fork	Create a child cursor for each of the branches of the UCM main path Then, assign each child cursor to the next node of its corresponding branch
OR fork	Select one branch of the main path, and move the cursor forward to the next node in that branch (the cursor can explore other alternative branches if needed, via backtracking)
AND join	When a cursor arrives to the node, it must wait for the cursors from the other incoming branches. Once all cursors reach the node, they are merged into a single cursor that continues to the next responsibility in the outgoing branch of the join
OR join	The first cursor that reaches the node advances to the next responsibility (in the outgoing branch of the join). If the join is reached by cursors coming from different incoming branches, a warning is shown to the user
End	Delete the cursor

Table 2
Examples of repairs used in update scripts.

Corrective action (for the UCM)	Description
removeNode(*node*)	An responsibility (*node*) is deleted from the path
insertBetween(*node1*,*node2*,*newNode*)	A new responsibility (*newNode*) is inserted in between the responsibility with the current cursor (*node1*) and its predecessor in the path (*node2*)
insertAfter(*targetNode*,*newNode*)	A new responsibility (*newNode*) is inserted after the responsibility with the current cursor (*targetNode*)
transform(*targetNode*,*newType*) *transform*(*targetNode*,*nextNode*, *newType*)	A responsibility node (*targetNode*) with a given type (e.g., fork, join, or plain responsibility) is assigned to a new type (*newType*). For some node types, the successor node in the path (*nextNode*) is required

the algorithm goes back to the beginning of the loop (line 4). Function *advanceCursor()* controls the movement of cursors over the UCM paths. As long as there are no mismatches, the normal flow of the algorithm will increase or decrease the number of active cursors, depending on the UCM node type being visited. Table 1 summarizes the rules for moving cursors across a UCM. The cursors will reach the end node of the UCM only if all responsibilities have been successfully matched with counterparts in the activation log.

However, if a responsibility matching fails, it means that an unanticipated responsibility was encountered in the log (line 10, in else branch). For instance, this mismatch happened with *LogView.addLogEntry* in Fig. 4. This kind of mismatches trigger modifications to the UCM path, which constitute the update scripts returned by the algorithm (lines 11–12, in else branch). After selecting a *script*, the algorithm applies it to the current UCM path and also makes the cursors move to the closest point of path synchronization (lines 13–14, in else branch). For a successful termination of the algorithm, all cursors must be at the end of the UCM, and furthermore, all the pointers of these cursors should be at the end of the log.

Upon request of the user, *Path-LogMatcher* returns alternative scripts for synchronizing a UCM. In Section 2.2 (Definitions 8 and 9), we mentioned that there are several options for repairing a UCM. However, the filter does not consider all possible scripts; it rather evaluates "local" UCM transformations that might solve the mismatch. To do so, the correlation algorithm relies on two backtracking points provided by functions *selectCursor()* and *selectAction()*. Once a mismatching point is found, *selectAction()* tries basic repairs for the UCM nodes and segments related to that mismatch. Examples of repairs are listed in Table 2. Specifically, function *selectAction()* is a trial-and-error search for appropriate instantiations of those repairs. Prolog rules were used for implementing this search, as well as for implementing cursor choices in function *selectCursor()*. We codified the *path-log-matching()*

algorithm in *JavaLog*[7] [27], a framework for developing Java programs that integrates object-oriented features with a Prolog engine. To constrain the search for scripts, a heuristic "distance criterion" between scripts was employed. This criterion prefers scripts that apply few corrective actions on a UCM, considering that such scripts will more likely preserve the main flow (i.e., the intent) of that UCM. The Prolog engine is configured with a threshold for the maximum UCM changes that a script can make, and partial scripts with changes greater than this threshold are discarded by the algorithm. The threshold value was empirically set to a (maximum) distance of three basic actions. This threshold acts as a control parameter for function *selectAction()*, but has no effects on the detection of UCM mismatches.

4. Case-studies

In previous work [11], we found evidence that *ArchSync* can help architects when searching for architectural or implementation elements that might need conformance revisions. In order to validate these claims, we performed a retrospective study of three software projects and exercised the *ArchSync* tool. Our objective is to evaluate whether *ArchSync* is useful for checking conformance to UCMs, given a series of implementation changes in the system.

The experiments were conducted according to the reference three-phase conformance process outlined in Section 2.1. For each case-study, phase I (architecture specification) was simulated by looking at historical changes in the configuration management repository, and selecting a number of changes that were architecturally-significant. By architecturally-significant, it is meant changes that have impact in the architectural design, and therefore, in the architectural documentation. Then, we sketched a sequence of versions and replayed the changes from one version to the next one. For the remaining phases, we carried out two types of experiments: one supported by *ArchSync*, and another one without the tool. In phase II (conformance checking), we have two main activities: identification of the source-code delta and the UCMs potentially affected by the changes in that delta (called activity A), and reasoning about code deviations with respect to the intended UCMs (called activity B). In phase III (architecture repair), the main activity is the re-synchronization of UCMs and code (if necessary). The participants in these experiments were developers with moderate domain experience and with some degree of involvement in the architectural design of the systems.

We defined two performance criteria: (i) the effort of activities A–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 the user and the savings when she explored UCMs and Java classes for inconsistencies (relative to browsing all relevant cases). In manual checks, these savings come from the participant's domain knowledge. That is, a human usually knows the "semantics" of a change and can avoid browsing unnecessary components or classes. In *ArchSync*-supported checks, the tool can guide the participant to inspect certain components or classes (and avoid others). To compute the *ArchSync* precision, we used the standard Information Retrieval metric based on false/true positives/negatives. In addition to the two quantitative criteria above, we qualitatively reported if *ArchSync* was able to identify design violations overlooked by the architects of the three projects.

4.1. Case studies

The case-studies came from different domains, namely: an academic project called *Universidad3D*, a commercial project called *InQuality*, and a government project.

[7] JavaLog homepage: http://www.exa.unicen.edu.ar/isistan/javalog.htm.

Table 3 summarizes the characteristics of these three projects, which are shortly described below.

4.1.1. Case-study 1: Universidad3D

This is a 3D educative game[8] 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 with features for scene definition, animation and navigation. *Universidad3D* is designed as a multi-tiered client–server architecture supporting 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. For the experiments, we scoped our modeling and analysis to the architectural design of the server, where most of the gaming functionality resides. The server's functions include: processing requests from the clients, executing the 3D game logic, and data management.

4.1.2. Case-study 2: InQuality

This is a commercial Web-based platform for Enterprise Quality Management systems[9] developed by the Analyte company. Basically, *InQuality* is a framework for managing and keeping consistent the corpus of documents and contents of an enterprise. The core includes functions for: documentation control, auditing, automated computation of indicators, tracing and monitoring of activities. *InQuality* has a client–server architecture that consists of 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 framework that is internally implemented as a multi-agent system. In particular, we modeled a sub-system that deals with the organizational structure and the edition of documents.

4.1.3. Case-study 3: A Government Organization

This project[10] involved the migration of a legacy system of an Argentine Government Organization to a Web-services platform. This platform is intended to support several applications handling social security and employment data of citizens. The (new) system is designed as a service-oriented architecture (SOA) with C# and Java implementations. For our experiments, we selected three Web services of the platform, each supporting several operations. A technological aspect of this system 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. As a result of this tooling issue, not all the changes could be properly captured and processed by the tool.

4.2. Setup

For each project, we selected two developers as participants, one responsible for the experiments without *ArchSync* (also referred to as "manual" experiments), and the other responsible for the experiments supported by *ArchSync*. Fig. 9 summarizes the activities involved in the two kinds of experiments. Note that a participant switches between the roles of architect → reviewer → architect, as she moves along the activities of setup → A–B → C (respectively) in the workflow.

In preparation for the experiments, we held a training course for the participants so as to provide them with the skills for writing UCM

[8] Universidad3D homepage: http://isistan.exa.unicen.edu.ar/u3d/index.htm.

[9] Analyte homepage and InQuality information: http://analyte.com/pdf/inQuality.pdf.

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

Table 3
List of case-studies and their main characteristics.

Project name	Type of project	Domain	Implementation	Subset used in experiments	Architecture
Universidad3D	Academia	Education, 3D game (multi-player online game)	Java, ~370 classes, ~25 KLOC	Java, ~190 classes, ~13 KLOC	Multi-tier client–server, 3D engine
InQuality	Industry	Lab information management system, process quality control	Java, ~1096 classes ~67 KLOC	Java, ~311 classes ~21 KLOC	Web client–server, event-based system
Government Organization	Government	Management of social security and employment data for individuals	Java and C#, 1324 classes ~89 KLOC	Java and C#, ~450 classes, ~34 KLOC	Web services, service-oriented architecture

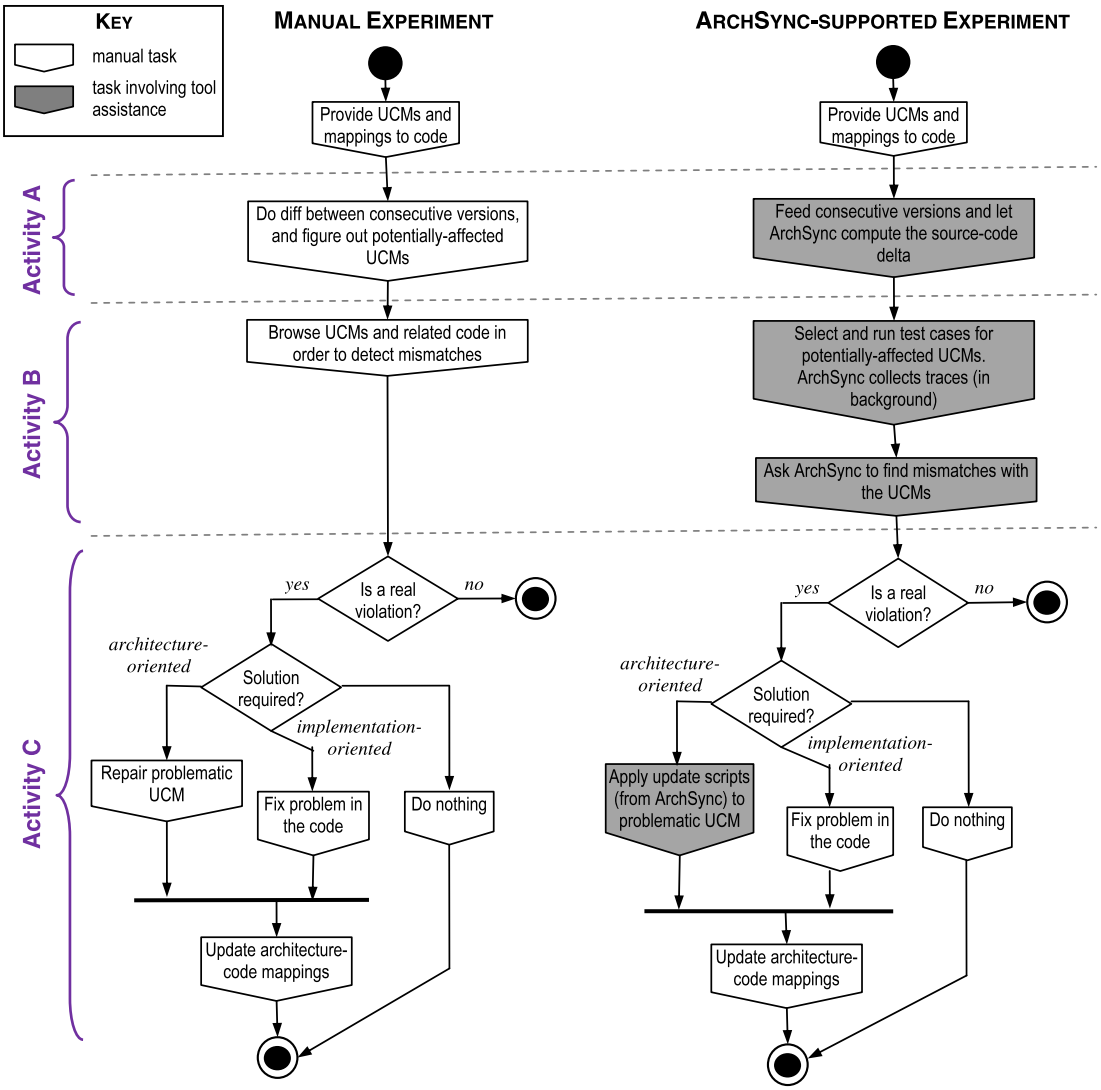

Fig. 9. Activities followed by the participants in (a) the manual experiment and (b) the *ArchSync* experiment.

scenarios and mapping UCM elements to object-oriented code. Then, the participants responsible for each case-study reviewed the project documentation (e.g., reports, UML diagrams, release notes, and ultimately Java code) and made adjustments in order to build an initial version of the UCMs and their mappings to code.

For the manual experiments, activities A–C were as follows. First, the participant was asked to interpret the diff file between consecutive versions in order to figure out the main changes (i.e., the source-code delta). She also had to manually figure out which UCMs were related to those changes (activity A). Second, she was asked to browse the UCMs and code to find deviations (activity B). Third, once deviations were identified, she had to decide on the best fix for them (activity C). In background, we used the *Mylyn* plugin[11] to monitor the classes browsed and time spent by the participant on each activity. The analysis was complemented with the *Metrics* plugin,[12] which provided us with code metrics about the deltas.

For the experiments with *ArchSync*, activities A–C were as follows. First, the participant was asked to feed a pair of consecutive versions into *ArchSync* and let the tool compute the delta (activity A in reference process). Second, the participant had to exercise tests in order to generate application traces. Third, she asked the tool to

[11] 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.

[12] Metrics homepage: http://metrics.sourceforge.net/.

Table 4
Sequence of versions (and architectural features) in case-study #1 (*Universidad3D*).

Version	Architectural features added or modified	Lines of code changed	UCMs affected	Architectural Elements involved
0. revision003	(architecture baseline)	–	–	–
1. revision060	Client–server communication	54	1	4 Responsibilities, 3 components
2. revision110	Player registration, game logic	108	1	1 Responsibilities, 1 component
3. revision129	Support for new players	149	2	2 Responsibilities, 1 component
4. revision184	Chat improvement	584	1	2 Responsibilities, 1 component
5. revision211	Support for new game states	167	1	2 Responsibilities, 2 components
6. revision225	Avatar selection mechanism	130	2	4 Responsibilities, 2 components
7. revision287	Avatar customization	391	1	1 Responsibilities, 1 component
8. revision562	Off-line playing mode	118	1	4 Responsibilities, 1 component
9. revision815	User registration, VoIP support	513	3	4 Responsibilities, 2 components

determine differences in the UCMs (these two steps correspond to activity B). Fourth, she had to assess the UCM scripts outputted by *ArchSync* and modify either the architecture or the code (activity C). Like in the manual experiment, we served from the *Mylyn* and *Metrics* plugins to gather measures about time and browsed classes. If any of the update scripts was a satisfactory solution to the UCMs (according to the participant), we considered that *ArchSync* was successful for that delta. Otherwise, we discarded the results. This information was useful to compute precision of the assistance.

The processing of architectural documentation varied depending on the project and the available assets. Being an academic project, the *Universidad3D* case-study included already a baseline of UCMs as part of the architectural documentation, which drove the implementation of the system. Nonetheless, the diagrams had just informal mappings to code. Here, the participants specified more precise responsibility mappings, as they performed the simulations. Building the initial architecture required around 2 h, and the modeling consisted of 7 components and 24 responsibilities. For the experiment, we selected 10 versions for the server as shown in Table 4, and processed the 9 deltas between pairs of consecutive versions.

In the *InQuality* case-study, the development team had followed an architecture-driven approach, but the existing documentation relied on UML sequence diagrams rather than on UCMs. The two participants translated those sequence diagrams to UCMs suitable for the experiments and they also specified the UCM mappings to Java classes. Building the initial architecture required around 1.5 h, and the modeling consisted of 9 components and 23 responsibilities. Like in the first case-study, we selected 10 system versions with varying features, and then processed the corresponding 9 deltas. As regards the third case-study, the UCMs had to be created from scratch. The two participants were responsible for specifying these UCMs and their mappings. This architecture modeling required around 40 min, and it consisted of 3 components and 18 responsibilities. In this case-study, we selected 8 system versions and processed the corresponding 7 deltas. Due to space reasons, the information about the versions and features analyzed for case-study #2 and case-study #3 is omitted.

To make comparisons across case-studies, it is desirable to have a consistent ratio of "coverage" between UCMs and code. For a given system version, this coverage is estimated as the number of classes (or methods) that received mappings from architectural components (or responsibilities) divided by the total classes (or methods) in that version. We set an average coverage of 50% for the three case-studies and their corresponding versions. Also, we managed to have a time of 10–12 days between consecutive versions, ensuring that the UCMs were checked at regular time periods in all case-studies.

4.3. ArchSync performance

In this section, the efforts spent by the participants and the precision of *ArchSync* across the three case-studies are analyzed. Figs. 10–12 show the evolution of time efforts per activity for all deltas, in both the manual and *ArchSync*-supported experiments. For *Universidad3D* (Fig. 10), there are blanks for delta 2 (between versions 1 and 2), delta 6 and delta 9. Although conformance issues existed in these three deltas (which were detected indeed during the manual experiments), *ArchSync* failed to suggest valid update scripts for the UCMs.[13]

Overall, the effort results show that the participant using *ArchSync* spent less (total) time for the UCM checks than the participant performing the same checks manually. In particular, we noticed speedups in activities A and B, which correspond to the conformance checking phase. In activity A, there were minor time reductions when comparing the *ArchSync* experiments against the manual ones. We found some exceptions towards the last deltas (in all case-studies), where the manual experiments performed better than the tool-supported ones. This effect for activity A is noticeable, for example, in case-study #2 (Fig. 11). For the *ArchSync* experiments, the delta processing times gradually increased, because the tool processes all the code of the versions, regardless of the changes. For case-study #2 in particular, the effect is also due to the agent-based framework behind *InQuality*, which provides well-defined guidelines for developers to follow when coding an application feature [28]. These guidelines help developers to quickly understand the design rationale of the system.

Activity B reported the most significant gain when using *ArchSync*, across all case-studies. However, we observed that the time differences between the two types of experiments decreased towards the last deltas. We believe this happened because the participants gradually became proficient in dealing with the changes. With respect to activity C, we observed different variation patterns in the case-studies but no time improvements of *ArchSync* over the manual experiments. For instance, in delta 4 and delta 7 of case-study #1, we found that the greater the time consumed by *ArchSync* the greater the number of classes involved in the delta. A similar relationship appeared in delta 2, delta 6 and delta 7 of case-study #3 (Fig. 12). For these cases, the participants reported that fixing the UCMs (with *ArchSync*) took a long time because they have to analyze components with several mappings to code (see discussion about browsing time below). Despite this issue, the UCM synchronization might trigger (re-)mappings of responsibilities, which must be set manually by the participants in both types of experiments. Thus, the time of activity C is always influenced by that manual effort.

In order to contextualize the time and browsing efforts discussed above, we estimated an "architecture change ratio" between consecutive versions. To do so, we collected statistics about the average number of components, responsibilities, Java classes, methods, and lines of code (LOC) affected by the changes within a delta. In case-study #1, the change ratios for components and responsibilities were 36% and 15% respectively. In the code mapped from those architectural elements, the change ratios for

[13] This problem was caused by an implementation bug in the correlation rules.

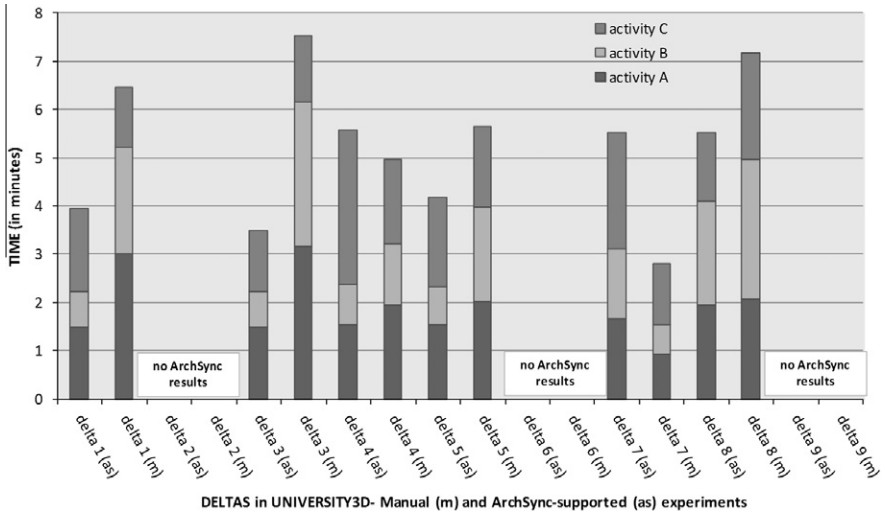

Fig. 10. Comparison of time efforts for the three conformance activities in case-study #1.

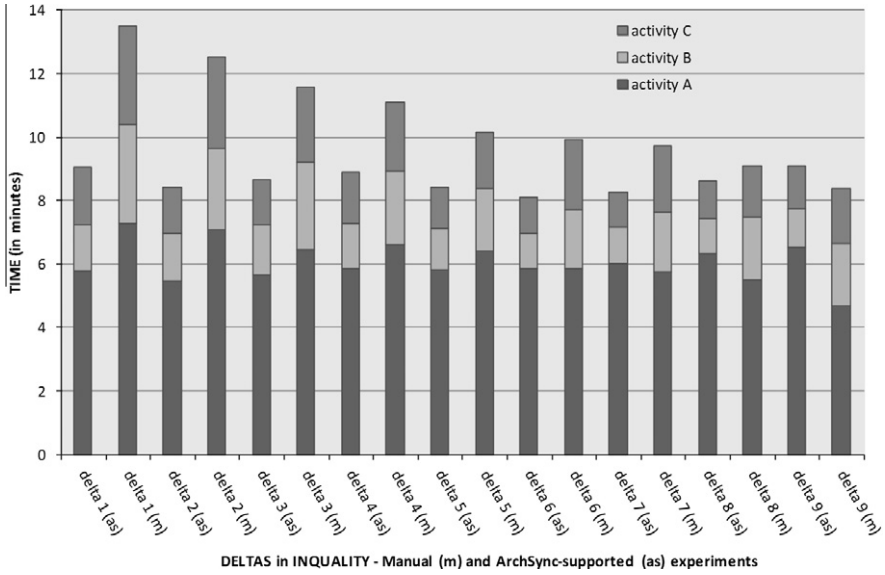

Fig. 11. Comparison of time efforts for the three conformance activities in case-study #2.

classes, methods, and LOC were 38%, 45% and 15% respectively. In case study #2, we had similar change ratios for the components and responsibilities, but the ratios for the mapped code were slightly higher. At last, in case-study #3, the change ratios for the architecture were considerable higher than in the two previous case-studies, with 60% and 73% for components and responsibilities respectively. The corresponding change ratios for classes, methods and LOC were 65%, 13% and 4% respectively. Note that the change profiles are different for the three case-studies, mainly due to the specific development characteristics of each project. Although more empirical analyses are needed, we believe that these profiles have correlation with the effort distributions observed in Figs. 10–12 (with and without *ArchSync*).

For activities A–C, we additionally computed the percentage of Java classes inspected by each participant while dealing with a conformance problem, relative to all relevant classes in the version. The effort savings were 35–42% and 60–65% in average (across the deltas of the three case-studies) for the manual and *ArchSync*-supported experiments respectively. These results show that *ArchSync* can still reduce the user's search for conformance-related classes, even when the tool lacks domain knowledge (in contrast to the participants). The number of scripts proposed for a delta was around 11–17 scripts per UCM. The precision of the assistance was computed in terms of the number of satisfactory scripts for all versions exercised with the tool (irrespective of the existence of conformance problems). The average precision for the experiments ranged between 0.72 and 0.81. We argue that these precision values are reasonable, given the heuristic nature of the tool assistance.

4.4. Sensitivity analysis for mappings

The previous experiments were based on a configuration of consistent mappings with a fixed granularity for the UCMs. In that configuration, mappings are viewed from an abstract perspective, that is, the mappings for a component or a responsibility are generally one-to-many (with respect to Java classes and methods). Since *ArchSync* is based on heuristics, it is interesting to analyze whether the outputs of the tool are affected by variations in the architecture-code mappings.

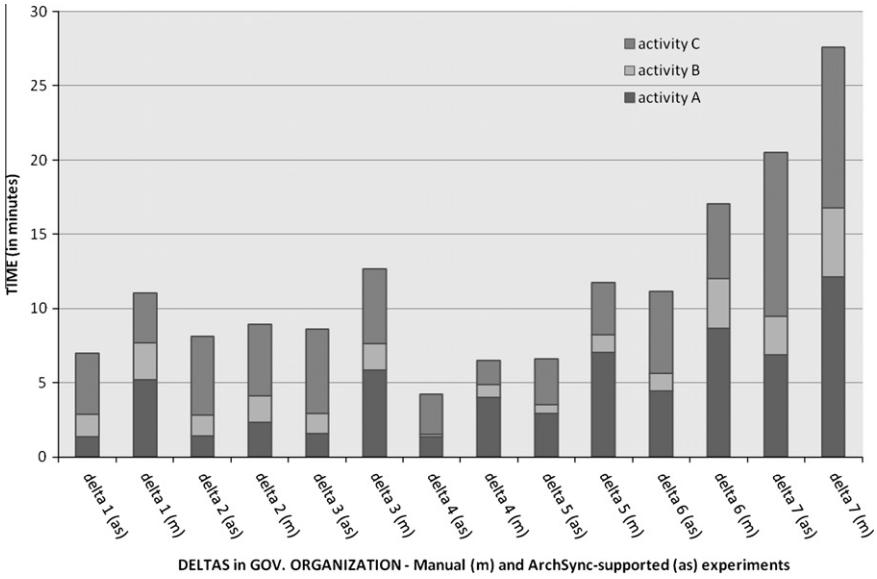

Fig. 12. Comparison of time efforts for the three conformance activities in case-study #3.

Along this line, we evaluated cases of inconsistent UCM mappings for the *Universidad3D* and *InQuality* case-studies. In particular, two types of inconsistencies were considered. We referred to them as *bounded* mapping and *erroneous* mapping, respectively. In the first experiment (bounded mapping), we selected a range of responsibilities and, for each responsibility, we modified its original mappings in such a way some Java methods were left unmapped. In the second experiment (erroneous mapping), we modified the mappings of selected responsibilities by replacing randomly certain methods by methods that were non-related to the responsibilities. Both experiments aimed at assessing the behavior of the tool when processing traces containing methods that are not necessarily "relevant" to the UCM responsibilities. Fig. 13 shows how the precision of the tool varies with the different mapping policies. The reference value is the precision (per delta) resulting from the original mapping set, as used in each case-study. Based on this configuration, we reduced the mapping set for the existing UCMs up to a specific percentage (bounded mapping). Afterward, and based on the initial configuration, we replaced a percentage of the methods from the original mapping set by erroneous methods (erroneous mapping).

In general, a considerable loss in tool precision was noticed in both case-studies. Analyzing the results, we found that as the percentage of reduction of methods increases, the tool precision decreases. As regards *Universidad3D*, excluding 66% (on average) of the methods from the mappings led to a drop of the precision from 0.81 to 0.62. On the other hand, replacing 8% (on average) of the methods by erroneous methods made the original precision decrease to 0.38. When it came to *InQuality*, taking off 30% (on average) of the methods decreased the precision from 0.80 to 0.72. On the other hand, considering 5% (on average) of the methods replaced by erroneous methods, the original precision went down to 0.46. In addition, we observed other responses of *ArchSync* when exposed to inconsistent mappings, namely: UCMs mistakenly tagged by the *DiffMapper* filter 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.

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.

4.5. Sensitivity analysis for synchronization time

Initially, the synchronization interval for *ArchSync* was just the delta between a version and the next one (remember that the reference interval is 10–12 days in the case-studies). As variations of this configuration, we delayed the time between UCM synchronizations, considering that a reviewer performs checks every two versions and every three 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.

From the experiments with incremental intervals between UCM synchronizations, we noticed a drop in tool precision for both case-studies.

Table 5 shows that precision in *Universidad3D* decreased to 0.75 when considering conformance checks every two versions. Moreover, precision decreased to 0.5 when considering conformance checks every three versions. Likewise, in *InQuality*, the tool precision decreased to 0.71 and 0.63 respectively. Although not shown in the table, we also observed lower number of (relevant) update scripts for each UCM. In *Universidad3D*, 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. Certainly, the threshold set for scripts in the correlation procedure (see Section 3.2) can partially contribute to a low precision, as "large" scripts to patch a mismatch might be pruned by the algorithm. In order to rule out this effect, we tried higher thresholds for the configurations above but the tool could not generate better scripts (either because of timeouts in the search or because the resulting script made no sense to the tool's user). To sum up, conformance checks must be conducted regularly for the tool to be effective.

4.6. Lessons learned and limitations

The purpose of the case-studies was to assess the viability of the *ArchSync* approach. In this context, the experiments showed that

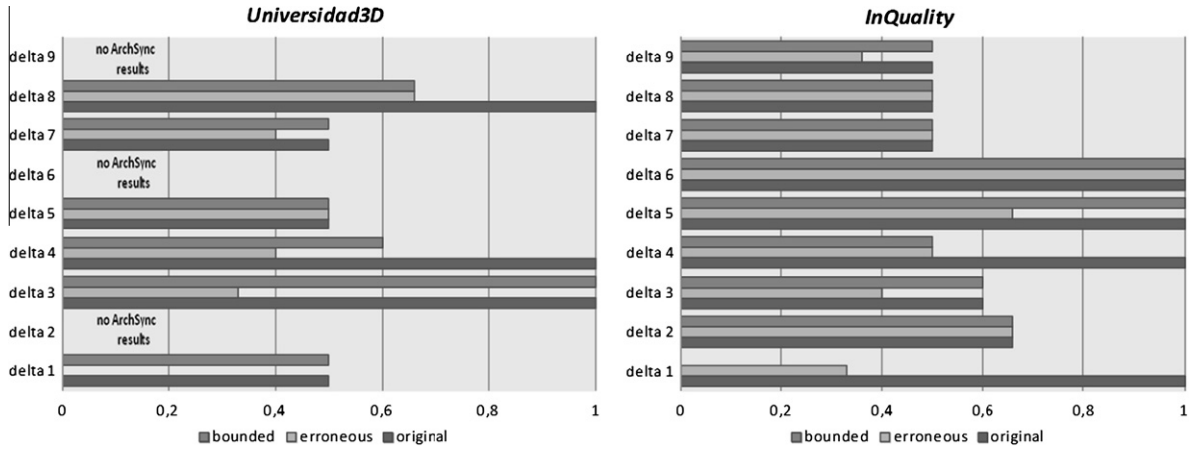

Fig. 13. Precision comparisons for different mapping policies in the *Universidad3D* and *InQuality* case-studies.

Table 5
Variations in tool precision for different time intervals (the italicized column shows reference values).

Delta	Interval 1	Interval 2	Interval 3
Universidad3D	*0.81*	0.75	0.50
InQuality	*0.80*	0.71	0.63

the tool does facilitate the synchronization of UCMs with code. Specifically, the tool helped the participants by reducing time and browsing efforts during conformance tasks, as well as by pointing UCM mismatches. The participants reported that visualizing mismatches in terms of UCMs helped them to narrow down the alternatives for resolving the problem. These observations should be interpreted in the context of participants having moderate experience about the system, which is the main user group *ArchSync* is designed for. In case of experienced developers, they should perform well without *ArchSync* assistance (either because they introduce few architectural violations, or because they are able to spot and fix violations quickly). This fact was actually hinted by the learning process that we observed in some participants, after they repeated the same conformance process on several deltas.

Regarding global time spent in a conformance exercise, the results of the *ArchSync*-supported experiments outperformed generally 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, since 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 revision 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 typically 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 the 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. We still found problems in the generation of scripts for certain versions. We had experiences of erroneous mappings or test cases that caused deflections in the responsibility activations, affecting 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 correlation 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 added directly to the code. The tool did not recognize these changes as relevant to the current UCMs. In the experiments, the participants coped with this problem by entering mappings for the new feature, in order to enable the heuristics to reason about the change.

The kind of mismatches detected by *ArchSync* takes a UCM-centered perspective of a conformance problem. In practice, this perspective does not consider the problem from a global architectural perspective (e.g., the combined effects of a set of UCMs on the architecture). Furthermore, update scripts can add links between responsibilities or add responsibilities to a UCM, but these scripts are not expressive enough to enforce architectural patterns. In the presence of several UCM mismatches, the architect must ultimately reason about their consequences for the whole architecture structure. For instance, it is possible that a reviewer using *ArchSync* does not solve a violation, but wait until she knows the compromises that might be introduced. Anyway, the tool can identify some violations that architects do not always realize themselves. For example, in case-study #1, there was a refactoring

of the *Game Logic* component. This refactoring propagated changes to other components, which were not (consciously) considered by the architect. When one of the participants ran *ArchSync* on a UCM related to *Game Logic*, the tool informed about an architectural violation that, at first sight, was judged as a false positive by the architect. A deeper analysis of the situation revealed otherwise. The participants reported similar situations in case-study #3, when dealing with the conversion of (existing) features into services.

Finally, another source of trouble for the heuristics is the execution traces. A dynamic analysis based on traces provides an approximate picture of the system's behavior, but there can be misinterpretations in the activation log and its further correlation with the UCMs. A positive aspect of execution traces though is that they expose component interactions not always detectable from static code dependencies.

4.7. Threats to validity

There are a number of threats to validity in our evaluation of *ArchSync*. A threat to construct validity is the retrospective nature of the experiments, in which the participants replayed changes from the issue tracker and code repository of the three projects. These changes were considered as architectural ones if they were visible in terms of UCMs. In these simulations, some relevant changes with no obvious UCM correspondences could have been overlooked.

The main threats to internal validity were the following. First, we defined a conformance process with three activities in order to facilitate effort comparisons during the experiments. A clear separation or sequencing of these activities is not always possible in practice (e.g., humans can perform the activities interleaved or rely on sub-activities). A different process choice or sequencing could have produced different measures. Second, there is a bias in the choice of versions for each case-study and in the input artifacts that we fed into *ArchSync* (e.g., UCMs, test cases). Third, we have to factor in the impact of the code instrumentation both on the application behavior and the size of the logs. Although the instrumentation showed no observable effects in the experiments, it might still bias the outcomes of the tool.

We believe that the *ArchSync* approach and our findings from the experiments are applicable to other case-studies, under the assumption that a consistent baseline of UCMs for the implementation is available. Along this line, the main threats to external validity were the following. First, the UCM notation is relatively easy to use for developers, although it is not a mainstream notation. In addition, the UCM notation admits various "modeling options" for representing the same architectural functionality. As demonstrated in Section 4.5, the architectural modeling can affect the results of the tool. Second, having a good traceability in terms of mappings between UCMs and code might not be realistic for some projects. For instance, some projects may only keep mappings from components to classes, or the mappings may not be maintained up-to-date over time. Third, the experiments (with and without *ArchSync*) were carried out by different developers. All these developers were familiar with their respective projects. Participants with other backgrounds, domain knowledge, or levels of expertise, might have spent different efforts. Related to this threat, we noted that the repetition of experiments through a sequence of deltas might affect the participants' performance, leading to gradual improvements toward the end of the sequence.

5. Related work

The correspondence between software architecture and implementation has been an active topic of research, with approaches that often employ visualization, reverse engineering and consistency checking techniques [3,6,7,9,16,22]. Some commercial tools (e.g., Lattix, SonarJ, or Structure101) provide basic support for monitoring structural compliance. However, the use of runtime system information at the architectural level has been little explored.

Conceptually, conformance approaches can be divided into three categories: (i) architecture-driven code generation, (ii) architecture reconstruction for comparison with intended architecture, and (iii) insertion of architecture checks in the code. Architecture-driven code generators are attractive, because the implementation will conform to the architecture by construction and the mapping function is usually predefined by the generation tool. On the downside, complete code generation is difficult, often limited to specific domains, and the architect must trust the code generator. The second category involves reverse engineering the code and then analyzing differences between the intended and extracted architectures. This approach is probably the most common for tools, as it provides flexibility regarding the mapping function and architectural views to be used. *ArchSync* falls in this category. Nonetheless, the reconstruction process can be labor-intensive, even with tool support, due to mismatches between the architectural views and the code. The third category consists of embedding architectural checks in the code. Along this line, techniques such as aspects are becoming popular for conformance [29]. Like in the second category, the architect is responsible for defining the mapping function; however, the mappings are likely to be distributed with checks through the code. Furthermore, the architect must define the conformance rules to be checked and the places in the code where those rules should be applied. In the following, we review a number of approaches in more detail.

DiscoTect [8] builds a C&C architectural view of a system using runtime information. This reconstruction approach maps low-level system events to specific architectural operations, which are interpreted by a Petri-Net engine to create instances of components and connectors. DiscoTect presumes an underlying architectural style (e.g., pipes and filters, client–server) and assumes specific coding conventions to implement that style. Other researchers have extended this approach [24] as part of an integration with the SAVE tool.[14] This tool can check conformance to sequence diagrams, comparing these diagrams against system runtime information. We apply similar techniques as DiscoTect for processing runtime events, but our mappings are not constrained by architectural styles. Instead, we base the reconstruction on the notion of responsibilities. This choice allows *ArchSync* to deal with different implementations more flexibly than with styles, at the cost of some precision loss when inferring architectural elements. Adding support for architectural styles (e.g., for an MVC style) to *ArchSync* would enhance the tool with style-driven scripts and responsibilities.

Egyed [25] describes a semi-automated approach to generate and validate trace dependencies between UML artifacts. This approach requires a set of test cases for the system, but unlike *ArchSync*, it also needs hypothesized traces linking some artifacts. The test scenarios are executed on the system to observe the lines of code they affect. A special algorithm processes this information to derive new traces and expose potential contradictions between traces. The main differences with *ArchSync* are that the UCMs are more abstract than conventional UML artifacts, and consequently, our correlation heuristic interprets execution traces as "refinements" of causal relationships between UCM responsibilities. More recently, Egyed [30] has proposed a technique for fixing inconsistencies in UML models by observing the evaluation of these models during consistency checking. In this approach, a tool assists the developer by presenting alternative repairs for problematic UML models. Since UML models are closer to the code than UCMs, the developer needs to look at detailed inconsistencies before deciding

[14] SAVE homepage: http://www.fc-md.umd.edu/save/.

on the right action. In terms of architecture conformance, Egyed's tool can detect a larger number of inconsistencies than *ArchSync*.

Rötschke and Krikhaar describe graph-rewriting toolset for monitoring and controlling the migration of legacy systems [31]. The toolset allows architects to specify the steps necessary for evolving the current architecture, as well as the consistency rules to be checked at each increment. This approach assumes a top-down development strategy in which changes happen first at the architectural level and then are propagated to the implementation. Nonetheless, it is unclear how this strategy would work when implementation changes leave the architectural specification out-of-date. Nentwich et al. [9] propose an approach for managing inconsistencies in XML documents using a language called xlinkit. Similarly to *ArchSync*, the user can choose from a set of possible repairs. In xlinkit, both the checks and repairs are treated in an abstract way, enabling flexible consistency policies. A perceived drawback is that the translation of checks and repairs to a particular specification must be done manually. Although the update scripts in *ArchSync* are less general than in xlinkit, they are easy to understand and apply by developers.

There is also conformance work based on configuration management techniques, such as ArchTrace [32]. ArchTrace provides tool support for versioning the relationships between a C&C architecture and its implementation, as the architecture or code evolve over time. The tool is equipped with user-defined policies implementing various types of synchronizations. These policies seem more flexible than those of *ArchSync*, as they can accommodate different development practices. Like in our assistance schema, ArchTrace policies require developer's intervention. However, the policies only consider structural aspects and not architectural behavior. Anyway, the policy-based infrastructure can be a good complement to *ArchSync*.

6. Conclusions

In this article, we have described a tool approach that helps architects to perform conformance checks between architectural scenarios and implementation. The goal of *ArchSync* is to prevent architectural drift of behavior, which is a novel aspect of our proposal. To this end, we have developed heuristic algorithms for the *ArchSync* tool that look at runtime system information and detect potential mismatches between intended and actual execution flows. Furthermore, the tool is able to suggest actions for fixing those mismatches. The approach relies on a UCM-based representation of the architecture, as well as on predefined mappings between architectural responsibilities and code. Another requirement of the approach is that the conformance checks on the UCMs must be performed on a regular basis.

Although the approach has room for improvements, the evaluation of the *ArchSynch* prototype in several case-studies has shown encouraging results. The *ArchSync* assistance 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., classes and UCM paths). Since the approach relies on heuristics, 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. Experiments to determine the scalability of the tool for large projects (and with disparate changes) are still pending.

There are several lines of work to improve the *ArchSync* assistance. On the modeling side, we are creating guidelines for developers to specify UCMs and systematically map them to code. Along this line, a candidate technique to explore is incremental mapping [24]. In order to facilitate the generation of application traces, we plan to link the UCMs with test-case suites for the architectural responsibilities. We will continue enhancing the reconstruction and correlation heuristics. For instance, complementing the behavioral checks with structural checks can make the tool more effective. Related to the differencing between intended and actual behaviors, it is important to extend the analysis to architectural properties associated to the UCM scenarios, such as execution time, security levels, or modification costs. For instance, it is possible to detect whether a UCM responsibility executed with errors [18], and then incorporate this information into the correlation algorithm. Another interesting topic is the applicability of *ArchSync* to check threads annotated with quality-attribute characteristics [34]. Furthermore, it is desirable to strike a balance between events logged by the tool and accuracy of its outputs. We will explore whether model-checking and configuration management techniques, such as [32,33,35], provide mechanisms to achieve this balance.

Overall, approaches like *ArchSync* are promising because they inform architects and developers about the traceability relationships between architecture and implementation, hence we foster an architectural thinking when making decisions about software changes.

Acknowledgements

The authors are grateful to M. Blech and J.P. Carlino, who greatly helped in the development of the *ArchSync* prototype; and to N. Vega, F. Vega and P. Aresqueta for their involvement in the case-studies. The authors also thank the anonymous reviewers that contributed to improve the quality of this manuscript.

References

[1] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, second ed., Addison-Wesley, 2003.
[2] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2002.
[3] N. Medvidovic, A. Egyed, P. Gruenbacher, Stemming architectural erosion by coupling architectural discovery and recovery, USA, in: STRAW'03, Software Requirements to Architectures Workshop, held at ICSE'03, 2003.
[4] D.L. Parnas, D.M. Weiss, Active design reviews: principles and practices, in: IEEE Computer Society, 8th International Conference on Software Engineering, ICSE'85, 1985, pp. 132–136.
[5] D. Kelly, T. Shepard, Task-directed software inspection, Journal of Systems and Software 73 (2004) 361–368.
[6] R. Kazman, S. Carriere, Playing detective: reconstructing software architecture from available evidence, Springer, Automated Software Engineering 6 (1999) 107–139.
[7] G. Murphy, D. Notkin, K. Sullivan, Software reflexion models: bridging the gap between design and implementation, IEEE Transactions on Software Engineering 27 (2001) 364–380.
[8] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, H. Yan, Discovering architectures from running systems, IEEE Transactions on Software Engineering 32 (2006) 454–466.
[9] C. Nentwich, W. Emmerich, A. Finkelstein, Consistency Management with Repair Actions, Portland, Oregon, USA, 2003, IEEE Computer Society, ICSE 2003, pp. 455–464.
[10] M. Abi-Antoun, J. Aldrich, D. Garlan, B. Schmerl, N. Nahas, T. Tseng, Improving system dependability by enforcing architectural intent, St. Louis, USA, Springer, 2005, Workshop on Architecting Dependable Systems.
[11] J.A. Diaz-Pace, J.P. Carlino, M. Blech, A. Soria, M.R. Campo, Assisting the synchronization of UCM-based architectural documentation with implementation, Cambridge, UK, in: IEEE Computer Society, WICSA/ECSA 2009, 2009, pp. 151–160.
[12] R.J.A. Buhr, Use case maps as architectural entities for complex systems, IEEE Computer Society, IEEE Transactions on Software Engineering 24 (1998) 1131–1155.
[13] R. Koschke, D. Simon, Hierarchical reflexion models, in: IEEE Computer Society, 10th Working Conference on Reverse Engineering, WCRE'03, 2003, p. 36.
[14] J. Buckley, A. LeGear, C. Exton, R. Cadogan, T. Johnston, B. Looby, R. Roschke, Encapsulating targeted component abstractions using software reflexion

models, Journal of Software Maintenance and Evolution: Research and Practice 20 (2008) 107–134.
[15] R. Koschke, P. Frenzel, A.P.J. Breu, K. Angstmann, Extending the reflexion model for consolidating software variants into product lines, Software Quality Journal 7 (2009) 331–366.
[16] M. Abi-Antoun, J. Aldrich, Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations, ACM, OOPSLA'09 44 (2009) 321–340.
[17] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, Wiley & Sons, 1996.
[18] A. Soria, J.A. Diaz-Pace, M. Campo, Tool support for fault localization using architectural models, in: IEEE Computer Society, European Conference on Software Maintenance and Reengineering, CSMR'09, 2009, pp. 59–68.
[19] R.J.A. Buhr, Making behaviour a concrete architectural concept, in: 32nd Annual Hawaii Conference on System Sciences, HICSS'99, Hawaii, USA, 1999.
[20] F. Bordeleau, D. Cameron, On the relationship between use case maps and message sequence charts, in: E. Sherratt (Ed.), IRISA, SDL Forum, 2nd Workshop of the SDL Forum Society on SDL and MSC, SAM, 2000, pp. 123–138.
[21] R.J.A. Buhr, R.S. Casselman, Use Case Maps for Object-Oriented Systems, Prentice-Hall, 1996.
[22] A. Postma, A method for module architecture verification and its application on a large component-based system Elsevier, Information and Software Technology 45 (2003) 171–194.
[23] D. Garlan, B. Schmerl, J. Chang, Using gauges for architecture-based monitoring and adaptation, Brisbane, Australia, in: Working Conference on Complex and Dynamic Systems, 2001.
[24] R. Koschke, Incremental reflexion analysis, Madrid, Spain, in: IEEE Computer Society, 14th European Conference on Software Maintenance and Reengineering, CSMR'10, 2010.
[25] A. Egyed, A scenario-driven approach to trace dependency analysis, IEEE Transactions on Software Engineering 29 (2003) 116–132.
[26] D. Ganesan, T. Keuler, Y. Nishimura, Architecture compliance checking at runtime, Information and Software Technology (2009) 1586–1600.
[27] A. Amandi, M. Campo, A. Zunino, JavaLog: a framework-based integration of Java and Prolog for agent-oriented programming, Computer Languages, Systems and Structures 31 (2005) 17–33.
[28] M.R. Campo, J.A. Díaz Pace, M. Zito, Developing object-oriented enterprise quality frameworks using proto-frameworks, Software Practice and Experience 32 (8) (2002) 837–843.
[29] P. Merson, Using Aspect-Oriented Programming to Enforce Architecture, Software Engineering Institute, CMU/SEI-2007-TN-019, 2007.
[30] A. Egyed, Fixing inconsistencies in UML design models, Minneapolis, USA, in: IEEE Computer Society, 29th Conference on Software Engineering, ICSE'07, 2007, pp. 292–301.
[31] T. Rotschke, R. Krikhaar, Architecture analysis tools to support evolution of large industrial systems, in: IEEE Computer Society, International Conference on Software Maintenance, ICSM'02, 2002.
[32] L.P.G. Murta, A. van der Hoek, C.M.L. Werner, ArchTrace: policy-based support for managing evolving architecture-to-implementation traceability links, in: IEEE Computer Society, 21st International Conference on Automated Software Engineering, ASE'06, 2006, pp. 135–144.
[33] F. Sousa, N. Mendoca, S. Uchitel, J. Kramer, Detecting implied scenarios from execution traces, in: IEEE Computer Society, 14th Working Conference on Reverse Engineering, WCRE'07, 2007, pp. 50–59.
[34] M. Gagliardi, W.G. Wood, J. Klein, J. Morley, A uniform approach for system of systems architecture evaluation, CrossTalk, The Journal of Defense Software Engineering, 2009.
[35] A. Rozinant, W.M.P. van der Aalst, Conformance checking of processes based on monitoring real behavior, Information Systems 33 (1) (2008) 64–95.