# Real-Time Reflexion Modelling in Architecture Reconciliation: A Multi Case Study

**Jim Buckley[1], Nour Ali[2], Michael English[1], Jacek Rosik[1] Sebastian Herold[1]**

[1] The Irish Software Engineering Research Centre (Lero), University of Limerick, Ireland
[2] School of Computing, Engineering and Mathematics, University of Brighton, UK

***Abstract***
**Context**
Reflexion Modelling is considered one of the more successful approaches to architecture reconciliation. Empirical studies strongly suggest that professional developers involved in real-life industrial projects find the information provided by variants of this approach useful and insightful, but the degree to which it resolves architecture conformance issues is still unclear.

**Objective**
This paper aims to assess the level of architecture conformance achieved by professional architects using Reflexion Modelling, and to determine how the approach could be extended to improve its suitability for this task.

**Method**
An in-vivo, multi-case-study protocol was adopted across five software systems, from four different financial services organizations. Think-aloud, video-tape and interview data from professional architects involved in Reflexion Modelling sessions were analysed qualitatively.

**Results**
This study showed that (at least) four months after the Reflexion Modelling sessions less than 50% of the architectural violations identified were removed. The majority of participants who did remove violations favoured changes to the architectural model rather than to the code. Participants seemed to work off two specific architectural templates, and interactively explored their architectural model to focus in on the causes of violations, and to assess the ramifications of potential code changes. They expressed a desire for dependency analysis beyond static-source-code analysis and scalable visualizations.

**Conclusion**
The findings support several interesting usage-in-practice traits, previously hinted at in the literature. These include 1) the iterative analysis of systems through Reflexion models, as a precursor to possible code change or as a focusing mechanism to identify the location of architecture conformance issues, 2) the extension of the approach with respect to dependency analysis of software systems and architectural modelling templates, 3) improved visualization support and 4) the insight that identification of architectural violations in itself does not lead to their removal in the majority of instances.

**Keywords:** Reflexion Modelling, Software Architecture, Architecture Consistency, Architecture Conformance.

## *1 Introduction*

Software Architecture aims to ensure prioritized non-functional requirements like maintainability and modularity are satisfied through appropriate macro-structuring of software systems [1]. However, often systems grow without an explicitly defined architecture, or have drifted over time from their originally designed architecture [2-3]. In such cases, it is unlikely that the desired non-functional requirements have been delivered.

Early work towards addressing this issue focussed on architecture recovery: deriving the software's architecture from the source code of the existing system, typically from the source code dependencies of these systems [4-5]. However, even though software architects were often allowed to confirm or refute the suggestions of such analyses, they did not drive the process, thus limiting their ability to impose their desired architecture on the system [6]. In addition, these approaches often suffered from the 'garbage-in, garbage-out' phenomenon [7], whereby any architecture derived from analysis of a system without an initial architecture (defined or adhered to) is likely to be flawed.

More recently approaches to retro-fit the intended architecture (as originally or retrospectively defined) onto systems, have been developed to address these issues. This work is referred to in the literature as architecture reconciliation [7] architecture conformance [3] or compliance checking [9]. Several techniques have been proposed in this area, ranging from allowing architects probe the architecture of specific points in the system [10-11] by defining textual rules, to more system-encompassing specifications like Reflexion Modelling (RM). In Reflexion Modelling [12], for example, the architect is initially prompted to explicitly state their ideal (as-intended) architecture for the system, as a simple vertices-and-edges diagram in which vertices represent architectural modules and edges represent the expected/allowed dependencies between these modules. The architect is then asked to map elements of the source code to the vertices in this as-intended architecture.

The approach parses the system to identify dependencies between the source code elements mapped to different vertices in the as-intended architecture, thus allowing corroboration or contradiction of the relationships proposed by the architect with respect to the as-implemented system. It is anticipated that the architect would then update the system to more fully align it with the as-intended architecture. Such an approach allows the architect drive the process from their architectural perspective, and focuses them on the parts of the system where specific architectural violations arise in the implemented system with respect to this perspective [13-14].

Several implementations of RM variants have been empirically evaluated through case studies [15-18], largely achieving positive feedback from industrial practitioners. Specifically, software architects have shown great enthusiasm for the information which it provides [13, 19] and have even proactively sought such analysis again [20]. However, most of these studies focus on the ability of RM to identify inconsistencies [21-22] but do not concentrate on how the capabilities and insights provided by RM approaches are utilized by software practitioners and if the violations are subsequently removed.

This research assesses how the capabilities and insights provided by *Real-Time Reflexion Modelling* (RT-RM) approaches are utilized by practitioners. Here RT-RM [18] refers to a variant of RM where new architectural violations are presented to the architect, as new mappings are made between architectural modules and source code elements.

The research assesses how RT-RM is leveraged, determines if RT-RM results in the removal of architectural violations and also identifies the ways in which the participants would like to see RT-RM adapted/evolved in the future. It addresses these questions through a usage-analysis of a Just-In-Time Tool for Architecture Consistency (JITTAC) [34], which embodies a RT-RM approach. The empirical analysis is performed over five in-vivo case studies in four different commercial organizations.

This paper is a substantial extension of the work presented by several of the authors in [18]. That paper reported on the first three case studies, providing a characterization of the modelling practices of the participants involved, their mapping (source code to RM vertices) preferences, and a characterization of the violations identified. In comparison this paper:

- Incorporates a larger data set, including two more case-studies, and retrospective interviews with the participants from all five case studies. This provided us with a larger, more representative data-set on which to base our findings and with the ability to explicitly assess the outcome of the RT-RM intervention longitudinally;

- Provides an expanded analysis of the data-set, resulting in several new findings. Specifically, while it does re-assess the modelling, mapping and architectural violations issues expounded upon in [18], with respect to the enlarged data set, it also includes findings on iterative analysis of systems through Reflexion models, findings on the outcome of RM in terms of the systems' architectural violations and an extended set of requirements for RM going forward.

- Presents an extended review of the related literature in this area (Section 2) which discusses the alternative approaches adopted and empirical work carried out in this and closely related areas.

The paper is structured as follows. Section 2 discusses the current state of the art with respect to various architecture conformance approaches. Section 3 focuses on one particularly successful approach: RM, discussing the approach, its empirical evaluations to date and how RT-RM builds on the approach. Section 4 describes the five case studies that make up the empirical component of this paper. Sections 5 and 6 present and discuss the findings across these case studies, with section 7 examining the threats to validity. Finally, section 8 concludes the paper.

## *2 State of the Art*

When a software system's implementation diverges from its designed architecture, it is referred to as *architectural drift* or *architectural erosion* [8, 2]. This usually happens during software evolution, when the software undergoes changes as a result of bug fixes and updates, but may also happen during initial implementation of the system [23]. Architectural drift may result in the goals associated with the system's as-intended architecture being lost [20-21], often with serious consequences [3, 24-25].

Architecture conformance aims to address architectural drift. It has been defined as a process of ensuring continued conformance of a subject system's implementation to its architectural design documentation and goals [26]. Here, design documentation is defined as any artefact created during the system's design (sometimes, even after the code is written) that documents the system's architecture. There have been many approaches suggested to increase architectural conformance, and these can be classified into several schools [8]. This section, reviews several of these approaches.

Tvedt Tesoriero et al. [16] divide architectural evaluation work into two main areas: pre-implementation architecture evaluation and implementation-oriented architecture conformance. In their classification, pre-implementation architectural evaluation involves the analysis of a proposed architecture to check whether it will fulfil the optimum number of the system's desired requirements. These approaches are used by architects during initial design and provisioning stages, before the actual implementation starts.

In contrast implementation-oriented architecture conformance approaches assess whether the implemented architecture of the system matches the intended architecture of the system [16, 22, 26]. Specifically, whereas architectural evaluation assesses the quality of the *proposed* architectural design, architectural conformance assesses whether the *implemented* architecture is consistent with the proposed architecture's specification, the goals of the proposed architecture, or both. Implementation-oriented conformance approaches can be split into two categories [27]:

- **Conformance by Construction**: These approaches strive to achieve conformance through automated or semi-automated generation of artefacts, composing the system from the architectural descriptions. Several, established approaches implementing conformance by construction exist already. For example, approaches such as: generative programming [28], round-trip engineering [29], and model driven development [30-31] are being used in commercial software development. However, it is more difficult to apply these approaches retrospectively on existing systems: while model transformations have the ability to map from the implementation back to architectural models, these transformations are predefined and usually reflect well defined patterns between the platform independent models and the platform specific ones. This somewhat constrains the mapping between the existing system and the architects' intended architecture as usually the mappings between the code and the models in existing systems do not follow strict patterns. In addition, when applied in the forward direction, these approaches only generate partial implementations of the system. Hence assessment of conformance still needs to be checked as the implementation is completed [32].

- **Conformance by Extraction:** These approaches analyse artefacts, or partial artefacts, of the implementation process itself (for example, source code dependencies) and/or artefacts that are available after the system's implementation (for example dynamic traces). A comparison of these artefacts with the system's as-intended architecture is then performed through textually-specified or graphically-specified rules and mappings. These comparisons facilitate software engineers' identification of potential discrepancies and violations. Most of the conformance-by-extraction approaches currently reported on have been applied retrospectively to software systems after deployment.

Conformance by extraction techniques can be further split into three categories, based on the analysis they employ [33]:

- ***Static Architecture Conformance*** is based on static techniques that analyse different assets produced by the implementation process, such as source code and data structures. Static analysis may be performed without a fully running system and thus allows conformance testing over a wider span of software lifecycle phases. Most of the techniques presented in the literature are static architecture conformance techniques [14-17, 21, 33-34]

- ***Dynamic Architecture Conformance*** uses run-time analysis techniques on an executing system. Thus, an instance of a running (and probably almost fully implemented) system is required. For example, de Silva and Balasubramaniam describe a non-intrusive approach to architecture conformance checking at runtime [35]. In another example [36], execution traces of a running system have been extracted as coloured petri nets which were investigated for architectural violations. The work of Popescu and Medvidovic [37] focuses on message-based systems and integrates recorders into components of such systems that are used to evaluate consistency between actual occurring events and prescribed events.

- ***Combined Static and Dynamic Architecture Conformance*** is based on applying both type of analysis, usually in sequence. An example is the Pattern Lint work done by Sefika et al. [38] where static analysis, followed by dynamic analysis was used. This complimentary dynamic analysis was used to detect the prevalence of violations, as identified through static analysis, in the executing system.

These architecture conformance approaches can also be categorized by the level of support they offer to the architect: The first can be loosely referred to as visualization tools that allow the architect to view information about the structure of the system, but not as an explicit architectural model. For example, Klocwork [39] provides various forms of dependency analysis, quality metrics and visualisation support that provide useful guidance for architecture conformance.

The second category allows the architect to declare their desired architectural model, either graphically [15] and/or through the definition of individual rules [33, 40-41]. Semmle, for example, uses a Source Code Query Language (SCQL) called .QL which, while expressive in defining constraints, is limited in its architecture-visualizing ability and thus its ability to guide reasoning about overall software architecture abstractions [42]. In contrast, graphical modelling approaches (like RM) allow the architect to create a vertices-and-edges diagram of the intended architecture and, to visualize the implemented system in the context of that intended architecture, although typically with less expressive constraints.

Passos et al. [42] presented an overview of three approaches: a rule-based approach [10], an RM type approach [43], and an approach that allowed architects to visualize the system as a hierarchical dependency matrix of source code elements [44] with architectural rules embedded. They recommended RM for organizations interested in systematically incorporating architecture conformance checking into their software development process, due to a well-defined architecture conformance process, centred on holistic high-level models as defined by architects. Indeed, several commercial tools have now been developed that incorporate similar functionality [45-46] and it is this type of approach that is the focus of this research. As such the next section will describe RM in greater detail.

## *3 Reflexion Modelling*

The traditional RM process as explicitly outlined by [13] was adapted by Rosik et al [20] to facilitate the application of RM during system implementation, as well as system evolution:

1. Before implementation of the system commences, the designer creates a hypothesised architectural model: the as-intended architecture.

2. During the implementation phase, developers and/or architects, frequently update a set of mappings which assign newly implemented source code entities to the entities in the as-intended architecture.

3. At any point during implementation or subsequent maintenance, a dependency graph of the system's sources can be extracted by parsing the system, creating a *source model* (referred to as the "as-implemented architecture" from this point on).

4. The relationships defined by the engineer in the as-intended architecture are compared with those extracted from the as-implemented architecture. Results of that comparison are presented to the developer through the Reflexion model periodically. The following relationships are represented in this model:

    - A solid edge represents a relationship present in both, the as-intended architecture and the as-implemented architecture (convergence).

    - A dashed edge represents a relationship present in the as-implemented architecture, but not present in the as-intended architecture (divergence).

    - A dotted edge represents a relationship present in the as-intended architecture but not present in the as-implemented architecture (absence).

5. By analysing the Reflexion model, engineers can become aware of architectural drift issues. Of most interest are the divergences where there are dependencies in the source code unexpected by the original architects. Absences may also be of interest but alternatively, they may just reflect places where the software is incomplete (when the technique is applied before the implementation is finished). In addressing divergences, architects may choose to take one of the following actions (derived from [21]):

    - The inconsistency may be corrected by updating the code base (changing the as-implemented architecture);

    - Mappings between the source code and the as-intended architecture may be updated, for example: reassigning an implemented entity to a different as-intended entity;

    - The implementation may be considered acceptable and the as-intended architecture may be updated accordingly.

Steps 2, 3, 4 and 5 are continuously repeated over time, towards prompting increased system conformance to the as-intended architecture.

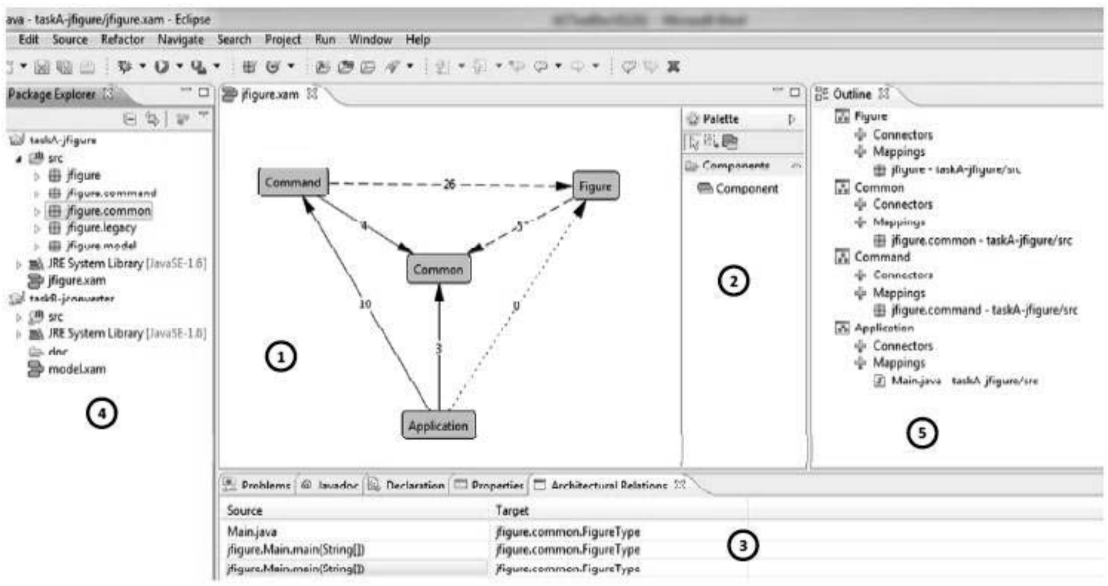
To further increase conformance, several groups [47-49] have proposed more timely violation detection in RM. For example, developers could be made aware of the violations they introduce with respect to the as-intended architecture as they code, via margin alerts at compile time and code assist [34]. We refer to these notifications as Just-In-Time (JIT), in that they notify the developer immediately after introduction, before the violation gets embedded/accepted in the code-base. This approach is in accordance with the findings of Layman et al [50], who suggest that the longer an issue persists in the code-base, the harder it is to fix.

Similarly, architects could get real-time notifications as they reconcile the architecture (RT-RM). That is, they can immediately be made aware of violations that arise based on new mappings they make between their as-intended architecture and the source code. These real-time alerts do not address violation embedding/acceptance, but instead are aimed at allowing the architect to explore the mapping more interactively during initial architectural reconciliation. It is this variation of RM that is the focus of this paper.

### *3.1 JITTAC: A RT-RM Tool*

A screenshot of the JITTAC tool, which was used in the case-studies reported on here, is presented in Figure 1. JITTAC stands for a "Just-In-Time Tool for Architecture Conformance" [34] and, as the name suggests, has JIT and real-time notification facilities. Only the real-time notification facilities were employed in these case studies.

As shown in Figure 1, JITTAC allows the architect to define an architectural model of the system (1) where components and their connections, can be dragged and dropped from a palette (2). Additionally, drag and drop facilities can be used to create mappings from the existing source code elements in the package explorer (4) to the components in this architectural model and a summary of these mappings is available in an outline view (5). Many source code elements can be mapped into one architectural component and the architectural models and mappings can be defined incrementally and iteratively. In addition, source code elements can be dragged directly onto a blank space in the canvas to create a new component.



| Source | Target |
|---|---|
| Main.java | jfigure.common.FigureType |
| jfigure.Main.main(String[]) | jfigure.common.FigureType |
| jfigure.Main.main(String[]) | jfigure.common.FigureType |

**Figure 1:** JITTAC

As mappings are defined between the source code and the architectural model in these ways, the results of the RM analysis are presented in real-time. JITTAC conforms to the RM process, as declared above, using dotted edges to represent relationships defined in the as-intended architecture, but not present in the implementation (absences), dashed edges to represent relationships present in the as-implemented architecture but absent in the as-intended architecture (divergences) and solid edges representing those relationships that are present in both (convergences). Typically, the architect will focus on the dotted and dashed edges in their efforts to address architectural drift.

The tool allows for further analysis of divergent edges. Specifically when the edge 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 “Figure” and “Common” in Figure 1). JITTAC then allows the architect to click on the Source in the Architectural Relations view, to navigate to the associated source code, which has an alert posted in its margin. If that code is changed to address the inconsistency, this change gets reflected back, removing the margin alert and updating the architectural model instantaneously: the divergent edge becomes a convergent one.

### *3.2 Empirical Assessment of RM Approaches*

Many empirical studies have been carried out assessing the original (non real-time) RM. In one of the first case-studies reported, Murphy et al. compared the layered (intended) architecture of a program restructuring tool with the actual implementation (implemented architecture) [15]. In another case-study presented in the same paper, RM was applied to the kernel of an experimental operating system developed at the University of Washington. Both of these case studies were on prototypes developed by students/researchers in the group and the system-creators retrospectively changed their systems to address the architectural violations.

Another RM approach was used by Tran et al. [21], with the goal of retrospectively repairing the architecture of two open source systems: the Linux Kernel and an editor called Vi iMproved (VIM). Their goal was to remove all anomalies excepting those that were either too risky or 'not worthwhile'. However in the first instance, they did not provide feedback to the original development team and, even though they did present their findings to the original author of the VIM editor, no comment is made on whether the author addressed the identified violations or not.

An RM process in a series of works [9, 16, 22, 26] was used to perform a series of evaluations on a software system being re-implemented, called the Experience Management System. In these studies the as-implemented view was shown to be useful in demonstrating to management that the project needed restructuring. However, their approach was manual and the desired architecture could not be achieved. Instead, they were able to document violations that would normally have been overlooked. An improved version of their technique has been used in another case-study of a Simulation and Analysis Tool [16]. However, as in the original study in this area [15], the violations identified in these evaluations were again addressed by members of the research team, who were also the authors of the system under study.

Knodel et al. [17, 33] built and used an RM based tool called SAVE (Software Architecture Visualisation and Evaluation Tool) for the purpose of architecture evaluation. The tool extended RM with hierarchical modelling capabilities as proposed by Koschke and Simon [51]. It was used to analyse software systems' architectures in a number of evaluation scenarios: academic, open source or commercial software systems of varying sizes, ranging from 10 to 600 KLOC [17]. Again, no reference was made to the system's subsequent evolution. Similarly, Knodel and Popescu evaluated a proposed extension to RM where the subsequent architecture conformance actions performed after these techniques were applied was not mentioned [33].

Kolb et al. [19] and Knodel et al. [52] also report on experiences using SAVE with an industrial partner. In this instance, conformance checking was adopted by the company involved as a standard instrument for ensuring higher quality products at the organisation. Identified violations, in already deployed systems, were fed back to the development team and a formal process adopted whereby these inconsistencies were removed. The results presented show a promising trend: a decrease in the number of inconsistencies over the product's life time. It should be noted however that, due to the product line [53] nature of the products involved, there was probably an agenda of heightened Architecture Conformance in the development teams.

In contrast, a more recent, longitudinal case-study in IBM, [20] showed that periodic (four-five monthly) identification of architectural violations, over two years, using RM did not serve to lessen architectural drift in a less formal (non-SPL) development context: During the study no violations were removed from the system based on the insights provided by the RM approach. The authors proposed Real-Time RM to address this by alerting developers as they introduced violations into the system and thus pre-empting retrospective remedial action.

Empirical evaluations of real-time architectural violation feedback are scarcer. Eichberg et al. [49] concentrated on the performance of the (real-time) algorithm employed. Knodel et al. [48] evaluated it on M.Sc. students and Mattsson [11] evaluated a Model Driven Development approach to the problem. An evaluation of the approach’s performance tells us little about its effectiveness as an Architectural Conformance technique. A student-based study has lesser ecological validity and the Model Driven Development approach employed by Mattsson [11] can only be employed during initial system development, as it is currently formulated.

## *4 Empirical Study*

The empirical study reported on here is a multi-case study of RT-RM. These are in-vivo case studies involving commercial organizations and their commercial software products. The study is motivated by the lack of empirical studies assessing how RM is used and leveraged in practice, particularly with respect to architectural violation removal.

### *4.1 Motivation and Research Questions*

Of the studies reviewed in section 3.2, none characterize participants’ usage of RM. Hence, there is limited guidance as to how the modelling and mapping facilities available to participants are used during RM (an exception being [18]). This is surprising given the popularity of such tools in practice, and the lack of reported results of this kind leaves open the possibility that current approaches to architectural reconciliation may be sub-optimal for their users.

This paper attempts to address this issue by observing architects employing the RT-RM technique during architectural reconciliation, highlighting the different modelling approaches and mapping approaches used by these architects. The goal is to identify potential improvements and thus to hone the approach going forward. Hence, the first research question posed is:

*RQ1: What system perspectives are of particular interest to architects in Real-Time Reflexion Modelling during architecture reconciliation?*

The architects may be interested in perspectives of the system that emphasise its structural or functional aspects, or even its cross-cutting concerns. Likewise, they may be interested in specific subsets of the system or the whole system. If the latter, then are scalability-handling measures are required? Without empirical evidence on the perspectives of interest to the architects, these questions remain open and this paper sets out to address this gap.

The second research question is:

*RQ2: What facilities of RT-RM are of interest to architects during architecture reconciliation?*

Given the lack of empirical evaluations concerning how RT-RM is performed, it is possible that the real-time feedback provides no additional utility. Alternatively, given real-time feedback, architects may find it useful to explore their models interactively to identify violations, or to model potential changes to the source code through Reflexion model manipulations (and the associated immediate feedback that RT-RM offers). Thus, this second research question probes the potential added-value of real-time information in RT-RM.

The evidence presented in previous research (see section 3.2), regarding violation removal as a result of violation identification through RM is mixed. In many cases, data on the removal of violations is simply not presented, or they have indicated that violations are removed by research team or not removed. No studies have looked at violation removal as a result of applying RT-RM, even though this adaptation has been proposed as potentially elevating violation removal. Therefore, our third research question is:

*RQ3: Does real-time architectural violation identification prompt the removal of violations in commercial practice, during architectural reconciliation and, if so, where is the locale of change?*

Here, 'locale' refers to the source code, the as-intended architecture or the mapping generated by the architect. Given that the probable answer to this research question is that some violations are removed and others are not, this paper also assesses the factors that influence the removal of these violations in the organizations. Hence the paper poses its final research question:

*RQ4: What are the factors that influence the removal of violations during RT-RM architectural reconciliation, in commercial software systems?*

### *4.2 Empirical Design*
An in-vivo, multi-case-study protocol [54-55] was adopted across four financial services organizations. These organizations have a large Irish-based presence, with large software portfolios and all were interested in heightening their systems' quality through architecture conformance checking.

The companies were responsible for selecting the participants and the target systems for these case-studies, within the constraints that the systems had to be developed in Java (to facilitate our tool support), that they should be over 25KLOC (to heighten the reality of architecture conformity as a concern) and that the participants were experienced developers, familiar with the architecture of the systems. We requested that the participant ideally be the software architect responsible for the system. Three of the companies selected one system for analysis and one company selected two systems for analysis, resulting in five separate case studies.

#### *4.2.1 Systems*
Table one provides some characteristics of the software systems chosen by the companies, in size order. The first three systems are an order of magnitude greater in size than the other two, as evidenced by all the size indicators presented. As would be expected, the number of developers also reflects this size gradient, although it should be noted that the number of developers refers to the participants' estimate of all the developers who have contributed to the code-base over the lifetime of the software, and so is inexact. Note also that the three systems presented first are much older than

the others, suggesting that there might be a greater number of architectural violations in these systems. Systems labelled 4a and 4b came from the same Financial Services company.

| Case | System Domain | Size | | | No. of Developers | Age (years) |
|---|---|---|---|---|---|---|
| | | KLOC | Java Files | Packages | | |
| 1 | Claims Mgt | 2,223 | 7,300 | 1,105 | >30 | >10 |
| 2 | Banking | 1,800 | 6,289 | 956 | ~18 | 8 |
| 3 | Underwriting | 1,200 | 3,347 | 486 | >20 | >10 |
| 4b | Marketing | 75 | 239 | 124 | 11 | 2 |
| 4a | Banking | 38 | 173 | 25 | 6 | 1.25 |

**Table 1:** The Software Systems Used in the Case Studies

### *4.2.2 Participants*

Table 2 characterizes the participants involved in the study, in terms of their experience and current role. All the participants were experienced Software Engineers and four of the five were the architects responsible for the systems under study during these sessions. In the fifth case (4a) there was no explicit architect for that product as agile development was practised in that team. Instead, all the team had architectural responsibilities for that system and this participant was selected by the Team-Lead for participation, based on his architectural knowledge of the system.

The distinction between "Product Architect" and "Chief Architect" is that the first and third participants were the Chief Architects for all software products that the Irish-based subsidiaries were responsible for whereas the second and fifth (4b) participants were responsible for the specific software system under study. Participant 4b also had a larger corporate-wide role in a team that defined the umbrella architecture of the company's software portfolio.

| Case | Years S.E. Experience | Current Role | Years Experience as Architect |
|---|---|---|---|
| 1 | 22 | Chief Architect | 12 |
| 2 | 8 | Product Architect | 3 |
| 3 | 12 | Chief Architect | 8 |
| 4a | 5 | Senior Developer | 0 |
| 4b | 16 | Product Architect | 2 |

**Table 2:** The Participants

### *4.2.3 Study Protocol*

A 20 minute demo of the RT-RM tool was given to each of the participants. The objective of the demo was to show the architecture reconciliation 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. All of the functionality used by the participants in their sessions, is as per the RT-RM approach description in section 3.1.

After the demo, the participants installed the tool 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, more typically, part of the system. Four of them (1, 2, 3, 4a) used the tool to do this but participant 4b used a paper-based model he had created in advance of the session. He then replicated his model in the tooling. The authors acted as observers of the sessions and aided in any technical support that was required for the tool. Each architecture reconciliation session lasted more than an hour. In the case of case study one it lasted one hour and forty-four minutes, but all other sessions lasted less than 90 minutes. Each session included a number of iterations, where the participants focused on increasing the depth or scope of their architectural models. That is, each iteration consisted of adding new components and/or mappings and probing arising violations. On average 4.2 iterations were employed per case study. The participants were interviewed several months after their session, to reflect on their experiences and to determine the actions they had taken as a result of their sessions.

#### *4.2.4 Data Collection*

The participants first answered questions to characterize their systems and their own professional experience. Then they started to use the tool for architecture reconciliation 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, and the participants' observations on their tool usage. Think aloud data was gathered in line with the three best-practice guidelines suggested by Ericsson and Simon [56]: the researcher should ask participants to report verbally *everything* that comes into their mind, *as* it comes into their mind and prompt them when they fall silent. Their utterances often fell into the category of them explaining the architecture and architectural violations to the observer. At the end of the session, the participants were asked open questions about the value of the results obtained, and the tool's usability. Screenshots of the architectural models that participants evaluated and a tool-produced file of the mappings defined by the participant were also captured.

Several months after the original session (ranging from four-to-seven months) the participants were interviewed to assess the impact that the approach had on their systems and to assess the validity of our findings with respect to their session. Thus the data-set for analysis consisted of:

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

- Transcripts of the participants' interviews

The gathered material of the sessions amounted to over seven hours of video recordings which was transcribed and, together with other material, thoroughly analysed and discussed to identify and record important findings regarding their modelling and mapping practices. This analysis initially consisted of an open-coding-like phase, in the spirit of Corbin and Strauss [57], where the analysis was informed by knowledge derived from the literature and the earlier case study results. This was done independently by two of the authors and subsequent discussions between these authors focused on the reliability, veracity and relevance of their independent findings for the research questions identified in Section 3.1. The findings were later presented back to the participants for verification and the results presented here are the outcomes of these researcher discussions/participant verifications.

Additionally, transcripts of participants' post-session interviews were analysed to see what action, if any, participants had undertaken in response to their architecture reconciliation sessions.

## *5 Findings*

In this section, we discuss our findings and come up with several observations to answer the research questions presented in section 4.1. Each subsection, discusses one of these research questions.

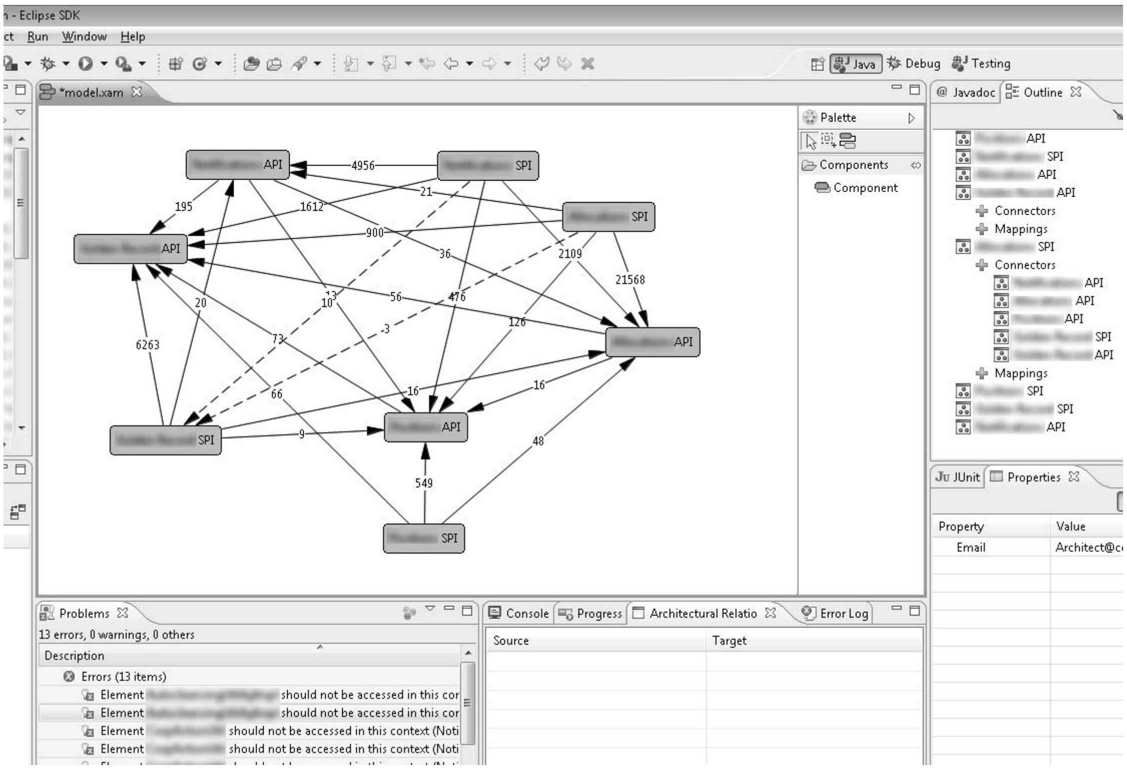
### *5.1 What system perspectives are of particular interest to architects in Real-Time Reflexion Modelling during architecture reconciliation? (RQ1)*

#### *5.1.1 Architectural Style Perspectives*

The participants in case studies 1-3 (henceforth called Participants 1, 2 and 3 respectively) used the approach to check the functional partitioning of their systems: specifically to assess if communication between these partitions was exclusively through defined interfaces. Each of the partitions was thus modelled in the intended architecture as an interface vertex which (should) provide the only access to their associated implementation vertex. Figure 2 shows one such model generated by Participant 2 on his fifth iteration. In this diagram the 'API' vertices represent interfaces and the 'SPI' vertices represent implementations - a naming convention in the company in question. Note that the violations in this diagram are where SPI vertices make direct calls on another SPI vertex.

In contrast, the architectural model used by Participant 4a and 4b was an N-layered architectural style. This can be seen in the model created by Participant 4b in his third iteration, as shown in Figure 3. Here there are four layers with the first and second layer broken down into two sub vertices. Note that in this model, the edge that spans more than one level in the layered architecture, is perceived as a violation, as would be expected in this layered style.

These behaviours suggest an observation ***(O1): architectural templates should be made available for architectural reconciliation sessions supporting different clichéd decompositions.*** Specifically, given the unprompted architectural styles adopted by the participants in these studies, an N-layered template and an interface-based template should be provided. In the former template exclusive access to different tiers could be assumed to be through adjacent tiers only, thus allowing default detection of convergent and divergent edges in the model. In the interface template exclusive access to implementations through interfaces could be presumed and again this, in conjunction with source code analysis, could be used to automatically detect violations.



13 errors, 0 warnings, 0 others

| Description |
|---|
| Errors (13 items) |
| Element [redacted] should not be accessed in this cor |
| Element [redacted] should not be accessed in this cor |
| Element [redacted] should not be accessed in this context (Noti |
| Element [redacted] should not be accessed in this context (Noti |

**Figure 2:** An Interface-based model from Case Study 2

This latter template suggests a specialist interface type vertex and interface-type edge: an extension of the port-type Component Access Rules proposed by Knodel and Popescu [33] for RM. This vertex/port could be responsible for checking that any element calling a method of the interface's implementation is doing so through the interface. Source code analysis could also reveal code that is located within the implementation vertex but that has interface-type properties and could prompt for its relocation. Interface vertices could also be useful in COTS (Components Of-The-Shelf) situations where the static analysis underpinning the tool does not have access to the source code of the component's implementation but may have access to its API, or may be able to deduce calls on components' APIs based on naming conventions. This would extend modelling beyond the source code of the in-house system itself (see section 5.1.3).

Both interface vertices and interface-edges could be distinguished in such models, as suggested by Prujit and Brinkkemper [58], to illustrate their interface-nature, and thus facilitate communication of the architecture with the wider development team.

### *5.1.2 A Feature Oriented Perspective*

Interestingly three of the five participants expressed a desire to model their system based on *features*, where their definition of features differed from participant to participant, but generally reflected some user-functionality of the system, in line with the definition in Wilde and Scully [59]. This was most apparent in case study 2 where, after his study, the participant stated:

*"It would be really neat if each node reflected some sort of user goal. It would be difficult to map (code to user goals), but it'd be great. Then we could see all the relations between features…"*

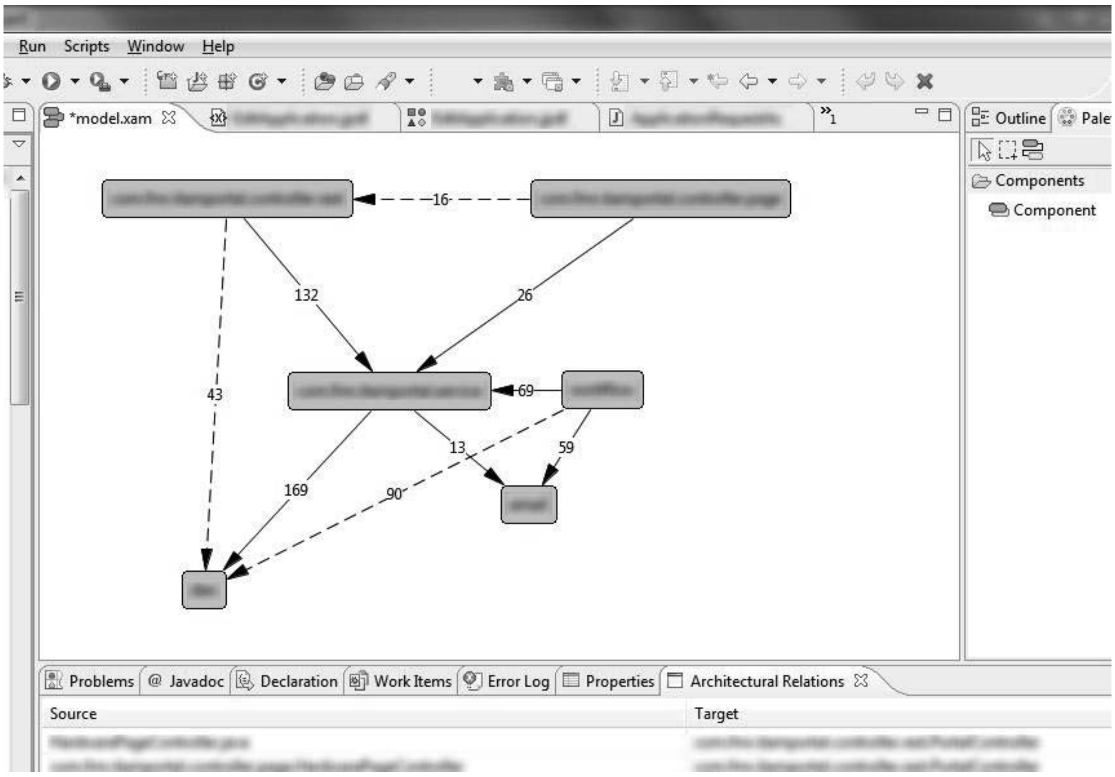


**Figure 3:** Reflexion Model from Case Study 4b

This sentiment was also echoed by Participant 1 in passing and Participant 3. This latter participant stated:

*"We did some work a while ago modelling part of our system in terms of user functions and this (RT-RM) could be used with that. Of course it's changed now (the modelling they did) and we'd need to do some work to get it back…, but it wouldn't be too out of date… It (RT-RM) would be really nice for that"*

These quotations suggest a desire for a complimentary view in RM whereby the vertices are used to represent features and the edges, dependencies between features. This is plausible because some RM tooling allows several concurrent models of the software system to be preserved at once [34] and a feature-based model could be one of these. Admittedly mapping code to feature-based vertices would be difficult, based on their delocalization in the code-base [60] the finely-grained nature of this delocalization and the difficulty in locating features in code, in general. These observations suggest that ***(O2) the current mapping facilities (drag-and-drop from the package explorer, as in JITTAC or regular-expression descriptions, as in the jRMTool [61] ) may need to be enhanced to help achieve this mapping*** and that ***(O3) the envisaged tooling might benefit from providing support for feature location [62].***

### *5.1.3 A System-wide Perspective*

Another modelling issue that was noted in three sessions was the participants' desire to model beyond the static code-base. They wished to determine the code-base's relationship to dynamically configured elements (Javascript for example) and to include 3rd party components. For example, participants 1 and 4b were particularly interested in identifying calls to a 3rd-party component and capturing events that the system listened to, from that component. Participant 4b, referring to a Business Process Manager component of interest said:

*"the web service layer would have had some logic that would have created a workflow instance… using calls into a JPBM API… not code we own. The JPBM then triggers events into this workflow package and there's a relationship from here to here and back to workflow that I know exists… and I'm struggling to see how the approach is going to capture that"*

It would be possible (indeed easy) to capture the calls to an external component via an interface-type vertex in the Reflexion model, if lexical conventions were adhered to in invocations of that particular interface. In contrast, identifying the events that the system listens to through, for example, a Spring framework is a much more difficult proposition, as is capturing any dynamically configured relationships through a layer of indirection. However, the expressed desire of three of the five participants was that, given the composition of today's software systems, ***(O4) architectural reconciliation should be scoped beyond static source-code analysis.***

### *5.1.4 Hierarchical or Partition-based Perspectives*

Over the architecture-reconciliation iterations, the participants generally moved to more encompassing architectural models or deeper architectural models, where elements of their earlier models were probed in more detail. The latter approach can be seen in Participant 4a and Participant 4b's sessions where they started off with one Reflexion model node per layer in their architectural model and then proceeded to divide the layers into individual components, as shown in Figure 3. Alternatively, the participants who adhered to a more interface-oriented model initially took a small set of interface-based partitions and expanded that set over their iterations. In all three cases, the models grew to a point where they were somewhat difficult to read, as illustrated by Figure 4, which is Participant 2's final model from the session. It should be noted that this 'final' model still left the majority of the system in a 'Rest-of-System' node at the bottom right hand corner. This suggests that the approach should ***(O5) explicitly deal with scalability issues.***

JITTAC has the ability to grey out user-specified vertices and their incident edges, as illustrated in Figure 5 (from Participant 1, during Iteration 3), and this is particularly useful when the system contains utility vertices that are widely invoked. Alternatively JITTAC can grey out all nodes not adjacent to a selected node, allowing the architect to focus in on one part of the system. These improvements were noted by several of the participants:

*"Ah, that's nice" (referring to the hiding facility)*

*"Better clean this mess up a bit. How do I em... grey 'em again?"*

But all participants suggested that additional improvements be made to address the scalability issue. The 'increasing depth' approach suggests that a ***hierarchical modelling capability would be useful*** and this was explicitly requested by one of the participants (4b). Indeed, this suggestion also compliments the observation in section 5.2.1 that participants often wanted to "dig into" specific nodes at more detail, to find the source of the violations.

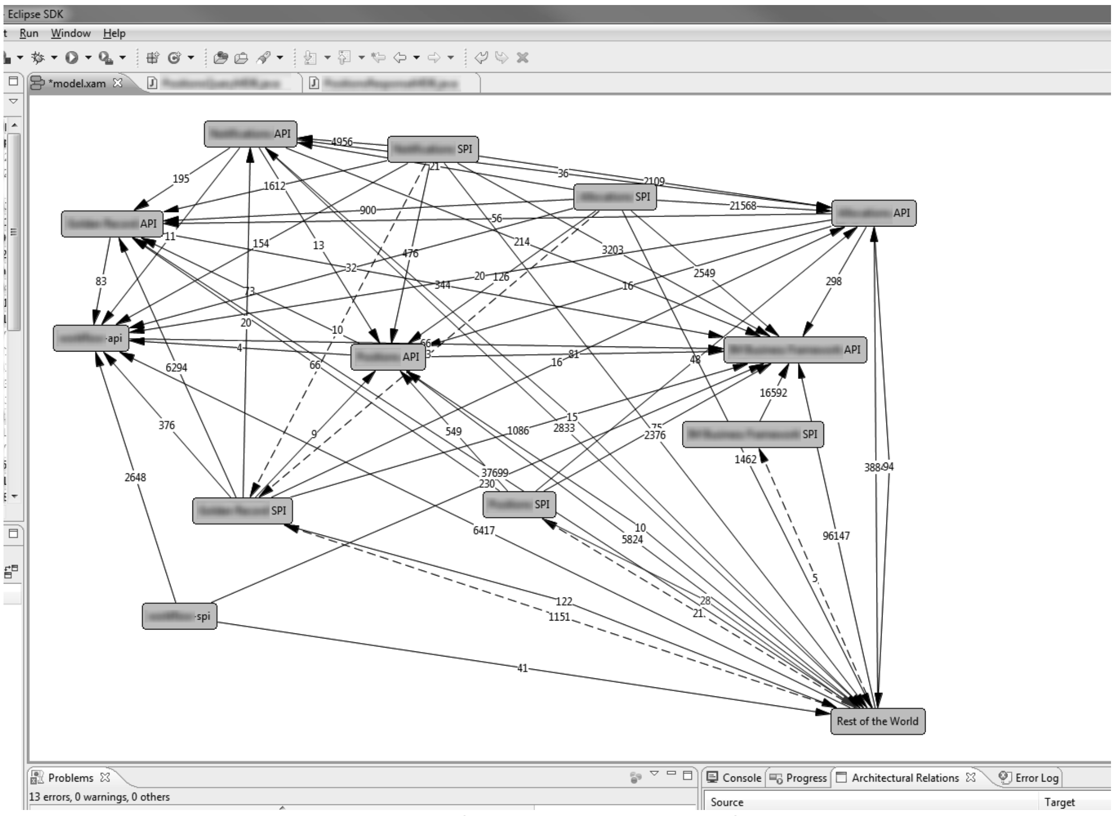


**Figure 4**: An illustration of the Scalability issues faced by participants

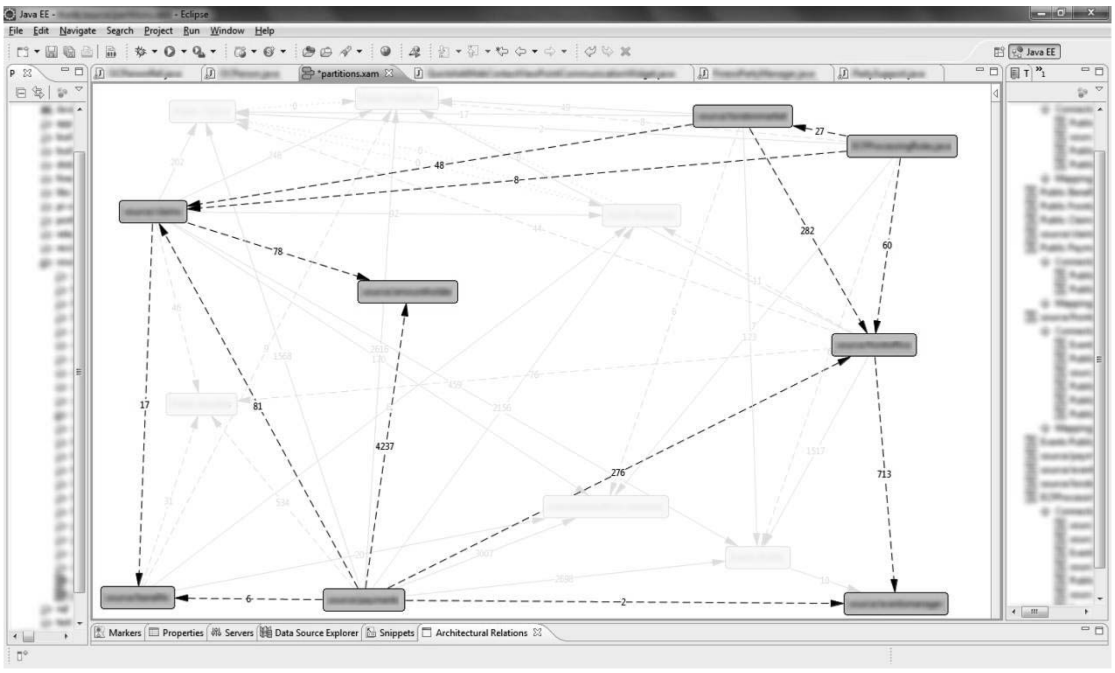

Java EE - Eclipse
File Edit Navigate Search Project Run Window Help
Java EE
*partitions.xam
27
48
8
282
60
78
17
81
4237
276
713
6
2
Markers Properties Servers Data Source Explorer Snippets Architectural Relations

**Figure 5:** Greying-out, as provided by JITTAC

The participants' 'increasing scope' approach suggests that there ***could be multiple simultaneous models at the same level of granularity, each containing a sub-set of the partitions and their relationships.*** Here navigation between these models could be based on the architect's selection of a particular partition and a drop down selection of the other models within which that partition appears. This implies a possible pre-session analysis where sets of partitions are composed based on cluster analysis [63], clone analysis [64] or connectivity analysis (for example, sub-sets could be formed, based on articulation vertices [65] when the entire partition set is modelled as a dependency graph). The current JITTAC implementation allows for simultaneous models of the system but for no such sub-set analysis.

### *5.2 What facilities of RT-RM are of interest to architects during architecture reconciliation? (RQ2)*

#### *5.2.1 Interactive Analysis of the Abstract Model*

In all of the participants' sessions they manipulated the abstract architectural model to obtain real-time feedback about the system. From the think-aloud data it became apparent that there were two rationales underpinning this behaviour and these are listed in Table 3. The first was that the participants used the model to focus in on particular violations at more detailed levels of granularity. To illustrate, consider this segment of Participant 1's think-aloud when considering a violating edge between two nodes that modelled two packages:

*"That's probably class <X>. I'm nearly sure. Let me just... (grabs class <X> from the package explorer and drops it on the canvas making a new node in the architectural model)... aah yes…, that accounts for well over half of them (the violations)"*

This type of analysis was apparent in four of the five sessions and happened seven times in total over these four sessions. It was always based on a strong hypothesis as to the source code element causing the violations, and these hypotheses were right in six of the seven episodes. Consequently, it can be considered a more optimal alternative than scanning through the “Architectural Relations" listing provided by the tool to identify prevalent offenders, where a prevalent offender can be defined as a source code entity that is frequently a start-point or end-point of the dependencies underpinning a violating edge.

This behaviour hints at an incongruity between the participants' needs and the “Architecture Relations” view, suggesting that ***(O6) approaches like these should provide architects with ranked lists of source code sources and targets underpinning each violating dependency in the Reflexion model, at different levels of granularity, where this ranking reflects their prevalence of occurrence.***

The second rationale for iterative analysis was to evaluate potential code changes before making them. To illustrate this, consider participant 3 when he found an interface in an implementation module:

*"Interface in wrong package... OK, let’s look at what would change if we moved it (drags a class from the package explorer to an existing interface vertex)... its fixed it..”.” ...Oh really cool.... that's really cool"*

Later, at the end of the session, when prompted about his feelings about the approach the participant said:

*"modelling the changes before you commit to them, that's really clever"*

A similar quote was given by participant 2 who also trialled a proposed change at the modelling level. In that case, he backed out of the change because the modelling exercise showed him that the proposed change would introduce a circular dependency:

*"aha.. a circular dependency: that’s why (the class was placed in what he thought was an inappropriate location)"*

Overall, three participants demonstrated this behaviour, a total of six times during their sessions, all actively commenting on its utility during the course of their session. This suggests that one of the core benefits of the real-time approach is the ability to model code changes in advance of actual change. Both this and the rationale of focusing in on violations argue strongly for ***(O7) the ability to quickly and easily alter the model and mappings at various levels of granularity and to provide real-time feedback on the effects of these alterations as part of any RM type approach***. This is an advantage/usage we did not originally consider when proposing RT-RM.

| Case Study | Rationale | |
|---|---|---|
| | Focus on Violation | Evaluating Code change |
| 1 | 3 | 1 |
| 2 | 2 | 2 |
| 3 | 1 | 2 |
| 4a | 1 | 0 |
| 4b | 0 | 1 |

**Table 3:** Rationale for Iterative Analysis of the Reflexion Model

### *5.2.2 De-Novo Mapping*

The tooling in this study allowed participants to create a mapping between the source code and their model in two ways:

1. They could build their models in advance and only then map the source code elements to those models to see if the edges were as they hypothesized. We called this approach *Retrospective Mapping*.

2. They could drag elements from the package explorer and drop them on the canvas directly to make Reflexion model vertices and see the resultant edges appear. In this instance they did not explicitly commit themselves to expected edges, and we called this behaviour *De-Novo Mapping*.

The researchers were not prescriptive in the approach to be adopted by participants. Participant 4a and 4b used the Retrospective approach. Only on one occasion did 4a 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"*. All the other participants used a mixed approach but predominantly relied on the De-Novo approach as their sessions progressed. For example, Participant 1 preferred to drag and drop the elements of the source code into the model as components directly:

*"(After dragging a package onto the canvas) This works reasonably well to the way... it just makes it convenient and it makes things quicker".*

While these three participants did use the Retrospective approach, their adoption of it was short-lived and seemed to be based on our presenting both options to them in the initial demonstration. This is best illustrated by a quotation from Participant 2:

*"(Tries the Retrospective approach) OK, why would I do that?...".*

This is interesting because Retrospective Mapping aligns with the classical RM approach. But three of the five participants preferred to use De-Novo mapping. This suggests ***that (O8) an advantage of RT-RM is that, often participants prefer to build their models directly from the source code, with real-time feedback, without explicitly stating a model in advance.*** This type of behaviour was also seen when participants wanted to focus in on the cause of violations as discussed in section 5.2.1.

### *5.2.3 Drag-and-Drop versus Lexical Mapping*

Participant 4a and 4b came from the same company and their systems adhered to a company-wide standard (layered) architecture. Regardless, the mappings required by these participants differed hugely. This can be seen by the fact that Participant 4b used eight mappings during his session and Participant 4a used 56. In Participant 4b's system, the packages in the package explorer directly related to each tier (some utility nodes were the exception). In Participant 4a's system, each package contained elements from each tier, leading to a much greater mapping effort for the participant when using the tool's drag-and-drop facility. The result was that Participant 4a typically had 1-1 mappings between source code elements (packages) and components, while Participant 4b had, on average, 1:5.6 mappings between them.

In the interview after his session Participant 4a expressed the desire for a lexical mapping facility in the tool, as per the original Reflexion Modelling tool developed by Murphy et al.: the jRMTool [61]. This was because his team's conventions demanded that the individual elements in each package had lexical signals as to their tier placement but that they were not placed in packages based on that layering. Interestingly, this issue also arose, to a degree for Participant 1 and Participant 2. In both systems several of the interfaces were placed within one 'interface-collection' package. While the participants were happy to drag these interfaces out individually, it would have been helpful if interface nodes could have been formed, based on the lexical matching evident between the implementation package and their associated interface within this 'interface-collection' package. However the vast majority of their mappings were suited to drag-and-drop operations, as were all of Participant 3's. These findings suggest that ***(O9) architects should be offered a drag-and-drop mapping facility and a lexical mapping facility.***

This raises interesting questions about the potential for confusion if both facilities were to be 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 architects forget which of the two mapping facilities they used for certain parts of the source, or which one has precedence in the case of overlaps?

Indeed, even with just the drag-and-drop mapping facility available, Participants 1, 2 and 4b mentioned difficulties in keeping track of the unmapped parts of the codebase. While the code mapped to each vertex was made explicit in the outline view, no perspective illustrating the unmapped source code was available in JITTAC and this proved to be a problem. This difficulty would likely be exacerbated by the availability of more than one mapping-based strategy and so should be addressed. Hence ***(O10) architects should be made aware of the source code that remains unmapped for each architectural model they create.***

With respect to the precedence of drag-and-drop or lexical mapping, participant 4a mentioned that *"(while) lexical analysis would come in handy... many files would not fall under a lexical definition (convention)"* This is entirely plausible: situations where conventions demand more than one naming approach, or simple programmer inexperience may lead to lower naming consistency in a software system that remains unnoticed for long periods of time. In addition, drag-and-drop was appropriate for the vast majority of mappings in three of the five case studies. This suggests that ***if both modes of mapping were to be made available at the same time, drag-and-drop should have default primacy, but the architect should be able to change this default.***

### *5.3 Does real-time architectural violation identification prompt the removal of violations in commercial practice, during architectural reconciliation and, if so, where is the locale of change? (RQ3)*

Table 4 presents the results of an analysis of the architectural violations identified by the participants in the study over several iterations of the RM process. As described earlier we consider an iteration as a distinct phase in the sessions where participants define or augment their as-intended architectural model of the system, make mappings between that model and the source code, and possibly, subsequently analyse or manipulate that model to explore and assess any violations. So for example, participant 2 created seven models, each an expansion of the previous model in that additional parts of the system were included each time. Each time the expanded model required additional mappings to the source code. For four of the seven models he assessed and explored the violations he had identified. The three other expansions highlighted no new violations.

| Case | Violations Identified | | | | | | |
|---|---|---|---|---|---|---|---|
| | Iter. 1 | Iter. 2 | Iter. 3 | Iter. 4 | Iter. 5 | Iter. 6 | Iter. 7 |
| 1 | 1(10) | 0 | 1(33) | 2 (26)[1] | | | |
| 2 | 1 (10) | 0 | 1 (3) | 0 | 1 (10) | 0 | 3 (1156) |
| 3 | 0 | 1(26) | 1 (86) | | | | |
| 4a | 0 | 1 (2) | 3 (31) | | | | |
| 4b | 1 (43) | 1 (90) | 1 (16) | 1 (5) | | | |

**Table 4:** Violations Identified per Iteration

The number in parenthesis in each cell in Table 4 represents the number of source code dependencies underpinning the violations identified. The number before the parentheses represents the number of violations, as measured by actual edges between vertices in the Reflexion model. For example the Reflexion model from iteration five of Participant 2's session is presented in Figure 2. In this model we see the 10 source code dependencies on one edge that were introduced in iteration five and the three source code dependencies that were introduced in iteration three on another edge. The 10 violations introduced in iteration one are not visible in the figure as they reflected a mistake in the mapping from source code to vertices in the Reflexion model and the participant changed the mapping before iteration two to remove them. Mapping changes like this are signalled by highlighting in Table 4.

It should be noted that, even with the large number of violations identified in Table 4, the level of violation removal was quite low. At the time of the post-session interview, several months after the empirical session, several violations had been addressed but no architect had removed more than 50% of the violations identified and two of the architects had removed none. In addition, the architects had no plans to address the remaining violations in the future.

### *5.3.1 Locale of Change*

Table 5 documents the changes that programmers made to the system to address the violations they identified. As shown in Table 4 and in Table 5, three of the participants changed mappings during their session. These reflected genuine mapping mistakes, where the name of classes and packages suggested to them that they

[1] As shown in Figure 4 this participant created a 'Rest-of-System' vertex, connecting it to his existing model, during this iteration. The reason that the number reported in this cell does not tally with the violations that can be seen in Figure 4 is that the participant subsequently inspected these dependencies (after the session) and, in our retrospective interview, stated that the violating edge incident on the 'Rest-of-System' vertex with 1151 dependencies was not, in fact, a violation.

belonged to Reflexion model vertices other than those that they actually belonged to. To illustrate, consider Participant 4a’s utterance:

*“Oh so… I haven’t got to mapping this into the correct location yet!”*

As shown in Table 5, in two instances (Participant 2 and 4) these redefined mappings served to address a ‘violation’, but in the other it had no effect.

| Case Study | Resultant Change Focus | | |
|---|---|---|---|
| | Code | Architectural Model | Mapping |
| 1 | | Add | |
| 2 | | Add, Trial | Yes |
| 3 | Yes | NoDen | |
| 4a | | | Yes |
| 4b | | | |

**Table 5:** Change Focus

Interestingly all bar two of the participants chose to change the architectural model, after discussing the issue with other members of their development teams. For example, Participant 1, after discussing the issue some time later with his team stated:

*"I talked to the team.... They said no way. We've got to bypass them (interfaces) or else we'd hammer it (the system's performance)"*

The architect agreed and changes were made to the architectural model of the system, rather than the system itself. This is an example of the case 'Add' where developer insight led the architect to change the as-intended architectural model to achieve additional architectural goals. This was also evident in case study 2, again for performance reasons.

There was also a related situation ('NoDen') where a participant agreed to change the model, not because the implemented architecture (as embodied by the source code dependencies) improved adherence to architectural goals but because it did not denigrate adherence to the currently stated architectural goals and because the violating dependencies were widespread in the code. Thus they would take some effort to remove. This is illustrated by a quote from Participant 3:

*"If we go down to separately versioned components (only) then the API becomes God... vitally important. (Otherwise its) difficult to get specific resources for this. It’s perceived in terms of finding time, which is a tough argument to make to management. It's funny (because) it adds to the maintenance overhead, so I'm very keen on (refactoring the code)"*

Finally, as discussed in section 5.2.1, there was another situation where the architect trialled the proposed changes at the RM level (‘Trial’) and it showed him a knock-on effect (a circular dependency) that he considered outweighed the potential benefits of aligning the code with the architecture. Hence, he reversed his change at the RM level, resulting in only a temporary change at the architectural model level.

So it seems that ***(O11) RT-RM, while prompting some removal of violations, is not sufficient to prompt removal of all, or the majority of, the violations identified when***

***applied retrospectively on a system that has drifted for some time. In situations where the architects do move to address architectural violations, they are more likely to change the architectural description than the source code.***

### *5.4 What are the factors that influence the removal of violations during RT-RM architectural reconciliation, in commercial software systems? (RQ4)*

The findings from this study indicate that there are several factors that influence the removal of violations and several factors which then influence the site at which the removal is performed. As discussed above, the site at which removals are performed can depend on architectural agendas (in this case 'performance') which are currently deemed more important than the architectural agendas espoused in the as-intended architecture, Alternatively, they can depend on the effort associated with changing the source code (when the violation does not denigrate the architecture).

In terms of whether a violation is removed in the first place, the participants cited several considerations. They cited the non-availability of resources to undertake the required re-structuring of the code-base, the lower priority associated with restructuring than enhancing the current systems' functionalities and the embedded-ness of the identified violations. This last rationale suggests that, if architectural reconciliation can be achieved in a JIT fashion, then the violations might be identified before they would become embedded, as it would warn programmers as they introduce the violations, not retrospectively.

With respect to the violations that were addressed, only one participant changed the source code. This participant (3) was an architect who had extensively coded on the project, identified a violation that *'particularly offends'* him and identified the change required: re-directing a localized set of calls to an existing interface. He stated that there were no gains to be made by by-passing the interface in this situation, and thus, that coupling should be decreased.

## 6 Discussion

Many of the observations reported above (O1, O5, O6, O9 and O10) highlight potential future directions for RM. For example, the architectural templates suggestion (O1), and the inclusion of drag-and-drop *and* lexical mapping (O9) serve to enrich the modelling and consistency goals of these approaches. While we are aware of no academic work on desirable strategies for mapping code entities to architectural vertices in the Reflexion model, we are aware of commercial tool-sets that allow both (for example Structure101 [46]).
The suggestion that there should be an interface template, echos the findings of Knodel and Popescu [33] who suggested that ports should be associated with architectural vertices in Reflexion models, constraining the allowable dependencies on those vertices to defined interfaces. More generally, Pruijt and Brinkkemper [58] argue for richer architectural modelling of systems, which could also include layering constructs.

O5, O6 and O10 suggest enriching the visualization interface presented to the architect. Under these suggestions, architects would be able to scale the approach to large architectural models, identify classes that cause a significant amount of violations, and identify code that was as yet unmapped. Koschke and Simon [51] have implemented hierarchical RM, as have a number of the commercial providers (for example [66]) towards the scalability issue. Unmapped code was highlighted by a view in the original RM tool (the jRMTool). But a review of the literature in the area suggests that the proposal to rank classes in order of significance, in terms of the violations they prompt, is novel.

With respect to O7: real-time feedback when (easily) manipulating the model and mappings, Buckley et al. performed two in-vivo case studies with a RM approach on a commercial Learning Management System (LMS) in IBM (Buckley et al. 2008). The second of these case studies implied the need for such a facility. There, the participant used the technique to remove the GUI layer of the LMS, so that the rest of the system could be integrated into another IBM system (Workspace Collaborative Learning). As in the studies here, the participant worked exclusively at the modelling level before proceeding to make the necessary code changes (in that case without real-time feedback). He cited the detail of code change, and subsequent compiler errors, as the reason for approaching the task at the RM level.

This behaviour can be contextualized in terms of the cognitive dimensions proposed by Blackwell et al [67]. Specifically, code is a ‘viscous’ representation to change due to its inherent level of detail and because programmers often face situations where code change prompts ripple effects they have not foreseen. Working at the abstract level provided by RM provides a less viscous representation that facilitates the cognitive dimension of ‘provisionality’: the ability to check out certain implications of change, before actually performing or reversing that change.

Of the more surprising requests from participants (O2, O3) one was for feature-oriented RM where individual features would form the individual vertices of a Reflexion model and this model would act as a dependency analysis vehicle for architects. Such an environment would be in the spirit of FEAT [68] where the level of visualization would be increased from Feature contents, as in FEAT, to feature contents and inter-feature dependencies. The participants’ comments hinted at the requirement for a Feature Location Technique as part of this environment and an iterative process for this is envisaged, in line with the process proposed by Kastner et al. [69]. In this process users would seed the tool with elements of the source code they associate with the feature and the tool would iteratively apply *Feature Location Techniques* to that seed set. The results would then be fed back to the user via a Reflexion model, as suggestions that can be accepted or rejected by the architect, for further applications of Feature Location techniques.

O4 suggests that the inability of traditional RM to go beyond source code dependencies is an issue for the architects. This has been noted in general in the literature [70] and has been the subject of some preliminary investigation [71].

One of the core advantages of RM proposed in the literature was the architects pre-commitment to an expected architectural model [13-14]: When the system was parsed and shown to differ from that architectural expectation the architect’s subsequent surprise created an element of “cognitive conflict” between expectation and reality, driving further investigation [14].

Instead, the behaviour observed here (O8) suggests that architects would prefer to create their model directly from the source code, retrospectively analysing the relationships that appeared rather than pre-committing to a set of expected relationships. Regardless, in most of the cases observed architects did seem to bring expectations to their iterations: specifically that communication should happen through interfaces, and between adjacent layers only. They also brought expectations as to the location of the dependencies underpinning the identified violations. So, it seems more likely that the architects just found the explicit statement of these expectations/their model un-necessary and dragging-and-dropping from the package explorer less clunky. This is evidenced by quotations like those presented in section 5.2.2.

In terms of O11, which refers to the likelihood of identified violations being removed, the study reflects other studies in the area, in that the findings were mixed. While Knodel et al. [52] and Kolb et al. [19] report that violations were addressed in their case study, Rosik et al [20] found that they were not. Other studies have reported that only academic researchers removed identified violations [15, 21]. In this study, some were removed but the majority remained. It seems that factors such as programmer availability, the embedded-ness of the violations and organizational priorities make the removal of violations company-specific. These findings suggest that future empirical studies in this area might usefully consider JIT violation notification, varied commercial contexts and be longitudinal in nature, to assess the subsequent outcomes of violation identification.

Also in relation to O11, when violations were removed, the majority of them were done at the model level, not at the code. Similarly, Buckley et al. [14] reported that when the as-implemented architecture of the system under study was shown to the system architect, he noted that the as-implemented architecture was inconsistent with how he viewed the system, but that it was correct and valid, in that it did not compromise any of the system's architectural goals. Given the extent to which the as-implemented architecture was embedded in the system, he adopted a 'NoDen' type behaviour and stated that he was going to change his own (internal) architectural model of the system. A similar behaviour was observed in a longitudinal architectural conformance study in IBM [20]. There, the architectural model was changed to accommodate a legacy component and to obfuscate several 'trivial' violations. However, in that instance, the hidden violations grew over time, resulting in bigger violations. These complimentary findings suggest that architects are more likely to approach consistency through modification of the as-intended model rather than the code-base, largely for pragmatic reasons, but also for the reason that insights from developers working on a system are seen as a legitimate input to the definition of a system's architecture [1, 35].

The findings with respect to the architectural templates desired (O1) and the locale of change subsequent to violation identification (O11), provide an interesting commentary on the architectural agendas at play during these software systems' development. Specifically, the systems seem to have been built with modularity and maintainability as a primary architectural concern. Defining APIs to packages and programming to those APIs (as per the as-intended architectures of four of these systems), divides the systems into more manageable, information-hiding [72] partitions. The APIs expose only limited details of these partitions to the rest of the system, a well-accepted approach to reducing complexity and easing maintainability [73]. Likewise programming to these APIs and programming to a layered architectural style (as in 2 of the case studies) facilitates replace-ability [74] as can be prompted, for example, by the evolution of new services, or new database and presentation layer technologies.

System performance is another apparent architectural requirement for some of the systems studied. Specifically, in two cases the development team highlighted their system-performance concerns to the architect, objecting to the programming-to-interfaces style the architect proposed. This illustrates a distributed cognition model of software across the team [75], highlighting the conflicting agendas of performance and maintainability in these instances. Tellingly, in both cases, the architect ultimately deferred to the development team and accepted the violations as necessary, suggesting that the performance requirement out-ranked the maintainability requirement at that point in time. This also suggests that the relative importance of performance over maintainability, as a non-functional requirement, may grow over time in line with customer concerns, as systems become larger and processing becomes more involved.

Finally, given the financial nature of all these systems, it is interesting to note that performance and modularity were the only two architectural requirements that arose during the case studies. It could be expected that issues such as security and traceability would also be major concerns in the development of such systems. The most likely reason for their non-emergence during this analysis is scoping: the technique applied only focused on the (java) source code dependencies in the systems whereas these other non-functional requirements were handled within the systems' frameworks.

Whatever the rationale underpinning the locale of change, these findings imply that the labels 'Architecture Conformance' [42, 52] or 'Compliance Checking' [9, 40] when applied to this field, are misnomers. They convey the impression that the code should conform to the as-intended architecture. Better labels would be 'Architecture Consistency' [77] or 'Architecture Reconciliation' [35], where the implication is that consistency between the architecture and the source code is the goal, regardless of the appropriate site-of-change.

## 7 Threats to Validity and Reliability

Validity refers to the extent to which empirical results are meaningful [77]. A related concern is reliability, which refers to consistency in data gathering and data analysis [77]. This section assesses the validity and reliability issues that can arise in (multi-)case studies [78] and the steps that were taken to mitigate against these concerns in this instance. With respect to validity, it discusses these issues under the validity-categorization schema of Shuttleworth [79]:

*Construct validity* is the degree to which the measurements taken relate to the phenomena under-study. Here the phenomena were the participants' usage of RT-RM (the perspectives and facilities they used/desired) and their removal of architectural violations (in terms of the extent of removal, locale of removal and rationale for removal/persistence). The measurements were the videoed protocol of the participants, their think-aloud data and their retrospective interviews.

The observation-type measurements we employed could report directly on many of the phenomena under study (for example the ‘architectural-style perspectives’ and the ‘interactive analysis’ usage). However, other phenomena could not be assessed so directly (for example ‘factors that influenced violation removal’, and the desire for a ‘Feature Oriented’ perspective). Analysis of think-aloud data and retrospective interviews were employed to report on these phenomena, and such analysis is open to individual interpretation and subjectivity, thus threatening the construct validity of the study. The researchers counteracted this possibility through discussion meetings where each researcher who analysed the data independently presented their results to another researcher who adopted and argued a counter position (colloquially known as a “devil’s advocate” role [79]). In addition, we employed a participatory verification step, where the five participants reviewed our findings on their sessions for misinterpretations and inconsistencies.

*Internal validity* is the extent to which independent variables (and only independent variables) affect the dependent variables in controlled experiments. Even though in-vivo case studies cannot be equated to controlled studies, the presence of RT-RM, as embodied in JITTAC, could be considered an independent variable and the participants’ behaviour and desired enhancements could be considered the dependent variables in these studies.
The presence of the research team at the sessions, and in performing the interviews afterwards, might be considered “another variable” that impacted on the dependent variable [80]. For example, it might have prompted participants to remove more violations and to suggest that the RT-RM approach, as embedded in JITTAC, was more congruent to their task than it was. Given the low rate of violation removal found in the studies, it is unlikely that the team’s presence prompted participants to remove more violations than they would have done otherwise. But if it did, the valid removal rate would be even lower than reported on here, reinforcing our findings.

Many of the “congruent-to-task” improvements arose naturally in the think-aloud protocol, but there is the possibility that more would have been identified if participants had been reporting to someone else. We attempted to mitigate this possibility by receiving these improvement-comments positively, with phrases like: ‘Oh that’s very interesting’.

*External validity* is the degree to which the conclusions of the study are applicable to software development in general. Our study was performed on five commercial systems from four different companies where four architects and one experienced developer (with some architectural responsibility for the system) performed in-vivo architecture reconciliation tasks. The systems had already been deployed and were of different sizes and ages. Three of them were extremely large systems and the others were of a 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, all the participants used JITTAC and JITTAC embodies one specific variant of RT-RM only. Additionally, as with all in-vivo case studies of this depth, the number of data-points is limited and this lessens the external validity of the study in general. We would hope that other researchers would add to the evidence in this area by performing additional in-vivo case studies with other RT-RM approaches and tools.

As stated above, *reliability* is concerned with the consistency of result gathering/analysis [77]. To improve reliability in data gathering, congruent data was collected from multiple sources. These included the video recordings of the sessions, participant interviews, participant-uttered observation and the screenshots collected. With respect to data analysis, two researchers independently analysed each session and had analysis-discussion meetings afterwards. As described above, in these meetings they each acted as devil's advocate to the findings of the other researcher when their analyses did not align, serving to make the analysis more transparent (explicit), reliable and reasoned across the research team. In addition, each participant in the study reviewed the findings associated with their session for accuracy.

Towards reliability going forwards, we documented in detail all the procedures (design and protocol) in each session so that the case study could be repeated in the future by other researchers.

## 8 Conclusions

This study has presented a number of observations from a multi-case study of RT-RM, identifying several utilities of the approach and several limitations. Most surprisingly it showed that (at least) four months after their sessions all the associated development teams had removed less than 50% of the architectural violations identified in their RM session and some had removed none. The primary reasons given were the effort involved, the organizations' focus on new business goals and the lack of perceived importance of the violations identified in comparison to these new business goals.

With regard to the architecture reconciliations that were achieved, the majority of practitioners using RM in these case studies favoured changes to the intended architecture rather than changes to the code. This reflected the teams' prioritization of performance, as embodied in the current implementation, over the longer-term maintainability gains implied by the intended architecture. In another case it reflected the (large) effort required to address violations that did not denigrate the architecture, in the opinion of the architect.

Participants would generally like to have richer modelling facilities in such approaches. Specifically, their behaviours implied a need for certain architectural templates and the need for interface vertices. They would like modelling facilities to cope with larger scale models, and also expressed a need to expand the technique further, to model the event driven/dynamic behaviour of systems and to model systems in terms of their user functionalities.

There were several usability issues that arose. On the positive side, the approach provided for real-time, iterative analysis of the system at the architectural model level, which allowed participants interactively assess changes to the code-base before they proceeded and allowed them check hypotheses on the specific location of architectural inconsistencies. In addition, participants predominantly chose to create elements of their architectural model directly from elements in their code base which, though inconsistent with the classical RM approach, was preferred by participants as less clunky. Such an approach seems intuitively to have less impact in terms of cognitive conflict, but the participants still seemed to have strong implicit expectations when they adopted this behaviour and so it is likely that cognitive conflict still played a motivating/focusing role.

The participants' behaviour suggested the need for both Drag-and-Drop and lexicon-based mapping from the source code to the architectural elements in the as-intended model. The predominance of Drag-and-Drop in these session, allied with the perceived more approximate nature of lexicon-based mapping suggests that Drag-and-Drop should be the default option, even if both were made available. However, participants expressed difficulty in remembering the mapping relationships that they had and hadn't created, a difficulty that could be exacerbated by a second mapping facility.

**Acknowledgement**

This work was supported, in part, by Science Foundation Ireland grants 12/IP/1351 and 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie) and by the University of Brighton under the Rising Star Scheme awarded to Nour Ali.

## References

1. Garlan, David. 2000. 'Software Architecture: A Roadmap'. In *Proceedings of the Conference on The Future of Software Engineering*, 91–101. Limerick, Ireland: ACM. doi:10.1145/336512.336537.
2. Perry, Dewayne E, and Alexander L Wolf. 1992. 'Foundations for the Study of Software Architecture'. *ACM SIGSOFT Software Engineering Notes* 17 (October): 40–52. doi:10.1145/141874.141884.
3. Knodel, Jens. 2011. 'Sustainable Structures in Software Implementations by Live Compliance Checking'. Stuttgart, Fraunhofer Verlag.
4. Schwanke, Robert W. 1991. 'An Intelligent Tool for Re-Engineering Software Modularity'. In *, 13th International Conference on Software Engineering, 1991. Proceedings*, 83–92. doi:10.1109/ICSE.1991.130626.
5. Cimitile, Aniello., and Guiseppe Visaggio. 1995. 'Software Salvaging and the Call Dominance Tree'. *J. Syst. Softw.* 28 (2): 117–27. doi:10.1016/0164-1212(94)00049-S.
6. Koschke, Rainer 2002. 'Atomic Architectural Component Recovery for Program Understanding and Evolution'. In *International Conference on Software Maintenance, 2002. Proceedings*, 478–81. doi:10.1109/ICSM.2002.1167807.
7. Babbage, Charles. 1864. *Passages From The Life Of A Philosopher*. Cooper Press.
8. de Silva, Lakshitha, and Dharini Balasubramaniam. 2012. 'Controlling Software Architecture Erosion: A Survey'. *Journal of Systems and Software* 85 (1): 132–51. doi:10.1016/j.jss.2011.07.036.

9. Hochstein, L., and M. Lindvall. 2003. ‘Diagnosing Architectural Degeneration’. In *Software Engineering Workshop, 2003. Proceedings. 28th Annual NASA Goddard*, 137–42. doi:10.1109/SEW.2003.1270736.

10. de Moor, Oege, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. 2008. ‘.QL: Object-Oriented Queries Made Easy’. In *Generative and Transformational Techniques in Software Engineering II*, edited by Ralf Lämmel, Joost Visser, and João Saraiva, 78–133. Lecture Notes in Computer Science 5235. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/978-3-540-88643-3_3.

11. Mattsson, Anders. 2012. ‘Modelling and Automatic Enforcement of Architectural Design Rules’. University of Limerick. http://ulir.ul.ie/handle/10344/2478.

12. Murphy, Gail C., and David Notkin. 1997. ‘Reengineering with Reflexion Models: A Case Study’. *Computer* 30 (8): 29–36. doi:10.1109/2.607045.

13. Le Gear, Andrew, Jim Buckley, Brendan Cleary, J.J. Collins, and Kieran O’Dea. 2005. ‘Achieving a Reuse Perspective within a Component Recovery Process: An Industrial Scale Case Study’. In *13th International Workshop on Program Comprehension, 2005. IWPC 2005. Proceedings*, 279–88. doi:10.1109/WPC.2005.4.

14. Buckley, Jim, Andrew P. LeGear, Chris Exton, Ross Cadogan, Trevor Johnston, Bill Looby, and Rainer Koschke. 2008. ‘Encapsulating Targeted Component Abstractions Using Software Reflexion Modelling’. *Journal of Software Maintenance and Evolution: Research and Practice* 20 (2): 107–34. doi:10.1002/smr.364.

15. Murphy, Gail C., David Notkin, and Kevin Sullivan. 2001. ‘Software Reflexion Models: Bridging the Gap between Design and Implementation’. *Software Engineering, IEEE Transactions on* 27 (4): 364–80. doi:10.1109/32.917525.

16. Tvedt Tesoriero, Roseanne, Patricia Costa, and Mikael Lindvall. 2004. ‘Evaluating Software Architectures.’ In *Advances in Computers*, 61:1–43. http://dblp.uni-trier.de/db/journals/ac/ac61.html#TvedtCL04.

17. Knodel, Jens, Mikael Lindvall, Dirk Muthig, and Matthias Naab. 2006. ‘Static Evaluation of Software Architectures’. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, 10 pp. – 294. doi:10.1109/CSMR.2006.53.

18. Ali, Nour, Jacek Rosik, and Jim Buckley. 2012. ‘Characterizing Real-Time Reflexion-Based Architecture Recovery: An In-Vivo Multi-Case Study’. In *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*, 23–32. QoSA ’12. New York, NY, USA: ACM. doi:10.1145/2304696.2304702.

19. Kolb, Ronny, Isabel John, Jens Knodel, Dirk Muthig, Uwe Haury, and Gerald Meier. 2006. ‘Experiences with Product Line Development of Embedded Systems at Testo AG’. In *Software Product Line Conference, 2006 10th International*, 10 pp. – 181. doi:10.1109/SPLINE.2006.1691589.

20. Rosik, Jacek, Andrew Le Gear, Jim Buckley, Muhammad Ali Babar, and Dave Connolly. 2011. ‘Assessing Architectural Drift in Commercial Software Development: A Case Study’. *Softw. Pract. Exper.* 41 (1): 63–86. doi:10.1002/spe.999.

21. Tran, John B., Michael W. Godfrey, Eric H.S. Lee, and Richard C. Holt. 2000. 'Architectural Repair of Open Source Software'. In *8th International Workshop on Program Comprehension, 2000. Proceedings. IWPC 2000*, 48–59. doi:10.1109/WPC.2000.852479.
22. Lindvall, Mikael, Roseanne Tesoriero, and Patricia Costa. 2002. 'Avoiding Architectural Degeneration: An Evaluation Process for Software Architecture'. In *Eighth IEEE Symposium on Software Metrics, 2002. Proceedings*, 77–86. doi:10.1109/METRIC.2002.1011327.
23. Mattsson, Anders, Björn Lundell, Brian Lings, and Brian Fitzgerald. 2009. 'Linking Model-Driven Development and Software Architecture: A Case Study'. *IEEE Transactions on Software Engineering* 35 (1): 83–93. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2008.87.
24. Van Gurp, Jilles, and Jan Bosch. 2002. 'Design Erosion: Problems and Causes'. *J. Syst. Softw.* 61 (2): 105–19. doi:10.1016/S0164-1212(01)00152-2.
25. Van Gurp, Jilles, Sjaak Brinkkemper, and Jan Bosch. 2005. 'Design Preservation over Subsequent Releases of a Software Product: A Case Study of Baan ERP: Practice Articles'. *J. Softw. Maint. Evol.* 17 (4): 277–306. doi:10.1002/smr.v17:4.
26. Tvedt Tesoriero, Roseanne, Patricia Costa, and Mikael Lindvall. 2002. 'Does the Code Match the Design? A Process for Architecture Evaluation'. In *International Conference on Software Maintenance, 2002. Proceedings*, 393–401. doi:10.1109/ICSM.2002.1167796.
27. Shaw, Mary, and Paul Clements. 2006. 'The Golden Age of Software Architecture'. *IEEE Softw.* 23 (2): 31–39. doi:10.1109/MS.2006.58.
28. Czarnecki, Krysztof, and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. 1 edition. Boston: Addison-Wesley Professional.
29. Henriksson, Anders, and Henrik Larsson. 2003. *A Definition of Round-Trip Engineering*. Linköpings University, Sweden. http://www.ida.liu.se/henla/papers/roundtrip-engineering.pdf.
30. Stahl, Thomas, Markus Voelter, and Krzysztof Czarnecki. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.
31. Mattsson, Anders, Brian Fitzgerald, Björn Lundell, and Brian Lings. 2012. 'An Approach for Modeling Architectural Design Rules in UML and Its Application to Embedded Software'. *ACM Trans. Softw. Eng. Methodol.* 21 (2): 10:1–10:29. doi:10.1145/2089116.2089120.
32. Herold, Sebastian, and Andreas Rausch. 2013. 'Complementing Model-Driven Development for the Detection of Software Architecture Erosion'. In *2013 5th International Workshop on Modeling in Software Engineering (MiSE)*, 24–30. doi:10.1109/MiSE.2013.6595292.
33. Knodel, Jens, and Daniel Popescu. 2007. 'A Comparison of Static Architecture Compliance Checking Approaches'. In *Software Architecture, 2007. WICSA '07. The Working IEEE/IFIP Conference on*, 12. doi:10.1109/WICSA.2007.1.
34. Buckley, Jim, Sean Mooney, Jacek Rosik, and Nour Ali. 2013. 'JITTAC: A Just-in-Time Tool for Architectural Consistency'. In *Proceedings of the 2013 International Conference on Software Engineering*, 1291–94. ICSE '13. Piscataway, NJ, USA: IEEE Press. http://dl.acm.org/citation.cfm?id=2486788.2486987.

35. de Silva, Lakshitha, and Dharini Balasubramaniam. 2013. 'PANDArch: A Pluggable Automated Non-Intrusive Dynamic Architecture Conformance Checker'. In *Software Architecture*, edited by Khalil Drira, 240–48. Lecture Notes in Computer Science 7957. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/978-3-642-39031-9_21.
36. Ganesan, Dharmalingam., Thorsten. Keuler, and Yutaro Nishimura. 2008. 'Architecture Compliance Checking at Runtime: An Industry Experience Report'. In *The Eighth International Conference on Quality Software, 2008. QSIC '08,* 347–56. doi:10.1109/QSIC.2008.45.
37. Popescu, Daniel, and Nenad Medvidovic. 2008. 'Ensuring Architectural Conformance in Message-Based Systems'. In *Workshop on Architecting Dependable Systems (WADS).*
38. Sefika, Mohlalefi, Aamod Sane, and Roy H. Campbell. 1996. 'Monitoring Compliance of a Software System with Its High-Level Design Models'. In , *Proceedings of the 18th International Conference on Software Engineering, 1996,* 387–96. doi:10.1109/ICSE.1996.493433.
39. Klocwork. 2014. *See your architecture, optimise your code* [online] available: http://www.klocwork.com/products/insight/architect-code-visualization/ [accessed 23/01/2015].
40. Herold, Sebastian. 2011. 'Architectural Compliance in Component-Based Systems. Foundations, Specification, and Checking of Architectural Rules'. Ph.D. thesis, Clausthal University of Technology.
41. de Moor, Oege, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. "Keynote Address:. QL for Source Code Analysis." In *SCAM*, vol. 7, pp. 3-16. 2007.
42. Passos, Leonarod, Ricardo Terra, Marco Tulio Valente, Renato Diniz, and Nabor Mendonçanda. 2010. 'Static Architecture-Conformance Checking: An Illustrative Overview'. *IEEE Software* 27 (5): 82–89. doi:10.1109/MS.2009.117.
43. Duszynski, Slawomir., Jens Knodel, and Mikael Lindvall. 2009. 'SAVE: Software Architecture Visualization and Evaluation'. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on,* 323–24. doi:10.1109/CSMR.2009.52.
44. Sangal, Neeraj, Ev Jordan, Vineet Sinha, and Daniel Jackson. 2005. 'Using Dependency Models to Manage Complex Software Architecture'. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 167–76. OOPSLA '05. New York, NY, USA: ACM. doi:http://doi.acm.org/10.1145/1094811.1094824.
45. Bischofberger, Walter, Jan Kühl, and Silvio Löffler. 2004. 'Sotograph — A Pragmatic Approach to Source Code Architecture Conformance Checking'. In *Software Architecture*, edited by Flavio Oquendo, Brian Warboys, and Ron Morrison, 3047:1–9. Lecture Notes in Computer Science. Springer Berlin / Heidelberg. http://dx.doi.org/10.1007/978-3-540-24769-2_1.
46. Headway Software. 2014. *Software architecture and dependency management tool for java, c, c++, .net and more* [online] available: http://www.headwaysoftware.com/products/?code=Structure101 [accessed 23/01/2015]
47. Rosik, Jacek and Jim Buckley. 2009. 'Design Requirements for an Architecture Consistency Tool'. In *Proceedings of the 21st Annual Psychology*

*of Programming Interest Group Conference*. Limerick, Ireland. http://www.thehealthwell.info/search-results/design-requirements-architecture-consistency-tool?&content=resource&member=none&catalogue=none&collection=none&tokens_complete=true.

48. Knodel, Jens, Dirk Muthig, and Dominik Rost. 2008. ‘Constructive Architecture Compliance Checking - an Experiment on Support by Live Feedback’. In *IEEE International Conference on Software Maintenance, 2008. ICSM 2008*, 287–96. doi:10.1109/ICSM.2008.4658077.
49. Eichberg, Michael, Sven Kloppenburg, Karl Klose, and Mira Mezini. 2008. ‘Defining and Continuous Checking of Structural Program Dependencies’. In *Proceedings of the 30th International Conference on Software Engineering*, 391–400. ICSE ’08. New York, NY, USA: ACM. doi:10.1145/1368088.1368142.
50. Layman, Lucas, Laurie Williams, and Robert St. Amant. 2007. ‘Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools’. In *First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007*, 176–85. doi:10.1109/ESEM.2007.11.
51. Koschke, Rainer, and Daniel Simon. 2003. ‘Hierarchical Reflexion Models’. In *Proceedings of the 10th Working Conference on Reverse Engineering*, 36 – . WCRE ’03. Washington, DC, USA: IEEE Computer Society. http://dl.acm.org/citation.cfm?id=950792.951359
52. Knodel, Jens, Dirk Muthig, Uwe Haury, and Gerald Meier. 2008. ‘Architecture Compliance Checking - Experiences from Successful Technology Transfer to Industry’. In *12th European Conference on Software Maintenance and Reengineering, 2008. CSMR 2008*, 43–52. doi:10.1109/CSMR.2008.4493299.
53. Pohl, Klaus, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer New York, Inc.
54. Kitchenham, Barbara, Lesley Pickard, and Shari Lawrence Pfleeger. 1995. ‘Case Studies for Method and Tool Evaluation’. *IEEE Software* 12 (4): 52–62. doi:10.1109/52.391832.
55. Runeson, Per, and Martin Höst. 2009. ‘Guidelines for Conducting and Reporting Case Study Research in Software Engineering’. *Empirical Software Engineering* 14 (2): 131–64. doi:10.1007/s10664-008-9102-8.
56. Ericsson, K. Anders, and Herbert A. Simon. 1993. *Protocol Analysis - Rev’d Edition: Verbal Reports as Data*. Revised edition. Cambridge, Mass: A Bradford Book.
57. Corbin, Juliet M., and Anselm Strauss. 1990. ‘Grounded Theory Research: Procedures, Canons, and Evaluative Criteria’. *Qualitative Sociology* 13 (1): 3–21. doi:10.1007/BF00988593.
58. Pruijt, Leo, and Sjaak Brinkkemper. 2014. ‘A Metamodel for the Support of Semantically Rich Modular Architectures in the Context of Static Architecture Compliance Checking’. In *Proceedings of the WICSA 2014 Companion Volume*, 8:1–8:8. WICSA ’14 Companion. New York, NY, USA: ACM. doi:10.1145/2578128.2578233.

59. Wilde, Norman, and Michael C. Scully. 1995. ‘Software Reconnaissance: Mapping Program Features to Code’. *Journal of Software Maintenance* 7 (1): 49–62. doi:10.1002/smr.4360070105.
60. Wilde, Norman, Michelle Buckellew, Henry Page, Vaclav Rajlich, and LaTreva Pounds. 2003. ‘A Comparison of Methods for Locating Features in Legacy Software’. *J. Syst. Softw.* 65 (2): 105–14. doi:10.1016/S0164-1212(02)00052-3.
61. *Reflexion Model Eclipse Plugin.* 2013. SourceForge.net. [online], available: http://sourceforge.net/projects/jrmtool/ [accessed 23/01/2015].
62. Dit, Bogdan, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. ‘Feature Location in Source Code: A Taxonomy and Survey’. Journal of Software: Evolution and Process 25 (1): 53–95. doi:10.1002/smr.567.
63. Christl, Andreas, Rainer. Koschke, and Margaret-Anne Storey. 2005. ‘Equipping the Reflexion Method with Automated Clustering’. In *12th Working Conference on Reverse Engineering*, 89–98. doi:10.1109/WCRE.2005.17.
64. Frenzel, Pierre., Rainer Koschke, Andreas P.J. Breu, and Karsten Angstmann. 2007. ‘Extending the Reflexion Method for Consolidating Software Variants into Product Lines’. In *14th Working Conference on Reverse Engineering, 2007. WCRE 2007*, 160–69. doi:10.1109/WCRE.2007.28.
65. Gibbons, Alan. 1985. *Algorithmic Graph Theory*. Cambridge University Press.
66. hello2morrow. 2014. *hello2morrow-Sonargraph* [online] available: http://www.hello2morrow.com/products/sonargraph [accessed 23/01/2015]
67. Blackwell, Alan F., Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, et al. 2001. ‘Cognitive Dimensions of Notations: Design Tools for Cognitive Technology’. In *Proceedings of the 4th International Conference on Cognitive Technology: Instruments of Mind,* 325–41. CT ’01. London, UK: Springer. http://dl.acm.org/citation.cfm?id=647492.727492.
68. Robillard, Martin P., and Gail C. Murphy. 2003. ‘FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code’. In *Proceedings of the 25th International Conference on Software Engineering*, 822–23. ICSE ’03. Washington, DC, USA: IEEE Computer Society. http://dl.acm.org/citation.cfm?id=776816.776969.
69. Kastner, C., A Dreiling, and K. Ostermann. 2014. ‘Variability Mining: Consistent Semi-Automatic Detection of Product-Line Features’. *IEEE Transactions on Software Engineering* 40 (1): 67–82. doi:10.1109/TSE.2013.45.
70. Burke, Joseph. 2012. ‘Web Architecture Dependencies & Web Architecture Recovery Techniques’. Master’s Thesis, University of Limerick.
71. Hassan, Ahmed E., and Richard C. Holt. 2002. ‘Architecture recovery of web applications’. In *Proceedings of the 24th International Conference on Software Engineering* (ICSE '02). ACM, New York, NY, USA, 349-359. DOI=10.1145/581339.581383 http://doi.acm.org/10.1145/581339.581383
72. Parnas D.L.. 1994. ‘Software Aging’. *Proceedings of the 16th International Conference on Software Engineering*: 279-287
73. Romano Daniele and Martin Pinzger. 2011. *'Using Source Code Metrics to Predict Change-Prone Java Interfaces'*. Proceedings of the 27th IEEE Conference on Software Maintenance. pp 303-312.

74. Cheesman, John and John Daniels. 2000. *UML Components: A Simple Process for Specifying Component-Based Systems*. Addison-Wesley.
75. Dubochet, Gilles, Chris Exton, and Jim Buckley. 2009. 'Computer Code as a Medium for Human Communication: Are Programming Languages Improving?' In *Proceedings of the 21st Working Conference on the Psychology of Programmers Interest Group*, 174-187.

76. Rosik, Jacek. 2015. ‘A Continuous Approach for Software Architecture Consistency’. Ph.D. Thesis, University of Limerick.
77. Mitchell, Mark L. and Janina M. Jolley. 2004. *Research Design Explained*. Fifth edition, Thomson-Wadsworth.
78. Yin, Robert K. 2003. *Case Study Research: Design and* Methods, 3rd edition, Sage Publications, Thousand Oaks, CA.
79. Shuttleworth, Martyn. 2009. *Types of Validity-An Overview* [online] available: https://explorable.com/types-of-validity [accessed 23/01/2015].
80. Greenwood Davydd J. (Ed.). 1999. *Action Research: From Practice to Writing in an International Action Research Development Program* John Benjamins Publishing ISBN: 9027217785.
81. Noland, E. William. 1959. *Hawthorne Revisited. By Henry A. Landsberger. Ithaca, New York.* Social Forces 37(4): 361-364.