JITTAC: A Just-in-Time Tool for Architectural Consistency

Jim Buckley, Sean Mooney, Jacek Rosik
Lero/CSIS
University of Limerick
Limerick, Ireland
{Jim.Buckley, Sean.Mooney, Jacek.Rosik}@ul.ie

Nour Ali
Visual Modelling Group
University of Brighton
Brighton, UK
N.Ali2@brighton.ac.uk

***Abstract*—Architectural drift is a widely cited problem in software engineering, where the implementation of a software system diverges from the designed architecture over time causing architecture inconsistencies. Previous work suggests that this architectural drift is, in part, due to programmers' lack of architecture awareness as they develop code. JITTAC is a tool that uses a real-time Reflexion Modeling approach to inform programmers of the architectural consequences of their programming actions as, and often just before, they perform them. Thus, it provides developers with Just-In-Time architectural awareness towards promoting consistency between the as-designed architecture and the as-implemented system.**

JITTAC also allows programmers to give real-time feedback on introduced inconsistencies to the architect. This facilitates programmer-driven architectural change, when validated by the architect, and allows for more timely team-awareness of the actual architectural consistency of the system. Thus, it is anticipated that the tool will decrease architectural inconsistency over time and improve both developers' and architect's knowledge of their software's architecture. The JITTAC demo is available at: http://www.youtube.com/watch?v=BNqhp40PDD4

***Index Terms*—Reverse Engineering, Software architecture discovery, software architecture consistency, compliance**

I. Introduction

Even if software architects produce a well-defined, well evaluated and well-documented architecture for a system, this architecture must be embodied in the system's implementation for the associated quality requirements to be realized. This situation is referred to in the literature as Architectural Compliance [7] or Architectural Consistency [5]. There are many challenges to achieving on-going Architectural Consistency, including time pressures on the development team during implementation/evolution, and lack of architectural awareness on the part of (possibly new) programmers. Hence this consistency can decrease during software development lifecycles, and the recovery costs can be large, as is amply illustrated in [8], page 16.

In an in-vivo case study performed by several of the authors [6], a Reflexion Modelling approach was applied at 3-monthly intervals during the re-development of a commercial system, with a view to preventing the introduction of architectural inconsistencies in that system *as* development proceeded. In this particular scenario, architectural control was deemed important by our industrial partner, as architectural drift and degeneration were the prime motivators for the re-development of the system in the first place. However, even in these circumstances the authors found that the developers were reluctant to retrospectively address the architectural inconsistencies they had introduced.

This paper presents JITTAC, an Eclipse plug-in (http://www.youtube.com/watch?v=BNqhp40PDD4), which aims to address this concern by supporting on-going architecture consistency for software development teams in a real-time, proactive manner. It allows architects to check if the currently implemented system is consistent with the as-designed architecture, as an initial basis for on-going consistency checking. Subsequently, it informs developers, as they code, of the architecture-consistency implications of their changes. It is envisaged that such a tool will lessen the introduction of inconsistencies over time and will make programmers more aware of the architectural consequences of their coding actions.

Section 2 discusses the JITAAC tool in more detail. Section 3 presents our preliminary evaluations of the tool and our proposed future studies. Section 4 discusses some of the related work. Finally, section 5 highlights conclusions and further work.

II. A Walkthrough of Jittac

Consider the scenario where an architect is faced with a system whose implementation may have drifted from its original as-designed architecture. 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 component and the architectural models and mappings can be defined incrementally and iteratively.

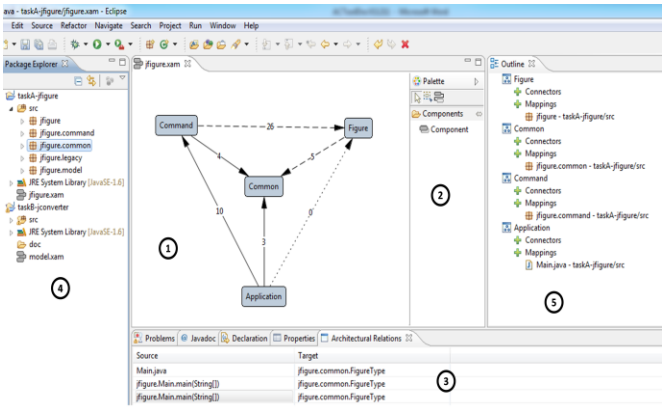

Fig. 1. An Architect's view of the JITTAC tool

As mappings are defined between the source code and the architectural model, the results of a Reflexion-Modelling type analysis are presented, in terms of the edges between the modeled architectural elements: Solid edges represent convergences where there exists a relationship both in the architectural model and in the source code implementation. Dashed edges represent divergences where there is a relationship in the implementation but not in the architecture. Dotted edges represent a relationship specified in the architecture, but not present in the implementation. Typically, the architect will focus on the latter two types of 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 Command and Common). JITTAC then allows the architect to click on the Source in the Architectural Relations view, to navigate to the associated source code. If that code is changed to address the inconsistency, this change gets reflected back to the architectural model instantaneously and the divergent edge becomes a convergent one. For a fuller description of this Architecture Recovery functionality, please refer to our description of the prototype version of the tool in [1].

While the prototype tool does provide some functionality towards addressing on-going architectural drift, JITTAC builds on this functionality substantially, to more fully consider the scenario where the architectural drift has been addressed and the architect is happy for development work to proceed. As the programmers work, they are given an enriched coding view (see figure 2) where any architectural inconsistency they introduce is marked in the coding margin on saving (figure 2(a)). In addition, as they code, an enriched auto-complete function re-ranks and color-codes the auto-complete options (again see figure 2(a)) in an architecture-aware fashion. Auto-complete options that would result in an architecturally consistent dependency are colored green and ranked first. Auto-completes options as-yet unmapped to the architectural model are colored orange and ranked second. Finally, auto-completes that would result in architectural inconsistencies are colored red and are ranked third. Thus, programmers become architecturally aware Just-In-Time: just before they commit to their source code change.

In another addition to the prototype tool, JITTAC allows the developer to right click on any line of source code that is causing an inconsistency (see figure 2(b)) and, through that, navigate directly to the architectural model to view the inconsistency, highlighted and in context. In fact, JITTAC allows architects to define several architectural models for each subject software system. Each model can provide a different architectural view or granularity. So, when the programmer introduces an inconsistency in the code, the tool navigates the programmer to the specific architectural model associated with that inconsistency.

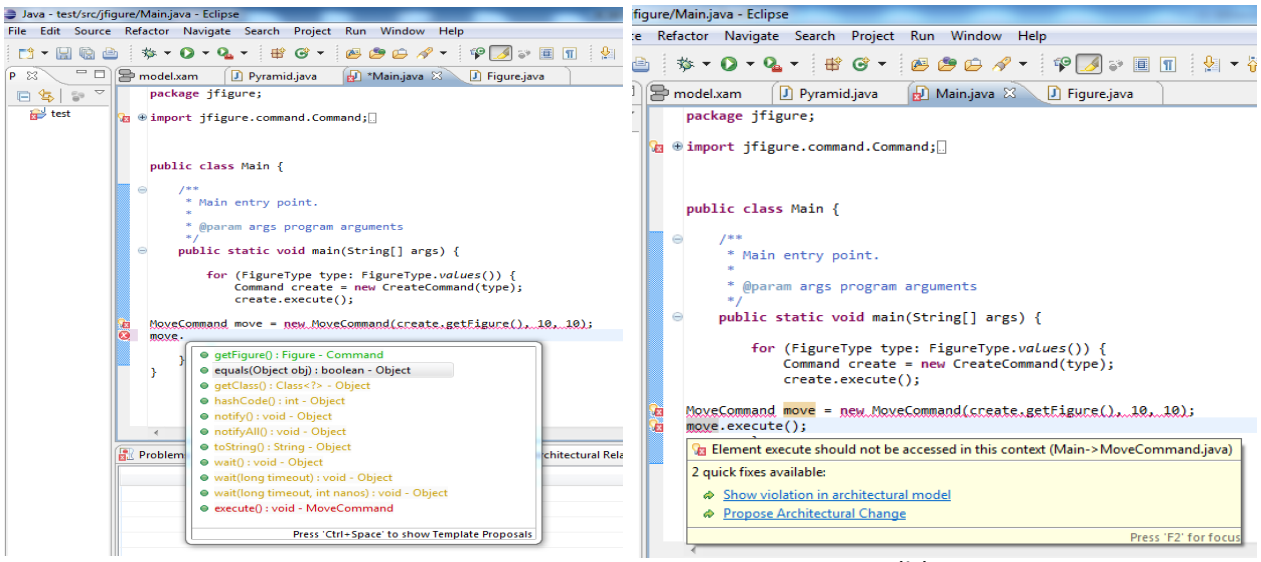

Fig. 2. A Developer's view of the JITTAC tool

Finally, the same right-click interface (see figure 2(b)) allows programmers, if they believe that the dependency is justified, to email the architect with a rationale for inclusion of the specified architecturally-inconsistent code (through the 'Propose Architectural Change' option).

III. Empirical Evaluation

In-vivo evaluation of the initial prototype version of JITTAC was carried out at two financial software companies based in Ireland, as was reported in QoSA 2012 [1]. This prototype version was called the ACTool (http://www.lero.ie/project/rca/arc) and it had many of the architecture facilities provided by the final tool: It had facilities for the architect to make explicit their designed architecture and to check the implementation's consistency with that architecture.

This prototype tool was used to check the consistency of three commercial systems in these organizations with their as-designed architecture, these systems ranging from 35KLOC to over 2.2MLOC. In general the real-time aspect of the system was well received:

"...the results were instant, that when you dragged your package or class it showed violations straight away"

This real-time feedback often prompted participants to generate model entities directly from the source code: dragging packages or classes from the package explorer directly onto the architectural canvas to create architectural components and get immediate feedback on their dependencies. This was typically done when the participant was happy with a 1:1 mapping between a software entity in the package explorer (code) and a proposed architectural component. They also did it when they wanted quick feedback on the workings of a package or class, or when they wanted confirmation that most of the observed inconsistencies in a model were due to one specific software entity. In one case, this was the default model-building behavior of the participant: a behavior that seems to directly conflict with Reflexion Modelling principles of building an as-envisaged model first and then checking it. This real-time feedback behavior was observed in all 3 sessions.

However, the prototype tool did not have all the facilities to support on-going architectural consistency checking during continued development of the system. Specifically, it did not have the enriched autocomplete of the final prototype, the ability to navigate to and highlight a programmer-coded inconsistency in the architectural model or the ability to email the architect to notify them of introduced inconsistencies. In addition, it did not have the facility to model, and show consistency with, more than one designed architecture for a given system.

JITTAC, the current prototype that implements these features, has currently been trialed in one other organization. Here it was again used to check the consistency of a commercial system with its as-designed architecture. The part of the system modeled was approximately 150KLOC. The results were consistent with those of the initial study: the participant used a real-time feedback approach as their default behavior and was positive about it.

Our next round of evaluation will concentrate on the just-in-time capabilities of the tool for controlling architectural consistency in an on-going basis. This will take the form of an in-vivo case study where the architect in an organization will check the level of consistency between the implementation of one of their commercial systems and its envisaged architecture, as of a specific date (see figure 3). He will use the company's commit repository to do the same evaluation for a past release of their system. This will allow us determine the rate of inconsistency introduction, over the lifetime of a release, without the JITTAC tool's support.

The tool will then be circulated to the company's programmers and they will be shown how to use it. At the same time, they will complete a short quiz focused on the architectural consistency of a number of source code dependencies. This will allow us to quantify their knowledge of how the as-designed architectural model maps to the code they work on.

At the next release of the system and after the programmers have used JITAAC to develop this release, the programmers will be asked to complete another quiz and the architect will again check the consistency of the system. These measures will allow us to determine any change in the programmers' knowledge of how the as-designed architectural model maps to the code base and it will allow us determine the rate of inconsistency introduction, over the lifetime of that release, with the JITTAC tool's support.

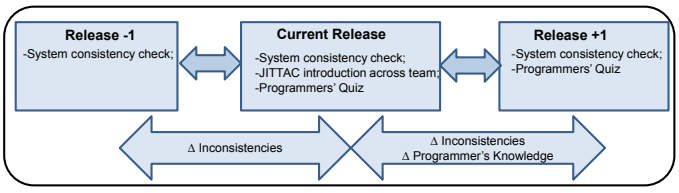

Fig. 3. JITTAC longitudinal evaluation plan

The difference between the rate of inconsistency introduction over the 2 releases will give us an initial indication of the utility of the tool, as will comparing the results of the quizzes that the programmers undertake. This, when coupled with qualitative analysis of extemporaneous factors over the 2 releases (programmer diaries and changes to the architectural model driven by programmers) will provide a rich data set on JITTAC's capabilities and limitations. This will ideally prompt more controlled studies of individual aspects of and the overall approach underpinning JITTAC.

IV. Related Work

JITTAC is based on the Reflexion Modelling approach proposed by Gail Murphy et al. [1], who produced the JRMTool prototype tool [3]. This tool was batch-oriented: a model is defined, the mappings to the code created and an analysis tool is executed to give feedback to the user at periodic intervals. There are many differences between JITTAC and

JRMTool, but the basic ones are that JITTAC provides instant feedback as code is mapped to the architecture, the architectural model is changed instantaneously when the code is updated, it provides developers with inconsistency awareness as they are developing through margin alerts and auto-complete, and it allows architecture team awareness: Developers can send emails to architects when inconsistencies are introduced.

Passos et al. [4] reviewed the most promising architecture consistency approaches in their 2010 paper. Ultimately, they suggested that Reflexion Modelling with real-time feedback was the most appropriate avenue, based on a well-defined existing process. The JITTAC prototype tool presented in this paper supports this real-time feedback and goes beyond it.

Most closely related to this work is the work of Knodel [7]. He has simultaneously developed a real-time Reflexion Modelling-based tool called SAVELife that gives real-time feedback to architects and developers on the consistency between their designed architecture and their implementation. However, SAVELife does not provide intellisense support to developers, the navigation from the code to the architectural model, or the communication between the developers and architects.

A recent approach for detecting architectural inconsistencies is the one defined by Haitzer and Zdun [9]. They define a Domain Specific Language (DSL) that can be used to generate architectural models from source code and identify inconsistencies. However, inconsistencies are not visually shown in the architecture model since they are implemented as rules. In addition, when code is updated the architectural models are not updated automatically; changes to the DSL code are needed in several cases. It also does not provide real-time architecture knowledge to developers as they update the code or architecture team awareness.

V. Conclusions and Further Work

In this paper, we present JITTAC a tool that provides just in time architectural consistency support to reduce the architectural drift phenomena. The tool allows both architects and developers to acquire architectural knowledge and receive feedback in real-time. It allows architects to check the consistency between the implemented architecture and the as-designed one, both during development and retrospectively. In addition, developers receive architecture feedback as they develop, because the tool informs them of the architecture consistency implications of their changes. It also facilitates the communication between developers and architects, allowing developers to express the rationale behind their inconsistencies when they deem them appropriate.

We have evaluated the JITTAC tool on four commercial projects to check their architecture consistency. However, we have not evaluated the tools features and usefulness for the on-going architectural support of developers. To evaluate this, we have designed a protocol to perform a longitudinal study in a commercial setting and, in liaison with a commercial partner, plan to execute it in the near future.

Our future work directions are towards extending the tool to provide fuller support for architecture recovery and consistency. The current stand-alone plug-in will be reengineered to a client-server implementation where a central repository of architectural models will be preserved on the server. This central repository will provide increased consistency in the team's architectural models over time, providing a greater degree of team awareness.

In addition, currently the architectural models generated in the tool reflect implementation-based, inter-component dependencies based on import statements, invocations and field accesses only. We hope to trial the visualization of novel dependencies such as annotation-similarity and OO constructs like inheritance. Finally, we intend to scale up the approach to probe architectural consistency for web systems and service oriented ones. For this we will have to apply different (dynamic) analysis techniques: not only the static analysis ones which are currently supported by JITTAC.

Acknowledgment

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie). Video production acknowledgements to Leah Morgan, Eoin Murray and David Moya Garcia.

References

[1] N. Ali, J. Rosik, and J. Buckley, “Characterizing Real-Time Reflexion-based Architecture Recovery in Practice” Proceedings of Quality of Software Architecture, 2012.

[2] G. C. Murphy, D. Notkin, K. J. Sullivan. “Software reflexion models: Bridging the gap between source and high-level models,” Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 18–23, 1995.

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

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

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

[6] Rosik, J., LeGear, A, Buckley, J. Babar, M.A., Connolly, D., "Assessing Architectural Drift in Commercial Software Development: A Case Study." SPE 41(1): 63-86, 2011.

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

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

[9] T. Haitzer, U. Zdun, “DSL-based Support for Semi-Automated Architectural Component Model Abstraction Throughout the Software Lifecycle”, Proceedings of Quality of Software Architecture (QoSA’12), pp. 61-70, 2012