Characterizing Real-Time Reflexion-based Architecture Recovery: An In-vivo Multi-Case Study

Nour Ali
Lero-The Irish Software Engineering Research Centre, University of Limerick, Limerick Ireland
Nour.Ali@lero.ie

Jacek Rosik
Lero-The Irish Software Engineering Research Centre, University of Limerick, Limerick Ireland
Jacek.Rosik@lero.ie

Jim Buckley
Lero-The Irish Software Engineering Research Centre, University of Limerick, Limerick Ireland
Jim.Buckley@ul.ie

ABSTRACT

Architecting software systems is an integral part of the software development lifecycle. However, often the implementation of the resultant software ends up diverging from the designed architecture due to factors such as time pressures on the development team during implementation/evolution, or the lack of architectural awareness on the part of (possibly new) programmers. In such circumstances, the quality requirements addressed by the as-designed architecture are likely to be unaddressed by the as-implemented system.

This paper reports on in-vivo case studies of the ACTool, a tool which supports real-time Reflexion Modeling for architecture recovery and on-going consistency. It describes our experience conducting architectural recovery sessions on three deployed, commercial software systems in two companies with the tool, as a first step towards ongoing architecture consistency in these systems. Our findings provide the first in-depth characterization of real-time Reflexion-based architectural recovery in practice, highlighting the architectural recovery agendas at play, the modeling approaches employed, the mapping approaches employed and characterizing the inconsistencies encountered. Our findings also discuss the usefulness of the ACTool for these companies.

Categories and Subject Descriptors

D.2 **[Software Engineering]**:

D.2.2 Design Tools and Techniques and D.2.11 Software Architectures

General Terms

Design

Keywords

Architecture Recovery, Architectural Conformance, Architecture Consistency, Reflexion Modeling, Architecture Evolution, ACTool, In-Vivo Multi Case Study

.

1. INTRODUCTION

Architecting software systems is an integral part of the software development lifecycle providing a basis for achieving non-functional requirements [1]. However, often the implementation of systems ends up deviating from the designed architecture. If consistency between the as-designed and the as-implemented architecture is lacking, the quality requirements addressed by the as-designed architecture are likely to remain unaddressed by the as-implemented system. Additionally, if the as-designed architecture is inconsistent with the implemented system, it may prove confusing for developers charged with evolving the system when they need to refer to both the code-base and to the architectural documentation. This inconsistency can hinder evolution of systems and, even if recognized, can prove costly to resolve [2]. Yet architectural inconsistency seems prevalent in the software industry worldwide [6] [16] [17].

An effective approach for identifying architecture inconsistencies in already implemented systems is Reflexion Modeling [8] [18]. In this approach, the system's designed architecture is defined in terms of logical components and their connections. Later, the elements of the source code are mapped onto this defined architecture and the relationships between the mapped source code elements are assessed for consistency with the connections between the components of the defined architecture. Thus, Reflexion Modeling shows where the source code connections are inconsistent with the designed architecture.

The ACTool [11] is a tool which embodies real-time Reflexion Modeling, in line with the suggestions of Rosik [10] and Knodel [2]. Specifically it not only allows the architect to define the as-designed architecture and probe which source code elements seem to deviate from that as-designed architecture in batch mode: It also alerts developers to the architectural inconsistencies they introduce*, as they introduce them* when coding, or when mapping the architecture to the source code. Thus, it raises awareness of inconsistencies in a timely fashion and, because it allows users navigate to the inconsistencies in the source code directly, it aims to lessen their persistence.

This paper describes our experiences when using the ACTool in-vivo. It reports on the first stage of our longitudinal studies of architectural consistency: where users first check (recover) their system against the designed architecture using the tool. When satisfied, the intention is to deploy the ACTool team-wide, in the 2nd stage of our studies: to assess its effectiveness against the continued introduction of architectural inconsistencies as time goes on.

The findings reported in this paper (stage 1) provide a characterization of architectural recovery in practice, highlighting the differing architectural recovery agendas at play, the different modeling approaches, the mapping approaches and the different types of inconsistencies that participants encounter. Such a characterization of Reflexion-based architectural recovery is lacking in the literature, leaving open the possibility that the current approaches are sub-optimal for users of this and similar architectural recovery techniques [2][14] [12]. Our findings also discuss the usefulness of the ACTool to these companies.

The paper is structured as follows: Section 2 presents an overview of the techniques that have been used in architecture recovery and consistency and our previous work related to Reflexion Modeling. Section 3 describes the empirical study we conducted on three deployed commercial systems. Section 4 presents and discusses our results. Section 5 presents related work. Finally, section 6 summarizes our conclusions and highlights our further work.

2. Background

2.1 Architecture Recovery and Consistency

Architecture Recovery techniques have been proposed to address the situation where architectural inconsistencies have arisen over time [3], [4], [5]. Several of these techniques rely on cohesion and coupling measures of the existing software to propose an appropriate architecture. However, these approaches have several limitations. For example, they are driven by the code-base and not by the architect's agenda. Additionally, if the architecture of the code-base is flawed, the cohesion and coupling measures reflect this flawed, as-implemented architecture.

In Koschke's review of the area, he suggests a more proactive, immersive role for the architect in any such approaches [6]. One approach that had already achieved this was proposed by Murphy et al, called Reflexion Modeling [7]. In Reflexion Modeling, architects graphically define their envisaged architecture of their system in terms of its components and its inter-component connections. This is represented in a diagram. The architect then maps elements of the code-base to each component node. A subsequent parse of the system identifies the actual connections between the architecture's components (as defined by the actual dependencies between the mapped code-base elements). This parsed data can be superimposed on the original diagram to identify:

- Connections which the architect envisaged (between architectural components) that do actually exist in the implementation,
- Connections that they envisaged that do not exist in the implementation and,
- Connections that they did not envisage that do actually exist in the implementation.

The literature suggests that Reflexion Modeling is effective in identifying inconsistencies between the as-implemented and as-designed architecture of existing systems [8] in hindsight. That is, it has been successfully trialled in situations where architectural inconsistencies have become entrenched in a deployed, evolving system over a long period of time and the organization has retrospectively decided to identify these inconsistencies.

2.2 Architecture Consistency Tool (ACTool)

In an attempt to avoid the costs associated with retrospectively addressing these inconsistences, our group trialed the periodic application of Reflexion Modeling during the implementation of a commercial system in IBM. We found that when Reflexion Modeling sessions were held every four months, during the re-development of a software system from scratch, the architects and developers liked the approach and were able to identify architectural inconsistencies in every session [9]. However, the developers did not remove any of the architectural inconsistencies identified, citing time-to-market pressures coupled with fear of possible ripple effects from refactoring changes [10].

These comments implied a high perceived workload for developers in revisiting and amending inconsistent code in hindsight. This, in turn, implied that a real-time, forward engineering approach would be more successful [11]. In this approach, developers would be informed of architectural inconsistencies as they made them [11]. This conclusion was also reached by Passos et al. [8], in their review of architecture consistency techniques: They stated that Reflexion Modeling was the most appropriate architecture consistency technique but that it should warn developers about the architectural inconsistencies they introduce, as they introduce them.

The ACTool, as illustrated in Figure 1, embodies this recommendation. It permits the definition and consistency checking of architectural models. To define a model (1), components and their connections, represented as solid edges, can be dragged and dropped from a palette (2). Later, mappings between the source code and the components in the architecture can be defined. This is done, by dragging elements of the source code (for example, packages, classes and interfaces) from the package or navigator explorer (4), and dropping them into the components of the model. It is worth noting that many source code elements can be mapped into one component. A summary of these mappings can then be viewed in an outline view (5). Architectural models and mapping can be defined incrementally and iteratively.

As mappings are defined, the results of the Reflexion Modeling analysis are presented, in terms of the edges between the defined architectural elements:

- Convergence: A relationship present in both the defined architecture and the implementation. It is represented as a solid edge.
- Divergence: A relationship present in the implementation but not present in the architecture. It is represented as a dashed edge.
- Absence: A relationship present in the architecture, but not present in the implementation. It is represented as a dotted edge.

In the case of convergent and divergent edges, the tool shows the number of source code relationships underpinning each architectural edge: for example 4 between Command and Common in Figure 1. When either of these edge-types is clicked upon, the tool lists the source code relationships underpinning the edge (see the Architectural Relations view (3) for the code relationships underpinning the edge between Command and Common). In an extension to the original JRMTool, the ACTool allows the user to click on the source in the architectural relations view to navigate to the associated source code.

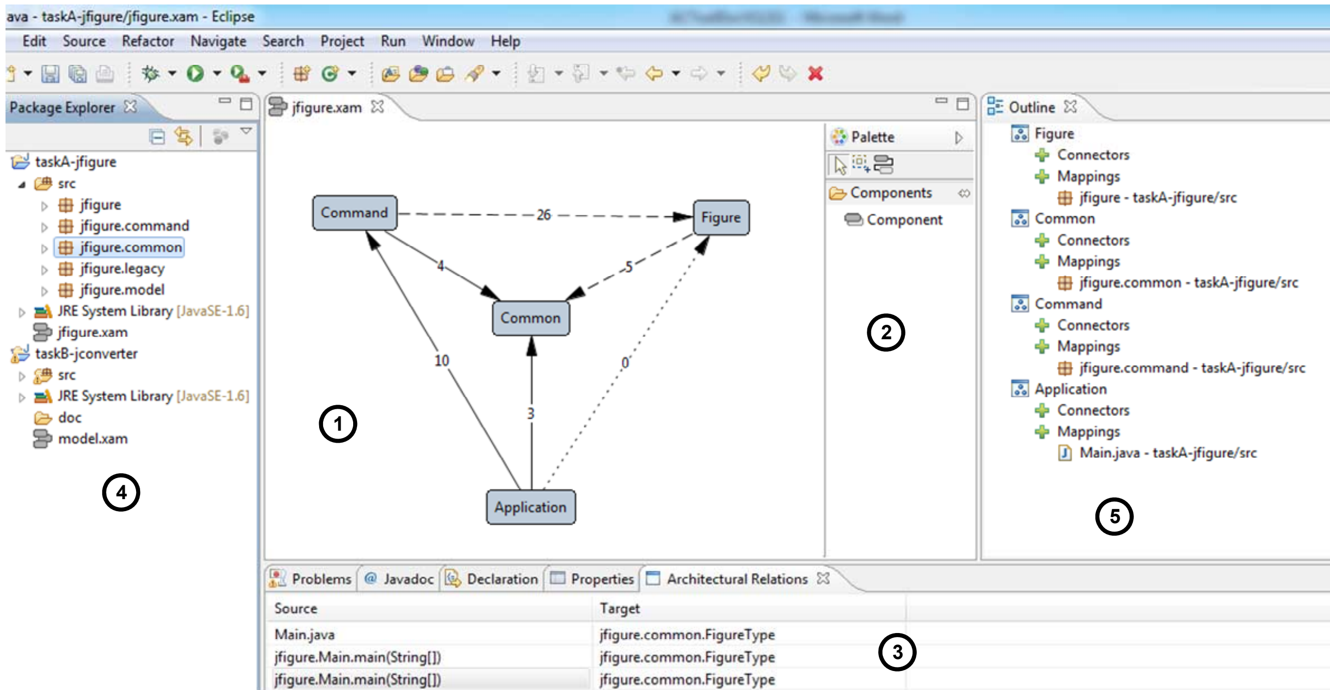

Figure 1. A view of the ACTool

It is important to point out that a user can also drag and drop an element from the package explorer view (i.e. the source code) into the architectural model without having predefined a component for that element. This action will automatically create a component in the architectural model and map it to the dropped element. Also, as you drop that source code element into the model, the relationships between that element and the rest of the architecture are shown instantaneously as divergences, because no relationships were defined in the architectural model in advance. If a user agrees that the divergent relationships should be in the architecture, then he/she only has to overwrite these relationships with a solid edge, using the palette.

Additionally, the ACTool also works in the coding view. Specifically, as the developer writes code and saves it, they will be notified of any architectural inconsistencies they have introduced through associated warnings in the margins. This facility aims to educate developers/architects about the architecture in a timely fashion, giving them the opportunity to remove any inconsistencies before these inconsistencies become entrenched and they move on to different agendas/locations in the code base.

3. EMPIRICAL STUDY

3.1 Motivation for this study

There is limited guidance, in the literature, as to the modeling and mapping facilities desired by practitioners when using Reflexion-based tools for architecture recovery. This is surprising given the success of such tools. Without this guidance, it is hard to assess for example, the relative merits of the lexicon-based mapping approach of the original Reflexion Modeling tool [12], and the drag-and-drop mapping approach of the ACTool. Indeed additional mapping approaches that may be required are obfuscated without such a characterization.

There is also only limited characterization of the architectural inconsistencies found in commercial systems. The one characterization we are aware of is based on the study of one commercial system only [9], and needs to be buttressed by additional in-vivo findings if it is to give any insights into the inconsistencies that arise in vivo. These insights are also important in guiding this tool's development.

Finally, the underlying Reflexion Modeling technique [7] is based on an approximate, abstraction-based mapping from the source code to the architecture and, a previous case study in this area [10] has suggested that this may in fact serve to obscure some source code inconsistencies from the user, significantly affecting the usefulness of the approach. Specifically, a convergent edge may represent a number of expected source code relationships *and* a number of unexpected source code relationships. The apparent convergence may direct the programmer's attention away from these inconsistencies. This study aims to probe these issues to improve our understanding of architecture recovery and consistency approaches.

3.2 Research Questions

The objective of this study is to characterize architecture recovery as achieved through Reflexion-based approaches and to hone their usefulness accordingly. Thus, the following are the research questions we attempt to address:

- What are the desirable architectural modeling characteristics during architecture recovery?

Table 1. Characteristics of systems under study

	Java Files	Packages	Project	LOC	Age	Deployment	No of Developers
SA	173	25	1	38582	1.25 years	5 months	6
SB	239	124	2	75,556	2 years	18 months	11
SC	7300	1105	1	2,223,872	>10 years	>120 months	>30

- What are the desirable source-code mapping characteristics?
- What kinds of architectural inconsistencies are identified using the approach?
- Are there cases when convergent edges obfuscate source code inconsistencies?
- Do companies find the ACTool useful in terms of identifying inconsistencies and solving them?

3.3 Case-study Context

An in-vivo, multi-case-study protocol [20] [21] was adopted across two financial organizations. These organizations both had an Irish-based presence, with large software portfolios and both were interested in heightening their systems' quality through architecture consistency checking. They were responsible for selecting the target systems for these case-studies.

3.4 Subject Systems Description

The first organization selected 2 separate systems (SA and SB) for study and the 2nd organization selected one (SC), much larger, system. Table 1 shows various size attributes of these systems. The three systems were developed in Java using Eclipse.

SA is a system that allows people in the organization to access customers' data, and provisionally update that data. People with specific administration roles in the organization are responsible then for approving or rejecting these update requests.

SB is a tool for external clients to access the company's information. There are two clients: institutional users and individual users gathering marketing material.

SC is a system that has evolved over many years to become a complex claim-management and payment software system.

3.5 Participants

For each session, a different participant used the tool. For SA, the participant (henceforth PA) was the lead developer for the system in question. He has 16 years of experience as a software engineer and 2 years as an architect, in a larger corporate-wide role.

PB in session SB had 5 years of experience in software development. He has participated in the development of the system but was not involved in explicitly defining the architecture of the system. He did, however, state that he understood the architecture and this was re-enforced by the team lead who nominated him for the role based on his understanding of the system.

PC in session SC had 22 years as software engineer and 12 years as a software architect. He was chief architect for the system in question and also participated in developing parts of the system.

3.6 Study Protocol

A 20 minute demo of the ACTool was given to each of the participants. The objective of the demo was to show the architecture recovery and consistency capabilities of the tool and to demonstrate how to use it. Interested readers can view a similar, but briefer demonstration of the tool at http://www.lero.ie/project/rca/arc.

After this, the participants installed the ACTool as a plugin to their Eclipse IDE, and chose the system that they wished to study. The participants stated their original architectural model of the system, or part of the system. 2 (PB, PC) used the ACTool to do this and PA using a paper based model he had created in advance of the session. He then replicated his model in the ACTool. The authors acted as observers of the sessions and aided in any technical support that was required. Each architecture recovery session lasted a bit more than an hour and included a number of iterations, where the participants focused on increasing the depth or scope of their architectural models. Therefore, in each iteration new components and/or mappings were added.

3.7 Data Collection

The Participants first answered a brief questionnaire, characterizing the system. Then they started to use the ACTool for architecture recovery and their session was videotaped. The participants' screens and remarks were captured, providing valuable data on the process of creating their architectural models, and mappings, the inconsistencies identified, the participants' observations on these inconsistencies and their tool usage. At the end of the session, participants were asked to probe some convergent edges for underlying inconsistencies and were asked open questions about the value of the results obtained, and the tool's usability. After each session was complete, all the assets created during the session were collected for storage and analysis. These included:

- Video recordings of participants' screens, their tool interactions, and their think-aloud.
- Screenshots of their architectural models, and the inconsistencies identified in each iteration.
- An ACTool-produced file containing the mappings, connections and components defined by the participant
- Notes were also taken during the sessions, highlighting any important, observed events.

The gathered material of the three sessions amounted to over 6 h of video recordings which was transcribed and, together with other material, thoroughly analysed and discussed to record important findings. This analysis initially consisted of an open-coding-like phase, in the spirit of Corbin and Strauss [22], where the analysis was informed by knowledge derived from the literature. This was done independently by 2 of the 3 authors and subsequent discussions between the authors focused on the

reliability, veracity and relevance of their independent findings for the research questions identified in Section 3.2. The results described here are the outcome of these discussions

4. Results and Discussion

PA and PC iterated through the Reflexion Modeling process 4 times, and PB did so 3 times. At the end of PA's session, 8 components and 8 mappings were defined, in PB's session, 10 components, and 56 mappings were defined, and in PC's session, 14 components and 14 mappings were defined. All 3 users found a number of inconsistencies in their systems, even though 2 of the systems were less than 18 months old.

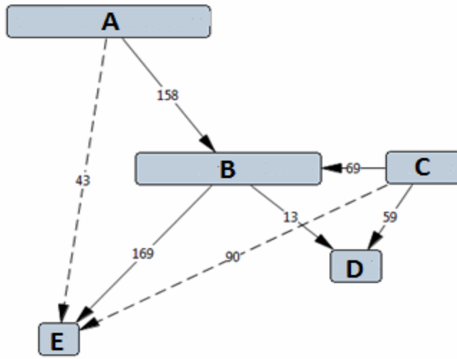

Figure 2. SA reflexion model in the second iteration

4.1 Desirable Architectural Modeling

In this section, we describe an overview of how the participants used the tool to model their architectures and discuss our conclusions.

The architectural model defined for SA was based on an N-tiered layered architectural style, as per the company standard. In the first iteration, 3 layers were defined and in the 2nd iteration the middle tier was expanded into 3 partitions (see nodes B, C and D in Figure 2). Then, in the third iteration, PA expanded the top layer into two partitions. This is shown in Figure 3 where the *A* layer in Figure 2 was decomposed into *A1* and *A2*. This was done to get a more in depth view of the top tier, detailing the inter-connections between its subcomponents (the edge between A1 and A2 in Figure 3). Unsurprisingly, giving the standard nature of the N-tiered architecture in this organization, the architectural model defined for SB, was also based on this N-tiered style.

The architectural model definition for SC was based on a more functional decomposition of the system. That is, he was interested in looking at functional "partitions" of the system and checking if communication between these partitions was exclusively through their defined interfaces.

These strategies suggest that:

Architectural templates should be provided in architectural recovery, supporting different decompositions.

Architectural templates create and enforce specific software architecture configurations. These findings suggest that our ACTool should be extended to support different templates that can be chosen by architects when defining new architectural models.

Here, the participants strategies suggested that an N-tiered style and an interface template should be made available, although others should also be considered. The tool could model different nodes representing different tiers, and default edges that allow relationships between adjacent tiers, i.e., messages can only be sent to immediately lower or upper layers. In the interface template, each node (component) would access other components through interfaces, The tool would automatically generate these parts of the architectural model. Additionally, this latter template suggests a specialist interface-type component node and interface-type edge. The component would be responsible for checking that any element calling a method of the interface is doing so through the interface and this could also be useful in COTs situations where you do not have the source code of the underlying implementation to interrogate. Both the interface component and the interface-edge could be distinguished in the model, to illustrate their interface-nature, and thus facilitate communication of the architecture to the wider team.

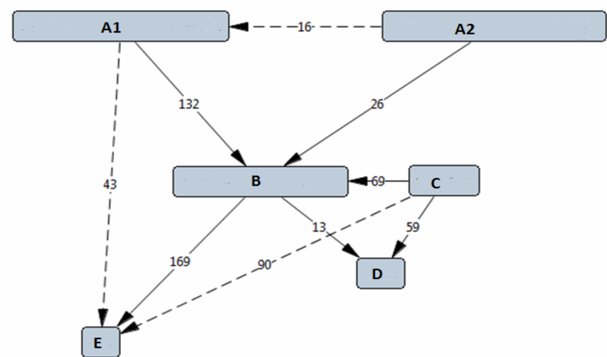

Figure 3. SA reflexion model in the third iteration

Another related issue that was noted in PA's session was the tool's inability to model anything outside of the static code-base. Its inability to determine the code-base's relationship to dynamically configured elements (Javascript for example) and its inability to include 3rd party components, even at the externally-visibility level frustrated PA slightly. He was particularly interested in identifying calls to a 3rd-party Business Process Manager and capturing events that the system listened to from that manager. While it may be a simple task to capture the calls to this Business Process Manager via the interface-type node suggested above and the lexical conventions of the calls on that interface, identifying the events listened to through, for example, a Spring framework [23] is a much more difficult proposition, as is capturing any dynamically configured relationships through a layer of indirection. However, the expressed desire of all the participants was that:

Architectural recovery should be scoped beyond the source-code base.

Over the architecture-recovery iterations, the participants generally moved to more encompassing architectural models. That is, they started with a small proportion of their system and added more component nodes, reflecting more of the underlying system, as they iterated through the process. An exception was noted in PC's session where, when inconsistencies were discovered, he chose to focus on the component in question and decompose it into finer grained elements. His aim was to identify more specifically the causative elements of the inconsistencies, which he preferred to do based on hypothesis and model

decomposition, rather than scanning through the architectural relations listing. This approach was also implied by PA, when he stated that he would break down one of the components (*C* in Figure 3) in his model to characterize the inconsistent edge. This suggests that:

A hierarchical modeling capability would be useful.

This has already been provided by several tools [19] and involves defining nodes that can themselves contain component architectures.

4.2 Desirable Mapping Characteristics

PB originally built his model and subsequently defined the code mappings to the model, as per the original Reflexion Modeling approach. He continued with this strategy during his session and only on one occasion did he drag and drop a source code element directly into the architecture as a new component. He said: "I am doing this because I do not know how this element works". PA used a mixed approach, where he not only defined components and mapped source code elements to them afterwards, but also dragged and dropped source elements into the model as new components. PC preferred to drag and drop elements of the source code into the model as components directly. Thus, he viewed the connections which appeared automatically and then confirmed whether a connection should exist or not by changing a divergent relationship (a dashed line) into a convergent one (a solid line). Relating to this PC said, "This works reasonably well... it just makes it convenient and it makes things quicker".

This is interesting because it suggests that 2 of the 3 participants, admittedly, small data-set participants, were more than happy to evolve the architectural model directly from the source code in a real-time manner, rather than to define it in advance. It suggests that:

An advantage of real-time Reflexion Modeling is that it gives programmers the ability to build models directly from the source code structures, in an incremental manner, with real-time feedback, rather than defining the model in advance of the mapping.

In effect, this change in protocol subsumes 3 steps of the original Reflexion Modeling process (create model, map model and compare model), and can be seen as an efficiency of this instantiation. This efficiency was desired across these 3 case studies.

PA and PB's systems adhered to a company-wide standard architecture, yet the mappings facilities required by both participants differed hugely. This can be seen by the fact that PA used 8 mappings and PB used 56.

In SA, the packages in the package explorer were directly related to each tier (some utility nodes were the exception). In SB, each package contained elements from each tier, leading to a much greater effort requirement for PB, when using the ACTool's drag-and-drop mapping facility. This is why PA typically had 1:1 mappings between source code elements (packages) and components, while PB had, on average 1:5 mappings between them. In fact, in the interview after his session PB expressed the desire for a lexical mapping facility in the tool, as per the original JRMTool. This was because his team's conventions demanded that the individual elements in each package had lexical signals as to their tier placement. Interestingly, this issue also arose for PC. In SC, many of the partition interfaces were placed within one 'interface-utility' package. While the participant was happy to drag these interfaces out individually, it would have been helpful if he could have dragged the interfaces associated with a package out, based on the lexical matching evident between the partition (package) and its associated interface. This suggests that:

Users should be offered a drag-and-drop mapping facility ***and*** *a lexical mapping facility.*

However, it also raises interesting questions about the potential for confusion when the 2 facilities are offered in conjunction with each other. For example: when the drag-and-drop mapping facility conflicts with the lexical mapping facility, which one should take precedence? Additionally, will users forget which of the 2 mapping facilities they used for certain parts of the source, or which one has precedence in the case of overlaps? Such confusion could lead to errors in the architectural recovery process.

PB mentioned that while "Lexical analysis would come in handy based on naming convention... many files would not fall under a lexical definition". This is entirely plausible: situations where conventions demand more than 1 naming approach, or simple programmer inexperience may lead to lower naming consistency in a software system that remains unnoticed for long periods of time. Thus, it suggests that:

Where both modes of mapping are made available, drag-and-drop should have primacy.

Even with the one mapping facility that was made available in these case studies (drag-and-drop), PB and PC mentioned difficulties in keeping track of the unmapped parts of the code-base. This difficulty would likely be exacerbated by the availability of more than one mapping-based strategy. While mapping information was made available in the outline view by the ACTool, no perspective illustrating the unmapped source code is available and this should be addressed:

Participants should be aware of the source code that remains unmapped for each architectural model they create.

4.3 Architectural Inconsistencies

Table 2 summarizes the number of architectural inconsistencies that appeared for each system, in each iteration (I). While these figures may look alarming at first, it should be noted that they were frequently indicative of a lesser number of (repeating) underlying issues. In addition, these inconsistencies were based on the individual participants' understanding and in several cases they said that they would like to go back to the team for absolute clarification.

Table 2. Inconsistencies appearing in the Systems

	I1	I2	I3	I4
SA	43	90	16	5
SB	0	2	20	NA
SC	0	26	86	3

Notwithstanding, a large number of source code inconsistencies were identified, many of which the participants did not anticipate in advance. These were not seen as system-critical issues but issues that did require attention for the maintainability of the system going forward.

In the following, we characterize the kinds of architectural inconsistencies that were detected during the sessions. It builds on a previous classification of Rosik et al, [10] as illustrated in our discussion:

A. Misplaced Constants (Constant Access)

These kinds of inconsistencies refer to when an element of a component is accessing misplaced constants in another component.

This kind of inconsistency was present in SB and SC. The total number of inconsistencies of this type was 39. In SB, it appeared in iterations 2 (2 inconsistencies) where a constant was misplaced in a component, and in iteration 3 where constants where misplaced in another (11 inconsistencies). To illustrate, PB stated "I think, we are using a defined constant in X and we should define this constant somewhere else".

This kind of inconsistency also appeared prevalently in Rosek et al's case study [10] and illustrates well the difficulties caused by the granularity difference between micro-implementation and macro-design, with respect to architectural consistency.

B. Accessing Forbidden Elements

This kind of inconsistency reflects when components are invoking other components they should not be.

In SA, this kind of inconsistency appeared in iteration 1 and 3, with a total number of 45 associated inconsistencies. Three different classes were calling methods of classes they were not supposed to. This kind of inconsistency was also present in SB. It appeared in iteration 3 with a total number of 5 inconsistencies. PB said, "I would have thought that X classes would only process data, and not access it (through accessor methods)". To solve this issue, PB mentioned that a new component should be added that would be in charge of caching data or that they should change the location of the functionality to another existing component.

C. Unorganized code

This inconsistency is related to having placed components in the wrong package hierarchy. This kind of inconsistency was present in SA, and appeared 16 times in the third iteration. PA said, "This guy (source code element) needs to move up to the X package". He mentioned that it was not urgent but that it should be changed for neatness and maintainability purposes. This kind of inconsistency was also present in SB. It appeared in iteration 3, PB saying, "X is in the Y project, and I do not know if it fits in Y". He later mentioned that X needed to be moved to another project. In total 4 inconsistencies of this type were identified in his session.

D. Ignoring Interfaces

This kind of inconsistency indicated that a component was making method calls to another component without using its corresponding interface. This kind of inconsistency predominantly appeared in SC, as was to be expected based on PC's initial agenda. When PC was asked about the solution to this kind of inconsistency, he said "some of them are trivial and some may not be". He mentioned that a possible solution is to create a wrapper around the components.

E. Genuine Omissions from the Designed Architecture

Several inconsistencies (not presented in Table 2) arose because the participants forgot to connect components in their architecture or because they were not aware of them initially. However, after navigating through the inconsistencies and the source code, participants would recognize that these connections would be needed. This kind of inconsistency also appeared in the previous case study [10].

To locate the specific inconsistencies, the participants typically focused on the architectural edges and went to the associated source code by double clicking on the architectural relations view, as anticipated.

However, several times, as mentioned in section 4.1, the participants PA and PC, expressed the desire to add subcomponents to the architectural model to localize the origins of inconsistencies. This worked particularly well for PC who had strong hypotheses about possible causes. In one instance, he was able to identify a specific class as a frequent source of inconsistencies quite quickly. It called methods in another partition 60 times without using the appropriate interface. PC said, "Somebody didn't realize that the interface is there and that they should not do it (bypass the interface)…Our build tools are not helping them here".

This desire to localize and prioritize the source code dependencies underpinning inconsistencies suggests:

A facility where the tool informs on the prevalent underpinning source code relationships causing a divergent edge should be available.

This facility can work by identifying the underpinning source code relationships, identifying any commonalities (for example data-type accessed, specific classes making invocations) and then ranking the most prevalent causes of architectural inconsistencies for that edge and presenting them to the user. This could be done by ranking and grouping the architectural relations view or by having an information widget appear when the user hovers over a divergent edge.

PC's session also raised the requirement for a data perspective. Specifically, recurrent accessing of a data-type was one of the prevalent causes of an inconsistency in SC. This suggests that there should be:

An architectural entity that can capture a data-type and-or access to this data type is required.

Such a facility was proactively suggested by another of our commercial partners in initial discussions. Our literature review suggests that this has not yet been undertaken in Reflexion-based architectural recovery approaches.

Several times PB was surprised by the inconsistencies that appeared, but later recognized they were right, and changed them to convergent edges. In other cases, PB would remove inconsistencies but mentioned that maybe the team architect should have the ultimate decision. PA also said that he would like to refer back to the entire development team for a more considered opinion on some of the inconsistencies he identified. *The participants gained substantial explicit knowledge, at an individual level, of the architecture through the architecture recovery sessions.*

The team will benefit from the sessions through participant queries to that team, and that a more consistent team-based understanding will be achieved.

4.4 Convergent Edges

In a previous finding [10], it was suggested that convergent edges may serve to obfuscate divergent source code relationships, a limitation alluded to as approximation by Murphy et al [14]. In this study, we asked the participants to randomly select at least 2 convergent edges, and to state if their underpinning source code relationships actually did converge, when studied in depth. Each participant did this, providing a pool of 7 convergent edges in total and 205 underpinning source code relationships. Interestingly there were no divergent source code relationships under these 7 convergent edges, suggesting that this issue did not arise in these 3 case studies. This implies that:

Convergent edges did not hide divergent source code relationships in these case studies.

4.5 Usefulness of the Tool

Although already well-established, in the literature, this case study provides further evidence on the usefulness of traditional Reflexion-based Architecture Recovery:

"It gives us a very sense of what has actually happened. It has always been very difficult to visualize or internalize this. This has been enormously interesting, it is always impossible to get this kind of metrics which we are getting here." (PC)

"To be fair, that ability is very useful, (referring to showing the number of the inconsistencies in components) …., it allows the people to walk on the dimensions of the problem" (PC).

"The tool is useful in terms that an arrow and screen is easy to visualize and understand the code"- (PA)

Using the ACTool to enhance this process, real-time architectural feedback was provided. Additionally, a drag-and-drop facility, for creating the architecture, augmenting the architecture or mapping to the existing architecture directly from the source code was available. These facilities were well received by the participants, as discussed in Section 4.1, where they often chose to incrementally build or augment their model directly from the code base and get consistency feedback instantaneously.

"I think that the results were instant, that when you dragged your package or class it showed violations (referring to inconsistencies) straight away" (PB)

The participants also liked the link from the architectural model, through the architectural relations listing to the source code itself:

"I think the list is very useful, it is a quick way to jump through them" (the number of inconsistencies – PC)"

"... Nice, absolutely it (navigation to source code) is useful" (PA)

"I like that it actually highlighted in the code where exactly those violations (referring to inconsistencies) were" (PB)

Indeed, all of the participants foresaw themselves removing some of the inconsistencies using this functionality of the tool

"Certainly I will solve some of them. Maybe some of them, the case is to change dashed lines into solid lines" (PA)

Participants still had some issues regarding the scalability of the approach using the ACTool. Specifically they felt that the architectural diagrams became unwieldy as they grew, resulting in PA asking for way-points in the edges and for PC to suggest automated layout algorithms that would limit edge-crossings. While automated approaches would alleviate many of these problems, it could be at the expense of implicit layout information that the architect might wish to convey (note that in Figures 2 and 3, the tier information is implicitly represented by the y-axis layout). Hence automated layout should probably only be introduced as an additional option when the need for layout-conveyed information is not as important. Another possible solution is the hierarchical partitioning suggested in Section 4.1.

One scalability facility that the participants liked in the ACTool was the ability to grey out specific nodes and their incident edges. This facility was particularly appreciated when modeling utility components that had a lot of incident edges.

"Ah, that's nice" (referring to the hiding facility –PA).

"I like that you can grey out the model and this cleans up the number of lines and makes it easier to understand and read" (PB)

In terms of overall quality, all 3 participants thought that the exercise, would serve to substantially "tidy-up" the code-base and thus improve the "maintainability" of the systems they were working on. PC noted 2 additional benefits:

- He thought the approach would be useful in partitioning the system into work units for development team management and for product release.
- He also noted that the analysis team in his organization, who are responsible for planning and estimating the impact of new evolutions of the system, would find the approach very valuable, allowing them to identify ripple effects of the changes they envisaged and allowing traceability from their planning of change, to its actual implementation.

4.6 Threats to Validity

Important for any empirical study is an assessment of its validity. Validity refers to the degree that empirical results are meaningful [13]. In the following, we discuss the validity and reliability threats that can be found in these case studies [15]:

Construct Validity deals with the extent to which the constructs as measured relate to the research phenomenon studied. In this instance, the phenomenon was an architectural recovery process and the measures were participants' protocol and think-aloud during that process. However this material is open to individual interpretation and subjectivity during data analysis. The researchers counteracted this possibility through discussion meetings. Hence, to increase construct validity we have also employed a participatory research approach where the participants (PA, PB, and PC) reviewed our findings for misinterpretations and inconsistencies.

Internal validity refers to the degree to which independent variables (and only independent variables) affect the dependent variables in a controlled experiment. This is of lesser relevance in industrial case studies, where control over variables is impossible to achieve. However, it should be noted that the architectural inconsistencies uncovered by the approach were not found outside of the sessions, suggesting that the sessions were largely responsible for their identification.

External validity is the degree to which the conclusions of the study are applicable to the wider population of software development contexts. Our study was performed on three commercial systems of two different companies where two real architects and one experienced developer performed in vivo architecture recovery and consistency tasks. The three systems had already been deployed and were of different sizes and ages.

One of them (SC), was an extremely large system and the others were of realistic industrial scale. Hence, the study had high ecological validity, a subset of external validity that refers to the degree to which the study is representative of reality.

However, similar to in-vivo studies, the number of data-points is extremely limited and substantially lessens the external validity of the study in general. We would hope that other researchers could add to the evidence in this area by performing additional in-vivo case studies.

Reliability is concerned with the consistency of the result gathering and analysis. To improve reliability we applied data triangulation where the same data was collected from multiple sources such as the video recordings of the sessions, participant interviews, participant observation and the screenshots collected of the diagrams. In addition, we have documented in detail all the procedures (protocol, and data) in each session so that the case study can be repeated by other researchers. Finally, several of the findings reported here were identified in the two companies, such as the need for defining hierarchical architectural models, and the usability of the drag and drop facilities to create models.

5. RELATED WORK

Passos et al. [8] reviewed the most promising architecture consistency approaches in their 2010 paper. They proposed Dependency Structure Matrices [24], Source Code Query Languages [25] and Reflexion Modeling [14] as possible candidates. Ultimately, in line with our own findings [11] they suggested that Reflexion Modeling with real-time feedback was the most appropriate avenue, based on a well-defined existing process [11].

This work and the ACTool itself is directly based on the Reflexion Modeling work proposed by Gail Murphy et al. [7] who produced a prototype tool (The JRMTool) to illustrate their initial approach [12]. This approach was batch-oriented where a model was defined, the mappings to the code created and an analysis tool was executed to give feedback to the user at periodic intervals. Over the years several enhancements have been suggested for this approach, including hierarchical Reflexion Modeling [19] and augmenting Reflexion Model information with information derived from CVS repositories [26].

Most closely related to this work is the work of Knodel [2]. He has simultaneously developed a real-time Reflexion Modeling-based tool called SAVELife that gives real-time feedback to architects and developers on the consistency between their designed architecture and their implementation. This tool has been trialed on students in an academic context (on an academic project), with largely positive effects. That is, the inconsistencies introduced lessened. However, the students did not feel as well disposed to the approach as anticipated.

Our work builds on our previous work [10] and Knodel's research [2]to provide the first evaluation of real-time Reflexion Modeling in practice. While this reporting of the case studies reflects only on the Architecture Recovery phase of the interventions, the real-time nature of the tooling is evident in the interactions of the participants. Additionally, our literature review suggests that this is the first effort to characterize how the largely successful Reflexion-based modeling is actually used in practice, in terms of the models programmers create, the source code mappings they employ and the facilities they desire.

6. CONCLUSIONS AND FURTHER WORK

In this paper, we have reported our findings of a multi-case study performed to characterize architectural recovery and consistency achieved through Reflection-based approaches, using the ACTool. The architectures of three deployed systems, from two different companies, were recovered and the architectural models, mappings, inconsistencies, consistencies, and tool usability were discussed.

Our findings suggest three main architecture modeling needs to be included for Reflexion-based approaches:

- Templates should be provided in architectural recovery tools for supporting different decompositions. In our case studies two kinds of templates would have facilitated the modeling of the architectures: an N-tiered architectural template and an interface based template.
- Architectural recovery should be scoped beyond the source-code base. They should allow users to include dynamically configured system dependencies, and external elements or third party components where source code is not present.
- A hierarchical modeling capability would be useful. This would require extending Reflexion Modeling nodes to include sub-nodes.

Concerning our findings related to the mapping in Architecture Recovery, the drag-and drop facility, in conjunction with real-time feedback, was found to be useful. It gave programmers the ability to build models in a new incremental manner, directly from the source code structures, rather than defining the model in advance. This allowed quicker recovery and consistency and allowed participants that did not have a complete understanding of the architecture to gain architectural knowledge quickly.

However, two improvements should be included in the ACTool related to the drag-and-dropping mapping facility provided:.

- Users should be offered a drag-and-drop mapping facility **and** a lexical mapping facility. As mentioned, earlier, there are cases where lexical mappings would make the mapping process more efficient.
- Where both modes of mapping are suggested, drag-and-drop should have primacy. We are aware that the coding conventions required for accurate lexical mapping, may not be applied consistently.

Participants should also be made aware of the source code that remains unmapped for each architectural model they create. This would guide the user during an incremental mapping (recovery) process and heightens mapping accuracy.

This paper has also found that users liked the way the tool allowed them to identify inconsistencies and navigate to them in the source code. We hope that this feature will facilitate removal of the inconsistencies. In addition, we have characterized architectural inconsistencies that appeared in our multi-case studies into five kinds. Two of these already appeared in a previous characterization performed in the literature [10]. Thus, this study allowed us to extend this characterization.

In the near future, we would like to replicate the same case study in different companies. This will enrich our external validity. Additionally, the replicated findings encountered in the study will guide us in extending our ACTool implementation.

We are also planning to extend our case study to be a longitudinal one. Our objective is to observe how developers can benefit from the tool while developing software systems and receiving instantaneous feedback on their architectural inconsistencies in their coding view. Intuitively, we believe that developers will not introduce persistent architectural inconsistencies when they receive immediate feedback. This will be the basis of our next case study.

7. ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie), and the CSIS department at University of Limerick.

8. REFERENCES

[1] Bachmann, F., Bass, L., Klein, M., & Shelton, C. 2005. *Designing software architectures to achieve quality attribute requirements*. IEEE Software, Vol. 152 Issue 4 - pp. 153-165.

[2] Knodel, J. 2010. *Sustainable Structures in Software Implementations by Live Compliance Checking*. PhD Thesis, Fraunhofer Institute for Experimental Software Engineering.

[3] Schwanke, R. W., Altucher, R. Z. and Platoff, M. A. 1989. Discovering, visualizing, and controlling software structure. *SIGSOFT Softw. Eng. Notes* 14, 3, 147-154.

[4] Girard, J., Koschke, R., 1997. Finding Components in a Hierarchy of Modules: a Step towards Architectural Understanding. *ICSM* 1997: 58-65

[5] Verbaere, M., Godfrey, M.W. and Girba, T. 2008. Query Technologies and Applications for Program Comprehension, *ICPC 2008*, pp. 285–288.

[6] Koschke, R. 2003. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey, *Journal of Software Maintenance* 15(2): 87-109.

[7] Murphy GC, Notkin D, Sullivan KJ.1995. Software reflexion models: Bridging the gap between source and high-level models. *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 18–23.

[8] Passos, L.T., Terra, R., Valente, M.,Diniz, R., Mendonça, N.C. 2010. Static Architecture-Conformance Checking: An Illustrative Overview. *IEEE Software* 27(5): 82-89.

[9] Rosik, J., Le Gear, A., Buckley, J., and Babar, M. A., 2008. An industrial case study of architecture conformance. *In Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement (ESEM '08)*, pp. 80-89.

[10] Rosik, J., LeGear, A, Buckley, J. Babar, M.A., Connolly, D. 2011. Assessing architectural drift in commercial software development: a case study. *SPE* 41(1): 63-86.

[11] Rosik J., Buckley J., and Babar M.A. 2009. Design Requirements for an Architecture Consistency Tool. *Proceedings of the 21st Working Conference of the Psychology of Programmers' Interest Group*, pp. 109-124.

[12] jRM Tool Eclipse Plug-In. Available at: http://jrmtool.sourceforge.net/

[13] Mitchell M.I. 2004. Research Design Explained. Fifth edition, Thomson-Wadsworth.

[14] Murphy, G.C., Notkin D. 1997. Reengineering with Reflection Models: A Case Study. *IEEE Computer* 30(8): 29-36.

[15] Yin, R.K. 2003. Case Study Research: Design and Methods, 3rd ed., Sage Publications, Thousand Oaks, CA.

[16] Murphy, GC, Notkin, D, Sullivan, KJ. 2001. Software reflexion models: Bridging the gap between design and implementation. *IEEE TSE*; 27(4):364–380.

[17] Knodel J, Muthig D, Naab M, Lindvall M. 2006. Static evaluation of software architectures. *Proceedings of Conference on Software Maintenance and Reengineering*, 279–294.

[18] Le Gear, A., Buckley, J., Collins, J.J., O'Dea, K. 2005. Software reconnexion: understanding software using a variation on software reconnaissance and reflexion modelling. *International Symposium on Empirical Software Engineering*, 34-43

[19] Koschke, R., Simon, D. 2003. Hierarchical reflexion models. *Proceedings 10th Working Conference On Reverse Engineering*, 36 – 45.

[20] Kitchenham B, Pickard L, Pfleeger SL. 1995. Case studies for method and tool evaluation. *IEEE Software* 12(4),52–62. DOI: 10.1109/52.391832.

[21] Runeson, P., Höst, M. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14(2): 131-164.

[22] Corbin, J. M., & Strauss, A. 1990. *Grounded theory research: Procedures, canons, and evaluative criteria*. Qualitative Sociology, 13(1), 3-21. Springer.

[23] Spring Framework Documentation: http://www.springsource.org/documentation

[24] Sangal, N., et al. (2005). Using Dependency Models to Manage Complex Software Architecture, *Proc. 20th Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA),* ACM Press, pp. 167–176.

[25] Verbaere, M., Godfrey, M.W. and Girba, T. (2008) Query Technologies and Applications for Program Comprehension, *Proc. 16th IEEE Int'l Conf. Program Comprehension (ICPC*), IEEE CS Press, pp. 285–288.

[26] Hassan AE, Holt RC. (2004) Using development history sticky notes to understand software architecture.*IPWC*, Bari, Italy, 183. DOI: 10.1109/WPC.2004.1311060.