Empir Software Eng (2018) 23:224–258
DOI 10.1007/s10664-017-9515-3

Architecture consistency: State of the practice, challenges and requirements

Nour Ali[1] · Sean Baker[2] · Ross O'Crowley[3] · Sebastian Herold[4] · Jim Buckley[2]

Published online: 15 May 2017

Abstract Architecture Consistency (AC) aims to align implemented systems with their intended architectures. Several AC approaches and tools have been proposed and empirically evaluated, suggesting favourable results. In this paper, we empirically examine the state of practice with respect to Architecture Consistency, through interviews with nineteen experienced software engineers. Our goal is to identify 1) any practises that the companies these architects work for, currently undertake to achieve AC; 2) any barriers to undertaking explicit AC approaches in these companies; 3) software development situations where practitioners perceive AC approaches would be useful, and 4) AC tool needs, as perceived by practitioners. We also assess current commercial AC tool offerings in terms of these perceived needs. The study reveals that many practitioners apply informal AC approaches as there are barriers for adopting more formal and

The original version of this article was revised due to a retrospective Open Access order.

Communicated by: Vittorio Cortellessa

Nour Ali
nouraliirshaid@gmail.com

Sean Baker
sean.patrick.baker@gmail.com

Ross O'Crowley
ross@orkidtech.com

Sebastian Herold
sebastian.herold@kau.se

Jim Buckley
jim.buckley@ul.ie

[1] Brunel University, Uxbridge, London, UK

[2] Lero, University of Limerick, Limerick, Ireland

[3] Red Orkid Limited, Dublin, Ireland

[4] Karlstad University, Karlstad, Sweden

explicit approaches. These barriers are: 1) Difficulty in quantifying architectural inconsistency effects, and thus justifying the allocation of resources to fix them to senior management, 2) The near invisibility of architectural inconsistency to customers, 3) Practitioners' reluctance towards fixing architectural inconsistencies, and 4) Practitioners perception that huge effort is required to map the system to the architecture when using more formal AC approaches and tools. Practitioners still believe that AC would be useful in supporting several of the software development activities such as auditing, evolution and ensuring quality attributes. After reviewing several commercial tools, we posit that AC tool vendors need to work on their ability to support analysis of systems made up of different technologies, that AC tools need to enhance their capabilities with respect to artefacts such as services and meta-data, and to focus more on non-maintainability architectural concerns.

Keywords Software architecture · Consistency · Architectural drift · Empirical study · Architecture recovery · Architecture conformance and erosion

1 Introduction

Architecting software systems is an integral part of the software development lifecycle, providing a basis for achieving non-functional requirements (Bachmann et al. 2005). However, even if the software architecture is well documented, designed and evaluated, the architecture must be embodied in the implementation to satisfy these requirements. When a software system's implementation diverges from its designed architecture, it is referred to as architectural drift or architectural erosion (De Silva and Balasubramaniam 2012). This usually happens during software evolution (Behnamghader et al. 2016; Ming et al. 2015) when the software undergoes changes as a result of bug fixes and updates, but may also happen during initial implementation of the system (Garlan and Schmerl 2004). Architectural drift can hinder evolution of systems, inhibit fulfilment of the goals of the architectural design (Bang and Medvidovic 2015) and, even if recognized, can prove costly to resolve (Knodel 2010).

Architecture Consistency (AC) techniques have been proposed to address architectural drift (Knodel 2010; Murphy et al. 1995; Rosik et al. 2011). Several of these techniques are Architecture Recovery approaches where artefacts of the implementation (for example, source code dependencies) and/or artefacts that are available after the system's implementation (for example dynamic traces) are analysed to extract the implemented architecture automatically. Other approaches typically assess the artefacts for some form of cohesion and coupling, resulting in an implementation-derived architecture for the system (Girard and Koschke 1997; Verbaere et al. 2008). However, these approaches have several limitations with respect to AC. For example, they are driven by the code-base and not by consistency with the architect's agenda. Additionally, if the implemented architecture of the code-base is flawed through erosion, the cohesion and coupling measures reflect this flawed architecture, resulting in a flawed recovered architecture (Rosik 2014).

An alternative set of approaches concentrate of aligning an explicit representation of the architect's designed architecture with the implemented architecture. Reflexion Modelling (RM) (Murphy et al. 1995; Murphy and Notkin 1997), for example, allows the architect define the system's planned architectural model graphically, in terms of logical components and their connections. Later, the elements of the source code are mapped onto the vertices of this graphical model (representing the components of the architectural model) and the relationships between the mapped source code elements are determined through static analysis of the code-base. These relationships are assessed for consistency with the connections between the components stated in the designed architecture, thus illustrating where the source code dependencies are inconsistent with the designed architecture. It has been argued that RM is very effective in identifying architectural inconsistencies in the implementation of software systems (Passos et al. 2010; Rosik et al. 2011) and several other approaches have been proposed for AC based on RM (Buckley et al. 2013; Knodel 2010; Stoermer et al. 2006). In addition, several other types of approaches have also been suggested based on, for example, matrix representations (Sangal et al. 2005) and more specific rule-based checking (Passos et al. 2010).

Many AC approaches have been empirically evaluated to determine their utility (e.g., (Ali et al. 2012; Knodel 2010; Mattsson 2012)) but these evaluations have predominantly focused on their effectiveness in *identifying* inconsistencies. In addition, while these evaluations have provided favourable results, few evaluations have been performed to understand how practitioners perceive AC approaches and their needs for adoption. Indeed, De Silva & Balasubramaniam's recent survey on academic and industrial AC approaches and tools indicates low adoption of these approaches in practice (De Silva and Balasubramaniam 2012). This paper focuses on the needs of real-world practitioners, in terms of the practical software development situations for applying AC and the tooling requirements they consider important for adopting an AC approach. It also identifies the barriers they perceive to adopting AC approaches in practice. Hence, by interviewing 19 senior software engineers and reviewing the tool offerings in this area, this paper focuses on the following contributions:

- Identification of any practises that these senior software engineers' companies undertake to achieve AC;
- Identification of the barriers to adopting/performing more explicit AC approaches in these companies;
- Situations where they perceive AC approaches would be useful;
- Identification of the AC tool needs they perceive;
- Assessment of current AC tool offerings in terms of these perceived needs.

The paper is structured as follows: Section 2 presents a review of the literature on AC and its terminology as used throughout the paper. Section 3 outlines our research questions and the research methodology undertaken. Section 4 presents our findings and a review of tooling in this area to assess, against these findings. Section 5 discusses our findings and threats to the validity of the study. Finally, section 6 concludes the paper.

2 AC Literature

2.1 Existing Approaches

Architecture Consistency approaches and tools can be classified based on how they extract the implemented architecture from systems. This can be by using static or dynamic analysis, or both. AC approaches that extract the software architecture by using dynamic analysis use 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 (De Silva and Balasubramaniam 2013). Garlan et al. (Garlan and Schmerl 2004) also discuss a framework for discovering the architecture at runtime. They suggest using state machines to capture the mapping between the runtime system and the system's architecture as a set of events. In another example Ganesan et al., extract execution traces of a running system as coloured petri nets which were investigated for architectural inconsistencies (Ganesan et al. 2008). The work of Popescu and Medvidovic 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 (Popescu and Medvidovic 2008). Dynamic approaches are in the minority though and, according to the review performed by Ducasse and Pollet (2009), most AC approaches based on dynamic analysis extract design views rather than architectural ones.

Approaches and tools that use static analysis, extract the software architecture by analysing source code (Knodel et al. 2008b). For example, in RM (Murphy et al. 1995) based approaches, the architect is allowed to specify their own as-intended architecture in terms of the logical components of the system and the anticipated dependencies between these components, usually through a vertices-and-edges type diagram. The architect can then map sets of source code entities (projects, packages, classes) to each vertex, or logical component. Then a static analysis of the system identifies the actual source-code dependencies between the vertices, allowing the architect to focus on the inconsistencies between the anticipated dependencies and the actual dependencies. The architect may then choose to align the dependencies by changing the source code, changing the as-intended architectural model, or changing the mapping between the model and the source code.

The work of Knodel (2010) extends RM with real-time inconsistency identification with a tool called SAVELife that gives feedback to architects and developers on the consistency between their designed architecture and their implementation as they proceed. However, SAVELife does not provide auto-complete support to developers, where potential inconsistencies are signalled to the developers as they create dependencies. Likewise, it does not support navigation directly from the code to the architectural model, or prompt communication between the developers and architects.

In contrast, JITTAC (Buckley et al. 2013) is a tool that extends RM with real-time architecture feedback, providing instant feedback as code is mapped to the architecture, and as the developer writes code: the architectural model is changed instantaneously. It provides developers with inconsistency awareness as they are developing through margin alerts, through auto-complete alerts and through the architectural model itself, also integrating a facility where developers can email architects with alerts when inconsistencies are introduced. In addition, a recent study by Pruijt et al. (2016) has shown that JITTAC was more holistic and effective, in terms of detecting all the source code dependencies in software systems than the other academic and commercial offerings in the area.

A recent approach for detecting architectural inconsistencies is the one defined by Haitzer and Zdun (2012). 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 a holistic architectural model since the enforced consistencies 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 and so it cannot provide real-time architecture knowledge to developers as they update the code.

Combined Static and Dynamic AC is based on applying both type of analysis, usually in sequence. An example is the Pattern Lint work done by Sefika in (1996) where static analysis, followed by dynamic analysis, was used. This complimentary dynamic analysis was used to detect the prevalence of the violations originally identified through static analysis, in the executing system.

Passos et al. (2010) reviewed the most promising architecture consistency approaches in their 2010 paper. They evaluated three candidate approaches: Dependency Structure Matrices (Sangal et al. 2005), a rule-based approach called Source Code Query Languages (Verbaere et al. 2008) and RM (Murphy et al. 1995). In their conclusion, they suggested that RM with real-time feedback was the most appropriate avenue for companies interested in systematically achieving AC, based on a well-defined existing process centred on holistic high-level models as defined by architects.

A more generalized approach to inconsistency issues in software design is proposed by Bang and Medvidovic (2015). They introduce the FLAME framework which is capable of analysing the operations performed concurrently on an (architectural) model by several users and to proactively report on inconsistencies detected by different consistency checking engines. This mechanism, in principle, would also be usable for checking architecture-to-code consistency. Their experiments show evidence that this proactive detection (as per JITTAC) is beneficial and leads to higher quality models and quicker inconsistency resolution. Another approach, proposed by Behnamghader et al., looked at the related field of assessing architecture change over a system's evolution, by comparing older versions of the architecture to newer versions of the architecture (Behnamghader et al. 2016). They did so using metrics that quantified the amount of architectural change (a2a) and architectural overlap (measured at a component-implementation level: c2c) over time.

Another set of approaches comes from the related field of software architecture-to-implementation traceability. In these approaches, information about how entities in the software architecture relate to units of the source code is represented in explicitly stored *trace links*. In contrast to most of the consistency checking approaches mentioned above, information about the links between software architecture and code becomes a first-class artefact or at least a central element whose creation, evolution and analysis can be supported by tools. Buchgeher and Weinreich propose an approach in which trace links are semi-automatically created between architecture design decisions and architectural/implementation units based on an IDE recording the activities of the developer (Buchgeher and Weinreich 2011). Developers have to provide information about the design decision they are going to work on first before they begin their task. The IDE records actions, such as the creation of a new component, and presents the developer a list of entities created or modified to be linked to the design decision. A similar approach, recording architectural changes for architecture to code mapping, is followed by Zheng and Taylor (2003, 2011). In their approach also, mappings between architectural behaviour and refined behaviour in source code can be created and managed. Murta et al. propose the ArchTrace tool particularly focusing on supporting the evolution of trace links between the independently evolving software architecture and the implementation of a system, in order to avoid inconsistencies (Murta et al. 2008). Changes in any of these artefacts can trigger so-called policies which describe if and how the set of trace links has to be modified to insure that a change fulfils the constraints triggering the policy.

Nguyen proposes *MolhadoArch* which builds upon a hypertext-based software configuration management system (SCM). Structured documents, such as software architecture specifications or source code, are represented as attributed graphs (Nguyen et al. 2005). Edges between elements in different documents, such as trace links between a software architecture and source code, are supported and versioned as well, such that this approach can support the evolution of trace links. In the context of model-driven software development, a few approaches to traceability further enriched the *traceability model*, allowing for richer information attached and related to the trace links. Tran et al., for example, define and use a metamodel supporting different types of links for traceability between system models at different levels of abstraction (Tran et al. 2011). Mirakhorli et al. propose an approach to automatically reconstruct trace links between architecture tactics and classes in code (Mirakhorli et al. 2014, 2012). They apply information retrieval and machine learning techniques for training a classifier that identifies classes related to tactics based on code snippets and tactic descriptions.

Javed and Zdun provide a systematic literature review of approaches to traceability between software architecture and source code (Javed and Zdun 2014). They state that the empirical evidence on the usefulness of traceability approaches for linking software architecture and source code is quite weak as most approaches are evaluated by single case studies or small experiments. Industrial adoption of such techniques could certainly benefit from a more mature empirical basis, especially since most approaches need adaptations in tooling based on the way individualistic behaviour of developers (Prechelt 1999).

2.2 Practitioner Studies

There have been empirical studies to explore the needs of practitioners in the area of software architecture. For example, a recent study surveyed 48 practitioners to understand the adoption of architecture description languages in practice (Malavolta et al. 2013) and defined a framework for defining new architecture description language requirements based on the results of that survey (Lago et al. 2015). The survey (Malavolta et al. 2013) also refers, to architectural consistency. However, unlike our study, these findings refer to consistency across ADL-based architectural views and not to consistency between the software system and the architecture. More aligned with our work, they found that 65% of their respondents used informal reviews to check architectural compliance in the system itself, using a variety of inputs (for example: test cases, documentation). The study reported here discusses the barriers and needs of practitioners for more formal approaches that keep software systems and their architectural intent (views) consistent. Consequently, in our empirical design protocol, we utilize semi-structured interviews instead of a survey: This is to enable a more in-depth exploration of aspects with practitioners.

AC approaches have been empirically evaluated. Studies suggest that they are effective but, by-in-large, these studies focus on identification of inconsistencies only (Lindvall et al. 2002; Tesoriero et al. 2004) or identification and removal of inconsistencies (Bang and Medvidovic 2015; Brunet et al. 2012, 2015; Buckley et al. 2015; Rosik et al. 2011). In order to assess if there was literature directly targeted at practitioners' needs in the AC area, we carried out a targeted literature review of the IEEE and ACM search engines with a query search logically equivalent to "software AND architecture AND (conformance OR consistency) AND (empirical OR practice OR study). The number of papers found, using each search engine, is detailed in Table 1. Those papers whose titles reflected a focus on software architecture were abstracted and, if subsequently deemed directed at practice experience reports or study, were reviewed in full. But, on inspection, few papers reported on practitioners' feedback on the specific techniques applied or their requirements for improved functionalities. These papers are reviewed in the text below Table 1.

Table 1 Literature review of empirically derived practitioner needs in AC

Query				IEEE	ACM
Software	AND architecture	AND conformance	AND empirical	3	8
			OR practice	14	33
			OR study	44	43
		OR consistency	AND empirical	8	81
			OR practice	34	312[a]
			ORr study	114	448[a]

[a] Only the first 100 of these articles were reviewed as, at that point, they had deviated substantially from the focus area and no articles of interest had been found in, at least, the previous 35 articles

There have been studies conducted with students. In Knodel's work (Knodel 2010), he asked student participants to fill out an 'innovation transfer success questionnaire'. In this case the reported data abstracted beyond individual perceptions to generic categories of satisfaction, predictability, usefulness, degree of novelty and adaptation. More detailed analysis of professional participants is required, as acknowledged by the author in his future work section.

Other empirical studies where practitioners' feedback has been provided are (Ali et al. 2012; Brunet et al. 2015; Buckley et al. 2015; Le Gear et al. 2005). In these works, the objective was to understand how practitioners have used the current AC approaches and report on the effectiveness of these approaches in identifying inconsistencies (e.g., what kinds of architectural inconsistencies are identified or how do developers identify inconsistencies). Several of the prevalent concerns/desires that practitioners expressed over these studies are that they wanted:

- Analysis beyond the source code to the entire system including entities like deployment descriptors and 3rd part components (Ali et al. 2012; Buckley et al. 2015).
- Enriched mapping capabilities between the designed and implemented architecture, based on lexical mapping and drag-and-drop mapping of large grained software structures like classes and packages, to architectural components (Ali et al. 2012; Buckley et al. 2015; Le Gear et al. 2005).
- The ability to enrich the modelling of the architecture, to allow, for example, the definition of interfaces to localize inconsistencies (Buckley et al. 2015).
- Scalable visualization of the designed/implemented architecture, for industrial systems (Ali et al. 2012; Buckley et al. 2015; Koschke and Simon 2003).
- The ability to rank the elements (e.g., classes, and methods) that cause the most architectural inconsistencies between architectural components (Ali et al. 2012; Buckley et al. 2015).
- Feature-oriented views of the system (Buckley et al. 2015), where a feature is considered an end-user functionality.
- The ability to specify architectural rules in a more intuitive way (Brunet et al. 2015).

Regardless of these concerns, the approaches trialled were very well received by the participants in general, as illustrated by the participant quotations included in those papers. It is surprising to note then that in their literature survey, Silva et al. (De Silva and Balasubramaniam 2012) found that there is little adoption of academic AC tools in industry. This is in line with our own literature review of the field which suggests adoption of an academic AC tool in one commercial company only (Knodel et al. 2008a). Tesoriero et al. (Tesoriero et al. 2004) suggest that the low take-up of academic tools in this area maybe because of the prototype nature of such tools, a conclusion supported by the same observations in (Rosik 2014). Hence, our study focuses on more commercial tools: It assesses the needs, as reported on by experienced software engineers, for AC approaches in practice, the associated AC tool requirements expected and a range of commercial tools' abilities to meet these requirements.

2.3 Terminology

In this section, we explain the AC terminology that is frequently used in the rest of the paper:

- *Planned Architecture (also the designed architecture or intended architecture):* This refers to the architecture that is the outcome of a design process (Hochstein and Lindvall 2005; Tran and Holt 1999).
- *Implemented architecture:* This is the architecture that is extracted from the source artefacts (Tran and Holt 1999). This architecture can also be represented in an architectural model.
- *Architecture Recovery (AR):* When the implemented architecture is extracted from the implemented system (Medvidovic et al. 2003).
- *Architectural Rule:* These are rules or constraints, explicitly expressed in the planned architecture, that are intended to be enforced in the implementation (Mattsson 2010).
- *Architecture Inconsistency:* This happens when a dependency exists in the implemented architecture but does not exist in the planned architecture or when a dependency exists in the planned architecture but is absent in the implementation (Ali et al. 2012).
- *Mappings:* In this paper, we refer to mappings as explicit relationships defined by users of AC tools in order to indicate the elements of the source code that correspond to specific elements in an architecture. These can be expressed in lexical terms, as in the jRmTool (Murphy et al. 1995) or by dragging and dropping, as in JITTAC (Buckley et al. 2013).
- *Just-In-time/Real Time Inconsistency Awareness*: The capability of providing architectural consistency awareness to developers, as they change the implementation (Buckley et al. 2013; Knodel 2010). This can be done via margin alerts, updates of the architectural model to show inconsistencies or by cues provided in auto-complete, e.g., auto-complete options can be colour coded reflecting their architectural (in)consistency (Buckley et al. 2013).

3 Empirical Study

In this section, we present our research questions and the data collection and analysis methods used in the empirical study we report. The research questions for this empirical study are as follows:

RQ1: What practises are currently in place in practitioners' companies to address AC?
RQ2: What barriers exist to adopting more explicit AC approaches in these companies?
RQ3: Which software-development situations would practitioners envision for AC approaches to be useful?
RQ4: What are the characteristics/features that practitioners require in AC tools?

To investigate these questions a set of interviews was conducted with commercial software practitioners, each playing a senior, technical, architectural role in a software development organization.

3.1 Participants and their Organizational Context

Participants were recruited through opportunistic sampling with several criteria subsequently applies (imposing purposive sampling (Oates 2005) on top of this opportunistic sampling). The opportunistic aspect of the sampling had to do with the authors' previously established contacts in the software industry. One of the authors was a past chair of the Irish Software Association meaning that he had a large number of contacts in the industry. In addition, two of the authors had connections with a number of companies, based on a previous history of architectural recovery\consistency studies (Ali et al. 2012; Buckley et al. 2015; Buckley et al. 2008; Le Gear et al. 2005). This initial pool of potential contacts was filtered down in line with several criteria. Firstly, participants had to have a senior technical role in their organization related to software architecture and had to be decision makers (budget holders/advisors), with regards to influencing the adoption of software practises or tools in that organization. All participants had to have more than 8 years' experience in Software Engineering. In total 19 such participants were approached and all 19 participated: They all confirmed they had technical architectural roles, albeit under a range of different job titles.

Seven participants came from companies that separated architectural tasks clearly from development, but eleven did not. Overall, the structure of their organizations with respect to architecture and development could be categorized as follows:

- An architectural team was responsible for the architecture of all of the company's software systems, but individual architects were also responsible for specific systems and the other developers on these systems could have some architecture responsibilities (P5, P13, P14).
- An architectural team was responsible for designing the architecture of software systems only and did not have any development responsibilities. Some architects on this team may have previously been on development teams and moved to the architecture team but others would have no previous technical/development background. In these teams, architects produced a high level architecture and a design team, or senior developers, then performed a detailed design of that architecture, which developers had to follow (P1 and P6).
- Five participants stated that, in their companies, there was an architect for each development team and the other developers in the team would not have architecture responsibilities (P2, P8, P9, P10 and P15).
- Development teams sometimes had senior developers that were more knowledgeable about architecture, and they combined development with an architectural role (P3, P4, P7, P11, P12, P16, P17, P18, P19). They would be in charge of reviewing more junior developers' work, but those junior developers could propose changes in architecture.

Table 2 gives an overview of the participants and their organizations. The 19 participants were from 17 different companies (P7/P16 and P13/P14 came from the same companies) and all of the participants were based in Ireland apart from P8, who came from Poland, and P18 and P19, who came from the UK. The companies of the participants are all international organizations with participants' teams interacting with distributed stakeholders and distributed software development teams. Seven were involved in the financial domain. All participants involved in the study have 10 or more years of experience in software engineering and all of them, had a year or more in their current organization.

Table 2 Description of the participants and their systems

Participant	Company domain	Title	Agile Soft. development?	Architect / developer separation	Size of system (LOC)	Experience in software engineering (Years)
P1	Financial	Senior Architect	No	Yes	2.5 M	>22
P2	Financial	Lead Architect	Yes	Yes	>2 M	>15
P3	Software Contracting	Lead Architect	Yes	No	100 K/ dev. Team	>15
P4	Financial	Principle S. Engineer	Yes	No	~100 K	18
P5	Financial	Head S. Engineer	Yes	Yes	5 M	>15
P6	Software Dev.	Architect	No	Yes	N/A	>10
P7	Ecommerc & Cloud Services	Sen. Mgr. Software Dev.	Yes	No	>2 M	>20
P8	IT and Telco Mgt Software	Director of Prof. Services	Yes	Yes	N/A (2 to 30 man-years of effort)	>20
P9	Business Information	Technical Leader	Yes	Yes	5–20 M	>10
P10	Financial	Senior Architect	Yes	Yes	0.5 M	>10
P11	Financial	Technical Architect	Yes	No	40K/ dev. Team	>10
P12	Software Contracting	Lead Developer	Yes	No	>5 M	10
P13	Manufacturing	Sen. Software Developer	Yes	No	>5 M	18
P14	Manufacturing	Principal S. Engineer	Yes	No	>5 M	21
P15	Energy Mgt.	Sen. Soft. Architect	Yes	Yes	100 K	>17
P16	Ecommerc & Cloud Services	Sen. Dev. Engineer	Yes	No	>2 M	>15
P17	Social Software	Senior Lead	Yes	No	20 K/ dev. Team	>24
P18	Financial	Team Lead	Yes	Yes	~1 M	>20
P19	Travel	Analyst/Developer	No	No	~1 M	>30

Of the 19 participants interviewed, 16 claimed to be using agile software development practises. Some of them had moved from a waterfall model to agile software development incrementally and mixed the two approaches. While this sample might seem disproportionately biased towards agile-using software development teams, recent surveys suggest that this is reflective of the current and emerging software development landscape (West et al. 2010; Techbeacon Is Agile the new Norm 2017). For example, techbeacon (Techbeacon Is Agile the new Norm 2017), in their recent survey of over 600 development and IT professionals found that 67% of the associated organizations used "pure" agile approaches or an approach leaning towards agile. 24% used what they stated was a hybrid approach with only 2% using a "pure waterfall" approach and 7% using an approach leaning towards waterfall. In our study, 84% of the sample population claimed to use agile or a hybrid as compared to the 91% reported in the techbeacon survey, but it should be noted that the techbeacon survey suggested a strong trend towards adoption of agile methods in organizations going forward.

Such a focus leads to interesting questions when juxta-positioned with architectural practice. For example, while several agile experts have expressed a desire for explicit attention to architecture early in development (Barbar 2009; Madison 2010) and Falessi et al. (2010) found that agile developers agreed on the value of architectural design patterns, Krutchen (2010) notes how ambivalent, or even derogatory, 'Agilistas' are to software architecture in general, describing it as "something of the past, or even evil". S, Krutchen and other respected commentators argue for a leaner scoping of architecture to align it with agile practises (Boehm 2002; Coplien and Bjornvig 2010; Krutchen 2010; Madison 2010). Such a vision includes lower architectural overheads, and more direct communication between participants (and thus, lesser documentation); one which is less proscriptive in the detail and allows 'wiggle room' for developers; one which is firmly based on value for cost.

However, our literature review suggests little material on actual perception of architecture by this community and their in-vivo practice. Notable exceptions include the case studies performed by (Ihme and Abrahamsson 2005) who looked at how correct usage of patterns in agile led to improvements in the resultant system, and by (Barbar 2009) which reported on how architectural practises were applied in an agile project and the challenges they seemed to encounter. In terms of barriers to architecting and actual uses, Falessi et al. (2010) performed a survey but this was limited to one organization at one location: The study presented here focuses on a larger, more representative sample of companies, but specifically with regard to architecture consistency.

Only one of the participants was using an AC tool at the time of the interview (P2). He said that he was using Structure 101 (Structure 101 Structure 101 webpage 2016). 8 were aware of AC research tools and trailed them at some stage, in the context of collaboration in academic projects: P1, P4 and P5 had used the ACTool (Ali et al. 2012) which is a previous version to JITTAC (Buckley et al. 2013), and P12 has used the ACTool and the jRMTool (Murphy et al. 1995), P13 and P14 worked with an adaptation of the jRMTool that could parse Progress (Le Gear et al. 2005), and P15 and P17 used the original jRMTool. Of the 8 participants that did not use any tool, P9 did consider using Structure 101 (Structure 101 Structure 101 webpage 2016) but did not adopt it. When the participants were asked if they knew of any other relevant AC tools not mentioned above, they said they didn't. Several mentioned SonarQube (2014), but did not consider it as an AC tool exactly, as it does not provide a view of the architecture, and does not explicitly support the specification of architectural rules.

3.2 Data collection and Analysis

Qualitative data (Yin 2003) was collected through face-to-face, semi-structured interviews, apart from one which was done by phone (P8). Participants were first asked to watch a demonstration of a RM based AC tool called JITTAC (Buckley et al. 2013), to provide an initial context for the topic of the interview (https://www.youtube.com/watch?v=BNqhp40PDD4). Then a guiding interview script (Available online at http://www.cem.brighton.ac.uk/staff/na179/interviewQuestions.htm), which was designed to address our research questions, was presented to the participants. The script served to heighten consistency in the interview protocol and ensured coverage over all aspects of the study. The questions were largely open-ended, allowing respondents to convey their experiences and views on AC. The interviews, on average, lasted just over an hour. They were audio-recorded where the participants agreed. These recordings were transcribed for analysis (P12-P19). In several cases, the participants preferred not to be audio-recorded but did allow the manual recording of contemporaneous notes during the discussion. This note-taking was focused on recording the participants' opinions on AC with respect to the research questions and recording notable quotes from the participants.

Data was analysed using qualitative methods as described by Seaman (Seaman 1999). We followed a largely Grounded Theory approach (Strauss and Corbin 1998) but within the context of the research questions. We first started by reading the transcripts and notes. Each time, we read an interview we marked relevant utterances. Often, we went back and reviewed interviews and notes we previously read, after reading another interview, to locate relevant answers and determine possible themes.

Then, we started formal coding and analysis of the interviews. To do this, we introduced the transcribed recordings and notes into the Qualyzer tool (Qualyzer 2016). The Qualyzer tool allows researchers to annotate (or code) text, and generally view and organize codes into hierarchies. In general the lead author annotated the text and the last author subsequently reviewed this coding independently. Any persistent inconsistencies were resolved during a series of meetings within the group of authors.

We then started formal coding (annotating) the interviews iteratively. In this case, we went through each single interview and added codes. These codes were usually low level and we adopted the terms that the interviewees used as at this stage: we did not interpret the semantics. Some of the utterances were annotated with several codes. For example, two codes "Usefulness for Architecture Consistency" and "Usefulness for Just In Time feedback" often overlapped: Just In Time feedback could be part of "Usefulness for Architecture Consistency" but some practitioners specifically quoted "Just In time" so we gave them two different codes.

In the two texts we started coding on, we performed horizontal analysis, i.e., the codes emerged from the interviews. After performing the coding of the third interview, we started performing vertical analysis. We reviewed the codes of the previous interviews. This process was facilitated by the Qualyzer tool as you can easily view the codes, navigate to the text and if changes are performed to the codes they are automatically reflected in all annotations.

During this process we grouped previous codes into clusters, if possible. Figure 1 (a) shows a subset of our coding. For example, at the top of the figure we created a code "Not Useful for Just In Time and immediate feedback" which was a code for all parts of interviews where participants intoned that Just In Time AC would not be useful. As can be seen in the figure, there were six places where practitioners mentioned this. The same code could be used several times in the same interview but in different locations of the interview, or in different interviews.

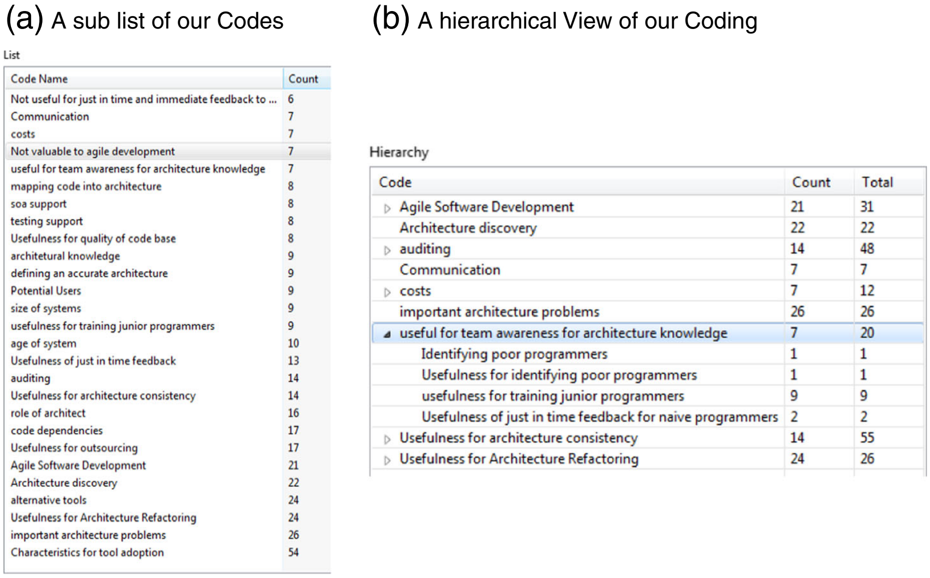
Finally, we structured the codes in Fig. 1 (a) using the hierarchy view in Qualyzer as shown in Fig. 1b. Some of the quotes would be directly related to upper hierarchy categories and some of them would refer to the subcategories. For example, Fig. 1b shows the hierarchy for a code called “usefulness for team awareness for architecture knowledge” which is composed of four other codes.

4 Results

4.1 Main Architectural Goals of Practitioners

As a prelude to asking practitioners about AC, they were asked about the core architectural challenges they face in their daily work. The most prevalent architectural goals mentioned by participants were:

- Migration of software from legacy systems into services and into new technologies without adversely affecting customers. 79% of our participants mentioned this as a goal (P2, P3, P4, P5, P7, P9, P10, P11, P13, P14, P15, P16, P17, P18, P19). For example P14 stated *‘every four or five years technology changes, new devices come on and we need to be in a position with our architecture that we can move with that without huge cost and huge impact’*.
- Scaling software to larger user bases, better performance, and big data without adversely affecting customers. 42% of our participants mentioned it as a goal (P5, P13, P12, P15, P16, P17, P18, P19). For example, P16 noted the complexity of the user-scaling issue: *‘when there is a threshold of millions of users you may (even) need to redesign problems that were not possible to foresee before….the (whole) system can be replaced’*


(a) A sub list of our Codes
(b) A hierarchical View of our Coding
List
Hierarchy

Code Name	Count
Not useful for just in time and immediate feedback to ...	6
Communication	7
costs	7
Not valuable to agile development	7
useful for team awareness for architecture knowledge	7
mapping code into architecture	8
soa support	8
testing support	8
Usefulness for quality of code base	8
architetural knowledge	9
defining an accurate architecture	9
Potential Users	9
size of systems	9
usefulness for training junior programmers	9
age of system	10
Usefulness of just in time feedback	13
auditing	14
Usefulness for architecture consistency	14
role of architect	16
code dependencies	17
Usefulness for outsourcing	17
Agile Software Development	21
Architecture discovery	22
alternative tools	24
Usefulness for Architecture Refactoring	24
important architecture problems	26
Characteristics for tool adoption	54

Code	Count	Total
▷ Agile Software Development	21	31
Architecture discovery	22	22
▷ auditing	14	48
Communication	7	7
▷ costs	7	12
important architecture problems	26	26
◢ useful for team awareness for architecture knowledge	7	20
Identifying poor programmers	1	1
Usefulness for identifying poor programmers	1	1
usefulness for training junior programmers	9	9
Usefulness of just in time feedback for naive programmers	2	2
▷ Usefulness for architecture consistency	14	55
▷ Usefulness for Architecture Refactoring	24	26

Fig. 1 Our coding using qualyzer

- Releasing different products or versions from the same code base. 42% of our participants mentioned it as a goal (P1, P12, P13, P14, P15, P16, P18, P19). This is best illustrated by a quote from P14, with respect to customer customization of their system: *'Every time we want to upgrade someone from one version to another major or even minor version in some cases we are dealing with huge conversion problems. We are dealing with the customization customers have in place... so that when we push out the next version we break all (of them).'*
- Maintenance of Software: The complexity and coupling within their current architecture was often perceived as holding them back. 32% of our participants mentioned it as an issue (P5, P10, P12, P13, P14, P19). One participant referred to his *'existing code base that is more or less a blob'* and several others referred to the implications of such complex coupling: for example, P5 and P13, noted that even senior developers had difficulty in understanding some aspects of the system and that they were *'afraid to touch the code because we change something here and break something over there'*.

4.2 AC: Current Practises

The practises reported during the interviews to support AC can be categorized into two groups: Communication Practises that communicate the architecture across the team and System Practises that aim to avoid, minimize, or detect architectural inconsistencies in the system's implementation through technical means.

4.2.1 AC: Communication Practises

Participants' perceptions of the level of architectural knowledge currently in their teams varied: five replied that the knowledge was not well-distributed at all, and three mentioned that senior software engineers are more knowledgeable about the architecture than more junior ones. In the forward direction (from architecture to implementation) this was sometimes attributed to the separation of architectural and development teams. But P1 also noted a considerable delay in development-team insights being relayed to and considered by the architectural team, a significant frustration for the development teams involved.

In terms of the practises used to make development teams more aware of the architecture, the following options were mentioned by more than one participant:

- The usage of wikis, skype, forums and sometimes training to share architecture knowledge;
- Conducting Scrum meetings. Usually they focussed on shared code, but hardly any diagrams;
- Having an architecture steering committee within the company that is aware of what each team is doing, with architects from each development team on that architecture steering committee.
- Providing heavy-weight architectural documentation to the development teams.

It should be noted that, of the communication channels between the architects and developers, the first two options mentioned are quite informal, consisting of wikis, forums and scrum meetings. In addition the communication channel is often transient (scrum meeting, skype) or persistent, but de-localized from the developers' work context (wikis, forums, documentation).

4.2.2 AC: System Practises

Despite the barriers to adoption, practitioners did mention that they attempted to minimize architectural inconsistencies by:

- Tools for ensuring rules are fulfilled in code. P1, P3, P4, P5, P9, P10 and P11 mentioned that they use SonarQube (2014) which allows the definition of non-dependencies between source code structures like packages. All of these participants proactively noted though that Sonar did not support architectural recovery and did not provide a holistic representation of the architecture for consistency.
- Using naming conventions (P2, P10, P13, P17). They mentioned that such conventions reduce inconsistencies, as developers can understand what a package or module does, determine their patterns of usage, and determine if they are stateless or not. For example, P10 said, "*It should be possible to tell what role a package plays just from its name*"
- Code reviews (P7, P11, P13, P16, P18). All apart from P11 mentioned that members of the team manually review the code of other team members for, among other things, AC. P11 mentioned that they used a software tool called Crucible (Crucible 2016) to facilitate the review.
- Built in scripting utilities for searching and exploring the code base to determine how individual parts call each other (P13, P14) or to ensure modularization. P13 said, "*it's not a big picture thing. It's really at a lower level".*
- Declaring the dependencies of each module in the code. P12 said, "*we declare what the dependencies of our module is. So if this module requires services from some others we can specify what that interface is and then there is no way someone can call something even though it is public from another module if it is not part of the interface*"
- Usage of model driven engineering code generation tools that follow the design (P13). P13 said, "*we would forward generate and then we would make changes subsequent to that. There would be a whole bunch of annotated tags in the code that you could reverse engineer back to Rational Rose's again (so you can say) hold on what's that happening there*".

Many of these system practises focus on informal approaches, manual approaches or on specific dependencies, rather than having a holistic architectural focus over the whole system. In the next section we look at the barriers practitioners perceive in arguing for a more holistic, formal AC approach to become part of their organizational culture.

4.3 Barriers for Adopting More Explicit AC Approaches

Several difficulties in adopting more formal, explicit AC approaches were noted by the participants:

1) Difficulty in quantifying architectural inconsistency effects, and thus justifying the allocation of resources to fix them, to senior management. P15 tried to estimate the costs:"*let's say 20% of development time is being spent on things that really they shouldn't have to do*

at all if you could pull back and redesign the platform, but because you can't do it you constantly have got to compensate...". Likewise, P5 mentioned that these architectural problems seem to have tangible costs such as increased time to market, demoralization of staff and a lesser ability to sell systems to potential customers.

But both were very clear that these were qualitative assertions. Likewise, P13 pointed to the same difficulty: "*I personally cannot estimate the cost… there is a cost, we have been trying to do this for a long time and we've had attempts before. It's not that easy*". P14 identified a substantial time-lag before any approximation could be made "*the value of good architecture is very hard to see... it is not something that you can build it and put it out and six months later say there's a return. It's going to be 10 years later or it is going to be 20 years later.*" However, P1 did acknowledge that, while it was difficult to estimate the costs, "*these issues are visible to senior management*" as their code is getting more expensive (in time and money) to maintain.

2) Architectural inconsistencies are not obvious to customers and this is seen as diminishing their importance. For example, P13 said "*...it should be easier for us to migrate into it (new technology with a simplified architecture) which is a perception I guess. But they (the customers) won't see it straight off. When a customer uses a screen, they don't know what the architecture is behind it.*" In support, P15 said, "*The customer cares as long as it's quick and it gives them what they want. And so then how do you justify in the short-term changes that don't impact the customer... As long as that's an issue and not easy to do, it doesn't happen in reality. Because people are under pressure to get stuff out of the door they don't come back to update that"*

However, P1, P14, P17 and P19 comments implied that customers are becoming aware of the issues: P14 said, "*...you have the suffering and pain of not being able to give them an easy way to upgrade*". Likewise P17 said, "*Customers feel architectural challenges as bugs and slower turn around on features*".

3) Reluctance in fixing architectural inconsistencies. Feeding on from the previous point, P13 stated that his organization would be reluctant to fix inconsistencies unless it impacted on the customers: "*It probably isn't ideal but it isn't going to affect the bottom line of our customers.*" But that wasn't the only reason cited for inaction. For example P14, who was from the same organization as P13, said: "*we are reluctant if code was written and it works: we don't want for somebody to come in and rewrite it into a better architecture*", suggesting that the problem is in trusting the quality of the change and its effects on the code-base.

Likewise P15 said that inconsistencies cannot be fixed because once an inconsistency is introduced, there will be other dependencies built around it, and that possible ripple effects will occur: "*the problem is when you build these dependencies it is like a rip in an old jumper you pull a little knit that you need and then you see the whole application comes with you because this depends on this and then there's a reference to one of these in here you will never use it, and then the application won't compile.*"

4) The effort required to map the system to the architecture: With respect to RM approaches specifically, practitioners noted the difficulty *of mapping code elements to architectural*

concepts as a precursor step. When the systems code structure or lexicons are not aligned to the planned architecture the mapping is viewed as a more time consuming process (P4, P8, P9, P16). In these instances, participants noted that the effort they would have to expend in architecture recovery is excessive. In addition, when there is a large code base, it is difficult to generate well-informed assumptions on how the code is structured/named with respect to that architecture. However, even when the code is strictly aligned to a planned architecture (e.g., packages are aligned with architectural components) the architect must first generate the planned architecture. Referring to this issue, P9 said, "*You still have to create the categories for grouping – i.e., the components*".

4.4 Software Development Situations where AC is Considered Useful

In this section, we present different situations, identified by practitioners, where they envision AC to be a useful tool in supporting software development. These situations are:

4.4.1 Increasing Architectural Knowledge Awareness

Fifteen participants mentioned that using an AC tool would help increasee the architectural awareness of the team and serve to train new project members or junior developers. These members could be new project leads, managers, or developers. For example, P5 noted that AC information would be: "*very useful for a new development manager or team lead... before taking over responsibility for an existing project.*" Likewise, P3 noted that it could "*train more junior developers to be more aware of the architecture design. Currently they can concentrate so much on the details of the code that they lose the higher-level context.*" P1, P13 and P19 made the point more generally, for the whole team. P1 said, "*An AC tool would help to share the architectural view with everyone in development*" and P13 concurred: *"... it just encourages the discussion".*

Referring specifically to the JITTAC tool demonstrated to participants before their interview, P14 liked the live information provided by the tool, comparing it favourably to documentation. He noted that "*it's a live picture of your architecture rather than a page somewhere that may not have been updated in six months or something like that".* He also said that it can help anyone, not just junior developers, because sometimes (more senior) developers work in different projects, or members move from one project to another.

4.4.2 Stopping Inconsistency Introduction

Closely related to architectural awareness, all participants noted that their architecture was constantly under pressure. For example, P13 noted problems caused by the scale of the software system, "*Because the code base is large, if people want to get something they just make a direct call and make direct calls to database tables when they should really be going through interfaces*". Quotes from P14 and P15 focus more on deadline pressures: "*a lot of the time a developer is working on something they are under pressure. There is a deadline they have got to get this work done and if I make that call it will work and I will get my solution out there.*"... "*probably the last thing on their mind is architectural consistency. They are just looking at just I've got get this out the door...*".

Sometimes though, the issue was more to do with lack of enforcement in the company. For example P14 said that they do not enforce internal architecture of modules*,* "*Within those modules there is also a structure which is like soap modules or packages or so on. I think... to keep those clean and have an internal structure because right now within that module we are not making any conditions about what can or cannot call anything else.* AC information was seen as a means of prompting renewed interest in enforcement and prompting developers to take the architecturally consistent, rather than the expedient, path.

4.4.3 Auditing In-House Code

All participants stated that evaluating architecture consistency through AC approaches could be usefully employed in auditing the quality of code. Several situations were identified where AC could be used for auditing the quality of in-house systems:

- To periodically determine the quality of an in-house project within the team.
- For internal contract acceptance. For example P1's organization has an analysis/design team and a development team. He noted that the analysis team could use the approach to validate what is actually delivered by the development team.
- To check the quality of releases before delivery to a client. For example, P3 said that it could be used as a quality mark, certifying a clean architecture design and consistency to that design.

4.4.4 Auditing for Outsourcing

Eleven out of the 19 participants mentioned the auditing benefit with respect to ensuring the quality of outsourced projects. For example, they noted the following outsourcing disadvantages that AC evaluations may (at least partially) address:

1. The outsourcer can lack deep experience and culture around architecture design and coding (P9).
2. P7 noted that contractors are less incentivized to produce the quality of code required to reduce long-term costs.
3. The outsourcing teams can be less aware of the relevant architecture knowledge. For example, P2 said, "*In each (internal) team, there's a good spread of knowledge of the architecture design. This usually doesn't hold true for outsourced projects*"

In this context the outsourcing client could use the approach to monitor the progress and quality of an outsourced project, and the quality of the final delivery (P5, P10). P12 said, "*if you have got your contract in and he has made changes and (they) fail the build... the contractors would be getting feedback immediately as well if they have broken the build in that fashion.*". P2 and P9 viewed an AC approach as one element in a suite of auditing tools including Sonar (SonarQube 2014), testing reports, copy-paste-detection tools, and simple build analysers.

The advantages also apply for the outsource provider. For example, P3 whose "*company, from time to time, takes on the maintenance of a client company's code base"* saw "*a value*" in

AC tooling *"in helping to judge the quality of the code (especially when an as-designed architecture has been given) or to produce an initial design when it isn't available.*"

Surprisingly, P14 noted the ability of AC tooling to reduce the amount of auditing required by customers of his company's outsourcing. He said *"if they understand that the architecture says that the change I make here has a limited impact and they can see then we don't need to re-certify everything, then that would reduce the effort on their part. If I test this piece here then I don't need to test everything else over there because it's not dependent upon it or whatever.*"

4.4.5 Evolving the Architecture

In a similar vein, 15 of the participants mentioned that once an implemented architecture is made consistent, there may be a need to support the process of moving to a new planned architecture. P14 said that AC tools could help in architecture evolution when components have to be replaced or when adding new functionality, as they help in assessing how components depended on others: "*a way we can address... what is the impact if we add a new API here or we change the nature of an API. What is the impact of that on the other modules."* Likewise P16 said, "*If you want to replace a component or a system you have to know which are the interfaces or endpoints that component or subsystem it uses.*"

Specifically, with respect to the JITTAC approach P15 said that real-time architecture feedback is essential for architectural evolution. He said, "*I think on top of that it enables architectural evolution,..., You can work away and you know that if you break anything the environment is going to let you know and that is key... and that supports I think architectural evolution during development. Without that is very difficult to achieve.*"

4.5 Features Required for AC Tool Adoption

We asked practitioners, "Would an AC approach be a "must have" or a "like to have"?" 16 out of the 19 participants stated that it was a 'like-to-have'. We then asked these 16 participants, "What are the features that are needed to make it a must have?". For the participants that answered "must have", we asked them, "What are the features that would increase AC adoption?".

In the following text, we present the features that practitioners stated were needed, in response to these 2 questions. The list is ordered based on the number of practitioners who requested the feature, as a proxy for the feature's importance. It should be noted though that 4.5.3, 4.5.4 and 4.5.6 received the same importance according to this proxy measure.

4.5.1 Expanding Beyond Source Code and to Deal with System Complexities

All participants agreed that getting dependencies from statically analysing source code only, was an issue for them. P6 said, "*The tool would have to understand all dependencies between code components*". Adding to this, P9 said, "*It's a real challenge today to understand the relationships between code modules given the number of different ways there are of introducing relationships. It's a real challenge to get an overview*". The participants expressed the kinds of dependencies that AC tools would need to support:

A. Dependency Injection in popular frameworks such as Spring, EJB, Hibernate or Struts. (P4, P5, P6, P9, P10, P11, P15, P16, P18, P19) For example, P16 said, "*some concepts cannot be mapped to packages or classes: There are for instance like other kinds of architectural frameworks that do not follow that structure.*"
B. Sometimes single systems are developed in several languages and AC tools should be able to cope with that (P1, P2, P4, P5, P13, P14, P19). P14 said, "*we would use Progress and we use C#. There may be other languages that we want to bring in or there may be ones even on top of Java like Scala or some of those so it would need to I think probably have a broad and possibly even need to interact between languages.*" The ability to interpret bytecode to be able to analyse languages built on top of Java Virtual Machine such as Scala was mentioned by several participants (P2, P9, P11, P14).
C. Web Services and RESTfull APIs (P2, P3, P6, P7, P8, P15, P16, P18, P19). Support for service oriented architecture was desired. P16 said, "*... only takes into account packages and classes. For instance, for web service architectures... that I am using this service or this or this.*"
D. Annotations (P9, P12, P13, P14, P16, P18). P12 noted their importance when he stated that "*They radically alter the behaviour of your software. They are not just documentations they enact code changes at runtime and compile time*"
E. Metadata represented in XML files (P4, P9, P12, P14, P19). For example, P14 said, "*Our API covers that but this is represented in an XML file... it's exposed for use by a client but it doesn't take the form of parameters to a function.*"
F. Dealing with Databases access from source code e.g., SQL, RDBMS (P12, P19)

4.5.2 Just-In-Time/Real Time Inconsistency Awareness

Twelve of the participants liked the idea of real-time alerts, although this may have been biased by the (JITTAC) tool chosen for the demonstration. Expressing a desire for a more real-time approach P13 said, "*The sooner you get that, the better because there is no point in discovering that after he's done a lot of work and then getting that to be reworked. So the earlier in the development process you can detect an issue then the better. It costs less.*"

Many of the participants (P3, P5, P9, P15, P16, P17, P18, P19) liked that this facility would allow them to develop the code and architecture in parallel. For example, P15 said: "*I like it to develop architectural in parallel with code because... it takes an element of communication out. It removes the need for the architect to come down and tell people that he has made a change because they see it straight away. That's great stuff.*"

Likewise in a more bottom-up perspective, P14 said that the facility would be good for integrating design and coding at the same time. He said, "*a lot of the time I do design in the code and I will try something out and I will see what it looks like.... So I would use it in design because then I can see what the impact of some change I am about to make is on something else.*"

4.5.3 Support for Fixing Inconsistencies

Some participants stated that remediation support has to be provided when inconsistencies are identified (P2, P3, P4, P5, P10, P12, P17). The participants described this support in terms of the following:

- Managing the process of fixing inconsistencies. The workflow (of fixing inconsistencies) within teams should be explicitly reflected. An example is to allow senior-developers / architects to prioritize certain inconsistencies and mark others as being ok, as an initial step (P2, P3, P4, P5, P10).
- Providing some suggestions for fixing inconsistencies e.g., actions or solutions to fixes. (P3, P4, P12). For example, P12 said: "*Just say I had a cast in my programme that didn't use generics, I would typically get a highlight in eclipse saying that this is an unsafe cast that you are making. It would not be a compiler error, it would be a warning. But when you click the little icon on the side of eclipse it says you can change it to use generics or you can add an annotation to suppress the warning. So that is useful to have an action to solve a problem. Similarly it is useful to have such features to solve architectural violations.*" P3, P8, and P12 suggested the usage of patterns and anti-patterns to be incorporated in tools to encourage and give developers guidelines on fixing inconsistencies and restructuring the code.
- Provide estimations of the costs and efforts for fixing inconsistencies. (P2)
- P17 delved more specifically into the utility of ACs for refactoring. He noted that the approach should have several hierarchical architectural models in this context: "*refactoring needs to be done between a number of models. You can play with the architecture that you define in <name-of-system> to see what is calling what on a big level. From that you can then divide up your modules, for you, think these are two completely different things right? Redesign in <name-of-system> where component goes and see how you have done."*

4.5.4 AC and System Evolution

The participants noted that architectural evolution should be supported by some estimation of the work, effort, risks, time and resultant quality required to move a system from one architecture to another (P1, P2, P3, P4, P5, P9, P10). They stated that it would be good if AC tools were equipped to provide this estimation.

- In terms of resultant quality, P1 said, "*It would be good if the tool showed some metric to monitor improvements, e.g., moved from a 4 to a 5 (or at least the system didn't get any worse)*".
- In terms of organizing the change and estimating effort/time, P2, P3 and P9 suggested a listing of the tasks required for this move, and help in prioritizing them.
- In terms of the risks, P5 suggested that the tooling should show the implications of certain choices. For example, it could show the implications of splitting a component, and then help in tracking the changes that need to be made in the code (P10). Also, the tooling should indicate which areas of the architecture are poor, explain why and indicate how to improve them (P8).
- Provide an audit of architectural changes to maintain an architecture timeline that can be browsed by architects for further effort/risk estimation (P5, P10). This could be applied to new developments and to maintenance
- Ensure that there is an agreement among architects for a change to e.g., each change must be signed off by two architects (P5).

In terms of easing the effort of evolution, as suggested in section 4.5.3, the tooling should support code refactoring (or change) to achieve consistency with an improved target architecture (Schmit et al. 2011). For example, P2, P4 and P5 suggested teasing out components that are hidden in the code, but which should be isolated to improve the design. P8 suggested the alternative of the tooling offering pattern-based advice, when moving towards target architectures.

4.5.5 Incorporating AC Into Collaborative and Versioning Repositories

Through repositories P1, P5, P9, P10, P12, P13, P16, P18 and P19 noted they could get a set of advantages: different versions of the architectural model would be available to track and could be available to all for collaboration; the user names of those who made changes and performed commits to code could be identified and the developers would not have to work with the whole code-base. P12 stated, "*If they are using SVN or GIT or CVS you have a blame functionality that will tell you the line of code checked in by the developers. At least you get the username that way.*" P16 also said, "*I would do just the change and then we should have also versions of the architecture and before committing to this central branch... the architecture is reviewed and then this update against the model in the main branch can be accepted or rejected*".

4.5.6 Incorporation into the Build Process and Continuous Integration

P15 noted, "*(you should) Get it into the continuous integration environment so that you don't have some poor unfortunate who is sitting there trying to make sure that everybody does the right thing. If they do something stupid it breaks the build...*" Likewise P12 said "*you expect the feedback to come through your build processor ANT, MAVEN or whatever you decided to use...I would use something that was part of my build process that would run every time... as part of a continuous integration.*" In total six participants suggested this requirement (P8, P9, P11, P12, P15, P16, P18, P19).

4.5.7 Defining Richer, Alternative, Architectural Models

Practitioners suggested that it was important to be able to define richer architectural descriptions, constraints or rules. Likewise, support for defining architectural styles (e.g., P10, P19) was seen as an important requirement. But probably the biggest modelling enhancement that the participants desired was to be able to define models around other quality attributes (P10, P13, P16, P18). For example P16 said, "*For me the architecture is more defined with the quality attributes than with the structural features, e.g., availability, response time.*" P13 added: "*So I mean there are other ways which you can evolve your architecture. You can change technologies, you can do things differently to have a higher performance.*"

4.5.8 The Incorporation of AC into Organization Culture and Processes

Several participants stated that AC was simply not part of their software development culture: "*Adopting an architecture consistency tool is like a habit, you either form a habit or not*" (P15). He continued, "*I think the important thing would be to get it in early. To get people get used to working with it so that they don't break the rules from an early stage.*" This lack of AC culture means that often organizations don't have a specific person who would drive this adoption: P13 said, "*the setup in the organisation here because somebody doesn't have overall responsibility for the whole thing... And we probably should have had someone with overall responsibility for dealing with issues like that to help drive that type of stuff.*"

4.5.9 AC as Free Open Source Tools

A factor that most participants mentioned is that a tool is easier to adopt if it is non-commercial. For example, P12 said, "*you do not need to convince senior managers for having free tools*".

5 Desired Features of AC and Current Tool Support

After conducting the study, and collecting the desired features from practitioners (see section 4.5), we analysed four existing commercial AC tools. Our objective is to identify whether practitioners desired features are satisfied in existing, widely used, commercial tools or if there are gaps that may cause practitioners to refrain from using them. We do not target a comprehensive analysis of the tools available in the market but instead aimed for several tools with widespread use. Hence, we believe that, even though it might be incomplete in terms of all tools, the analysis still provides valuable insights regarding the features of architecture consistency tools commonly used in practice.

For identifying relevant tools, we performed a web search via Google search for "software architecture management tools", scanned the results, and included pages describing said products according to several relevance criteria. Specifically our inclusion criteria consisted of: 1) the product description had to explicitly mention the use case of checking consistency between the implementation and the intended architecture of software systems; 2) Furthermore, we required tool vendors' product web pages to provide a (substantial) list of industrial customers, to select only tools with verifiably widespread usage in commercial software development; 3) Finally, either a full functional trial version of the tool or sufficient technical documentation regarding its features and characteristics had to be available. This was required to enable the analysis. We furthermore excluded tools specifically for enterprise architecture management and any tools that were not commercial.

Table 3 shows the four tools that were included in the list resulting from the web search and filtering. It presents their status with respect to the features derived from the interviewees' replies to the question "What are the characteristics/features that practitioners require in AC tools?" For example, for the first feature category, 'analysis beyond source code and code complexity', the table includes all the features mentioned in section 4.5.1. In addition to the explicitly mentioned desirable features, we added the set of languages supported by AC tools.

The review shows that current tools support checking of a broad variety of programming languages and extensibility in that dimension: Structure 101 and Lattix for example, explicitly mention their IDE modular structure which allows language-specific parsers to be added. Despite this broad variety, support for heterogeneous systems, consisting of source code of more than one language does not come as standard. To a certain degree, Structure 101 and Lattix could be adapted towards this scenario. Structure 101 can represent a system in a programming-language-independent abstraction, as a graph and its analysis can be performed on this graph representation. In contrast, Lattix comes with a feature for analysing UML and SysML models, e.g. UML class diagrams, component diagrams, or composite structure diagrams. If it is possible to reverse-engineer a heterogeneous system into such a model, abstracting from the specific programming languages used to implement the system, this feature of Lattix could be utilized to analyse these types of systems. However, both approaches ultimately need adaptation to integrate code written in different languages. For example, Java

Table 3 Feature list of four selected tools

	Addressed issue (see Sec 4.5.)	Structure 101 (2016)	Sonargraph (2016)	Lattix Architect (2016)	Coder Gears Product Family (Jarchitect, CppDepend, Xclarify, VBDepend) (2016)
Expanding the mapping beyond source code and to deal with source code complexities.					
Supported Languages	4.5.1	Natively: Java,. Net (C#) Third party parsers for ActionScript, Doxygen, Caché ObjectScript, Pascal, PHP, SQL, SysML, UML, Scitools Understand (Code analysis tool, supporting Ada, Cobol, C/C++, C#, Fortran, Java, jovial, Pascal, PL/M, Python VHDL)	Java, C/C++, C#	ActionScript, Ada, C/C++/ Objective-C, Delphi Pascal, Fortran, Java, JavaScript,. NET, Oracle, SQL Server, Sybase. Language-independent system/dependency specification in Excel and SysML/UML possible.	Java, C/C++, Objective-C, VB
Dependency Injection	4.5.1.A	Spring	Spring	Spring, Hibernate	No
Multiple languages per project/system supported	4.5.1.B	No	No	No	No
Web Services and Restful APIs	4.5.1.C	No	No	No	No
Meta data (annotations, configuration files,…)	4.5.1.E/D	Annotations, Spring (see DI)	Spring (see DI)	Annotations	Spring (DI)
Databases	4.5.1.F	Extension to analyse SQL	No	Oracle, SQL Server, Sybase	No
Just-in-Time/Real Time Inconsistency Awareness					
Just-in-Time Checking in IDE	4.5.2	Yes	Yes	No	No
Support for fixing inconsistencies					
Support for fixing architectural inconsistencies	4.5.3	"Recording" of refactoring in sandbox, generation of action lists	"Recording" of refactoring in sandbox	Impact analysis based on simulating refactoring	no direct support
Architecture Consistency and EvolutionSupport					
Assessing and Monitoring Code Quality	4.5.4	Two complexity metrics ("Fat" and "Tangled")	Suite of code quality metrics, can be bundled together	Suite of code quality metrics	Suite of code quality metrics

Table 3 (continued)

	Addressed issue (see Sec 4.5.)	Structure 101 (2016)	Sonargraph (2016)	Lattix Architect (2016)	Coder Gears Product Family (Jarchitect, CppDepend, Xclarify, VBDepend) (2016)
			with thresholds as target values as "quality model"		
Support for Architectural Change	4.5.4	Comparison of measurements for snapshots of the system, Change impact analysis	System snapshots can be used for historical analysis	Charts and reports for historical data	Trend charts of metrics values over time
Integration with Development Tools					
IDEs	4.5.6 &, 4.5.5	Eclipse, IntelliJ IDEA, Visual Studio	Eclipse, IntelliJ	Eclipse	Visual Studio
Build and deployment	4.5.6 & 4.5.8	Ant, Hudson/Jenkins, Maven, Sonarqube	Ant, Maven, Jenkins, JIRA, CodeBeamer	Arbitrary, via command line tools	Hudson, Jenkins, TeamCity, CruiseControl
Collaboration and versioning tools	4.5.5 & 4.5.8	Integrated repository and versioning system	Snapshot/history is managed in product specific database	Arbitrary, via command line tools	Indirect integration via CI tools mentioned above
Architecture Representation and Expressiveness					
Primary architecture representation	4.5.7	Graphical. Layers and Slices. Static dependencies-focused	Graphical. Layers and Slices. Static dependencies-focused	Graphical or matrix. Static-dependency focused.	Matrices, Static Dependencies Graphs (both for descriptive architecture only)
Definition of architectural templates	4.5.7	No	No	No	No
Definition and checking of quality attributes (beyond maintainability)	4.5.7	No	No	No	No
Licence					
Free / Open Source	4.5.9	No / No	No / No	No / No	No / No

provides the Java Native Interface (JNI) to access code written in other languages. To support this in Structure 101, a call to JNI functionality encapsulating a call to a C function would need to be distinguished from an ordinary call to a Java function and transformed into the generic graph structure differently. Other techniques, such as embedding code written in one language into code of another language, e.g. Embedded SQL, would again require different transformations.

In order to support AC of systems with dependency injection, which is common in many execution environments such as Spring or J2EE, tools need to access information beyond pure source code, e.g. configuration files specifying which dependencies need to be created at runtime. It can be observed that the Spring framework is supported in three of the tools. More generally, the need to analyse metadata is not very well addressed. Support for this is mostly limited to annotations as constructs of the supported languages (e.g., Java, C#) and does not go beyond that to analysing arbitrary technology or system-specific metadata.

Lattix and Structure 101 support the desired feature of being able to deal with databases. Modules exist in Lattix to analyse the structure of data models for a couple of different relational databases. An extension of Structure 101 also enables the analyses of SQL schema definitions. Different database concepts, such as tables, stored procedures, triggers, etc., or groups of these can be represented in the architecture views as modules and dependencies indicate relationships such as invoking stored procedures or functions accessing data fields.

None of the tools support the need for architectural analysis of web services and RESTful API's. Neither do they support technical artefacts such as WSDL specifications nor conceptual issues, such as the dynamic binding of services.

Structure 101 and Sonargraph both explicitly address the use case of developers coding and requiring just-in-time feedback regarding architectural inconsistencies (section 4.5.2). Both, similar to JITTAC demo, hook into the automatic build process of the supported IDE to inform the developer about such inconsistencies as soon as possible.

The category, "Support for fixing inconsistencies", reflects the need described in section 4.5.3. Restructuring and refactoring as means to fix inconsistencies are supported in varying degrees. Structure 101, for example, and Sonargraph in a similar way, support this somewhat by recording changes the user would like to make to resolve architecture inconsistencies and allowing exportation of action lists for the supported IDE that the developer can follow to fix the code accordingly.

All tools can be integrated with further tool support assessing code quality, e.g. with metric tool suits helping to detect badly structured code and typical bad smells. These are mainly concerned with code quality in terms of maintainability and evolution of the system (see Sec. 4.5.4). But an integration of consistency checking with support to define and check adherence to other quality attributes is missing.

The category "Integration with development tools" comprises the points in sections 4.5.6 and 4.5.5 and also touches point 4.5.8, since a smooth integration into the existing technical environment, might encourage developers and architects to use an AC tool as part of their culture. All current AC tools integrate into popular IDEs, build and continuous integration environments (CI), and collaboration and versioning tools. They either plug into IDEs directly or support importing projects created by some of the most popular IDEs. For example, in the Coder Gears product family, the C++ product comes as an IDE plugin. Integration with build and CI environments allows checking for architecture consistency as part of an automated build process and is supported by all tools. Support for versioning and collaboration is either proprietarily implemented in the tools themselves or supported as part of their integration with existing build and CI tools.

It is worth mentioning at this point that AC tools are often either directly part of a richer software analysis tool suite or can be integrated into such a tool suite. Klocwork, for example, is a general source code analysis tool suite that provides static code analysis, metrics, and reporting functionality. Structure 101 or Lattix can be added as plugins to Klocwork to add architecture checking functionality.

The features comprised in the category "Architecture representation and expressiveness", address facets expressing architectural properties beyond the graphical, static dependency-focused representation. In all existing tools, the primary representation is based on static dependencies. They all provide a graphical view, in many cases optionally complemented by matrix representations which avoid the visual clutter that graphical representation may portray for large systems. In these matrices, modules are represented as rows and columns and the number of dependencies between two modules is shown in the corresponding cell.

There is no tool support for defining architectural templates and adding elements, dependencies, and dependency constraints based on these templates or based on common architectural styles, besides grouping elements in horizontal, often-technical layers and orthogonal, often functionally-motivated slices.

Additional constraints beyond the specification of undesired dependencies can be specified in Sonargraph and the Coder Gears product family. Both tools come with built-in query languages which allow the user to formulate queries in an SQL-like syntax and retrieve matching elements or structures from the codebase. This feature can be utilized to retrieve elements that violate architectural constraints, including required structural properties.

6 Discussion & Threats to Validity

6.1 Discussion of the Findings

In this section, we will discuss several of our findings and present our Lessons Learnt (LL) that can direct further research and commercial tool development for heightening adoption of Architecture Consistency tools. Several LLs require more work by researchers and those are indicated with "(R)", others by Commercial Tool Vendors and indicated with "(T)" and others require the collaboration between AC Commercial Tool Vendors and Researchers. Those are indicated with "(RT)".

Several empirical studies have been conducted in the literature to evaluate the usage of AC tools. These studies often report on identifying the inconsistencies between the designed and implemented architecture (Knodel 2010) and the subsequent removal (or not) of inconsistencies by developers after identification (Ali et al. 2012; Buckley et al. 2015; Rosik 2014). The studies presented in (Ali et al. 2012; Buckley et al. 2015) also characterized users' architecture modelling, in terms of their mapping of code to the architecture, the types of architectural inconsistencies that they encountered and their usage of an AC tool in general, over five industrial case studies. Likewise Brunet et al. (2015) studied how developers of the Eclipse open source project used an AC tool for describing architectural rules, the kind of inconsistencies they found, and how they dealt with them. However, none of these empirical studies have explored the stated AC needs of practitioners in practice and whether AC tools meet these requirements.

When practitioners were asked about their main architectural challenges in section 4.1, they did not mention 'keeping the architecture consistent with the system' in their listing of challenges. The probable explanation for this finding is that AC is not viewed as a goal/ challenge in itself, but more as an enabling activity that supports other architectural challenges. For example, the study revealed and identified the need for AC approaches with respect to auditing development, controlling evolution and ensuring the quality attributes of a system. Likewise, most practitioners viewed AC approaches as supporting architectural knowledge awareness in a team. This aligns with the findings of Falezzi et al. (2010) with respect to the importance of architecture in general for agile participants: communication in the team, to support system reviews, to ensure system quality, and to support architecture evolution. But unfortunately, *few of the core architectural challenges noted by participants (*e.g. *configuration of new versions, migration) are explicitly supported by these existing tool offerings as presented in* Table 3. *Instead they tend to concentrate on supporting (isolating) maintenance and architectural knowledge awareness*. Interestingly, some initial work has been suggested, using AC approaches for migration (Buckley et al. 2008; Herold and Buckley 2015) and this would seem to be a rife area for future research. LL1(R): Researchers should make a clear link between architecture consistency and how it supports software activities like testing, migration, and building. In this aspect, more empirical studies, theoretical approaches and integration of tools should be considered.

The perception of practitioners is that current AC offerings only address the non-functional requirement of maintenance and not others. Maintenance is an important architectural consideration but there are several others and often these are more customer-facing (performance, availability, scaling, again noted in section 4.1). These insights seem to be echoed by the survey in (Malavolta et al. 2013) which also reports that industrial practitioners using ADLs desire enhanced capabilities to analyse the non-functional (they refer to these as "extra-functional") properties of their system, including performance, scalability and security. The survey found that one third of this analysis is performed manually. Architectural Consistency researchers should seek to expand their approaches towards non-functional properties. Also, the integration of checking consistency with support to define and check adherence to other quality attributes is missing in the commercial tools reviewed here. LL2(RT): Researchers and tool vendors should focus their efforts in supporting architectural consistency checks of all quality-driven architectural concerns.

Our study has shown that, although only one of the participants has adopted a formal AC tool in practice, all of the participants employ informal practises to keep the architecture consistent (see section 4.2). These are typically related to communicating the architecture to the team, code styling and promoting the proper use of APIs or interfaces. However, many of the communication practises are transient and several are physically delocalized from development of the system itself. The practises which are more closely tied to the system (naming conventions, code styling, code reviews) are typically manual (and thus more error prone), less formal and can be non-holistic in terms of system scope. Several of the tool offerings address these holistic, formal issues: for example, RM and DSMs (Passos et al. 2010). But organizational barriers exist for adopting more formal AC approaches and tools. These include an inability to quantify the benefits of architectural consistency to management, particularly in the short term, and the low visibility of the resultant benefits to customers.

There has been some research in the area of technical debt and specifically architectural debt in relation to the code. This has largely concentrated on using metrics such as coupling, number of defects, number of dependencies, etc. (MacCormack and Sturtevant 2016; Sarkar et al. 2009). However, these metrics are not really visible to customers. Other research has focused on measuring architectural debt based on effort in releasing features (Nord et al. 2012).

Even though authors state these measurement efforts are initial, their research direction does align with the finding from this study LL3(R): Researchers in AC should focus on providing measures of the costs and customer implications associated with having an inconsistent software architecture. While this finding is, to some degree, generic to all software reengineering efforts, only two such studies in architecture consistency have been performed (Knodel 2010; Sarkar et al. 2009) and, because these studies are (by necessity) case studies, they have low external validity. Additional focus should be on more empirical case studies that serve to broaden the evidence base with respect to the tangible effects of increased architecture consistency.

In a similar vein, *participants stated that only recovering the architecture or identifying inconsistencies based on structural dependencies in source code is not enough*. This source code issue was also observed in our previous industrial case studies (Ali et al. 2012; Buckley et al. 2015). Indeed, it seems that a fundamental requirement for adoption of AC tools by most of the practitioners in this study was an ability to model beyond one-language source code. They observed that current software technologies cross the boundaries of code (e.g., XML, meta-data, annotations, databases or configuration files) and involve different frameworks, languages and third party components. They noted that many of the architectural relationships within and across these different languages/frameworks/configuration files are not captured by the current tooling and that, therefore, many kinds of relevant inconsistencies would not be found. The fact that most AC tooling, as presented in Table 3, map architectural components into packages or files of one programming language at a time, severely limits their adoption. LL4(RT): Researchers and tool vendors should consider providing support for AC of a software system in a holistic way that encompasses multiple (possibly specialist) languages, frameworks, third party components and other artefacts (e.g., databases). This includes the modelling and mapping of software elements, and the identification of dependencies and inconsistencies that cross these different languages, frameworks and components.

With respect to RM-type tooling specifically, the participants cited the effort in mapping the code-base to the architectural model as onerous. Interestingly this was not noted in observational studies where RM was actually performed (Ali et al. 2012; Buckley et al. 2015), presumably based on the incremental approach adopted by participants in those studies where increasing proportions/granularities of the system were mapped over time. Model Driven Engineering approaches can automatically implement mapping in tools. However, in our previous studies (Ali et al. 2012; Buckley et al. 2015), practitioners thought that manual mapping heightened the semantic correctness of that mapping. Therefore, it seems a trade-off should be made between improving the correctness of mappings and the effort of mapping. LL5(R): Researchers need to conduct more studies and derive additional approaches for mapping software systems to architectural models, to reduce the effort for practitioners, while preserving the semantic correctness of those mappings.

We also have noted practitioners' stated desire for pre-defined, architectural templates for common architectural styles, or providing them with a language for defining these templates and architectural constraints. As analysed in Table 2 the current commercial tools do not provide this kind of support, even though these aspects have been widely investigated by researchers. LL6(T): Commercial tool vendors should provide support for AC to well-known architectural styles by defining architectural templates and constraints.

In addition, several other requirements were suggested as pre-requisites for adopting the technology. These include the ability to deal with web services/RESTful APIs. None of the analysed tools support the analysis of AC based on web services, RESTful API's and Web

Service Description Language specifications. Also, they do not support Service Oriented Architecture (SOA) conceptual issues, such as the dynamic binding of services that may require analysis beyond static dependency analysis. From section 4.5 it can be noticed that this is one of the important features that practitioners need in order to adopt AC tools and there is no available support based on the current tool offering presented in Table 3. There has been initial research to support dependency analysis of service based systems (Forster et al. 2013) and dynamic dependencies analysis (). Also, there is an open source project that provides a tool for analysis of SOA dependencies (SOA Dependency Analyser 2016). However, we do note that these approaches do not support architecture consistency checking. LL7(RT): Researchers' and tool vendors' efforts are needed to focus on supporting AC approaches and tool support for SOA architectural consistency checking in software analysis, conceptual architectural diagram recovery, and mapping SOA software characteristics from their dynamic and distributed dependencies.

There was a reluctance to address inconsistencies, particularly in the source code, where there was a perception that ripple effects would occur. Interestingly, all of the `participants that had used AC tools before their interviews, mentioned how they had identified inconsistencies, but had not subsequently fixed the majority of these inconsistencies. These results are in line with observations previously reported in empirical studies of RM (Rosik et al. 2008) and other kinds of consistency approaches (Brunet et al. 2015). In those studies several reasons were given for the lack of remediation that align with the findings reported here: the inconsistencies did not impact directly enough on customers or on the quality of the system, and their entanglement in their (highly coupled) code-bases suggested that remediation might lead to possible errors and ripple effects. By supporting inconsistency resolution through verified, automated approaches, the effort and error-proneness decreases, thus making resolution a stronger possibility. It should be noted that the commercial tools in Table 3 do provide some support towards fixing inconsistencies, but not automated refactoring.

A seeming paradox in the findings is that, practitioners suggested that support around fixing inconsistencies was needed for tool adoption, yet they do not perceive many of the inconsistencies as critical. A feature that current tool offerings do not provide is an indication of how critical or important an inconsistency is. This would encourage practitioners to rank and fix important inconsistencies, allowing them to justify that the time/effort they spend: LL8(RT): Researchers and tool vendors should provide ways and approaches for ranking and fixing identified architectural inconsistencies between an architectural model and an implementation, thus linking the fixing process to a quantification of the quality improvement for the software system.

A final lesson is derived with respect to supporting architectural consistency in two different temporal modes: checking the consistency periodically (e.g., during code submission) and/or by allowing developers receive warnings as they develop software. Many of the participants suggested that architectural inconsistency feedback should be real-time, so that entanglement does not build up over time, again decreasing the cost of fixing the inconsistency. We noticed that participants that applied an agile software development process as presented in Table 2 particularly liked the fact that developers could receive feedback as they were developing. They believed that, in this way, developers could make architectural decisions in parallel with development. In addition, it would allow developers to rationalize about the architecture during their day-to-day work and increase everyone's participation in evolving the system at the source-code *and* architectural levels, a finding echoed by Coplien who argued for less prescription and 'wiggle room' for developers (Coplien and Bjornvig 2010). However, this does not preclude consistency checks at the point of code submission. Indeed, it is it is in line with our findings and intuitive that, while real-time AC information should be provided, only the submit-time check should prohibit inconsistencies out-right. This allows developers experiment during development but forbids them submit inconsistent code. LL9(RT): Researchers and tool vendors should provide real time and just-in-time consistency checking approaches augmented with submit-time checks that are more prohibitive.

6.2 Threats to Validity

Important for any empirical study is an assessment of its validity. Validity refers to the degree that empirical results are meaningful (Mitchel 2004). Given the qualitative nature of the empirical study Internal Validity (Perry et al. 1997), which refers to the exclusive relationship between the independent and dependent variables, is of lesser relevance here.

However, External Validity, which refers to the degree to which the results reflect the population under study (Perry et al. 1997) and Reliability (Oates 2005), which refers to the consistency of data capture and interpretation, are more relevant to this type of study. In terms of External Validity, a relatively small number of participants were involved in this study and this limits the external validity of the findings, in terms of population "sampling". It should be noted though that this is accepted in qualitative studies where generality is ceded to a deeper, richer analysis of a smaller number of data points (Strauss and Corbin 1998). It should also be noted that all participants in this study were commercial software architects with responsibility (or an advisory capacity) for a budget, and thus were of high ecological validity with respect to the focus of this study: the adoption of AC approaches/tooling in commercial organizations. (Ecological validity is a subset of external validity which refers to the realistic nature of the data obtained (Perry et al. 1997)). Regardless, further studies should be carried out to buttress the findings presented here, providing a greater number of data points, possibly probing more quantitatively the findings generated here.

These studies would need to provide the participants with exposure to a wider range of AC approaches, to provide a wider context for their observations. In this study, participants were shown a RM-based approach to AC, based on the assertion that RM has been deemed the most appropriate approach currently available (Passos et al. 2010). However, it is possible that exposure to RM-type approaches, with only a passing description to other AC approaches, may have limited the data generated by the participants. In addition, as the review shows, many of the AC tooling requirements identified by participants here have been addressed by current offerings (tool integration, support for multiple languages). This suggests that the results reflect a lack of awareness by practitioners of the tooling available for AC, as well as deficiencies in the current tooling.

In terms of Reliability, all the interviews carried out here, bar the first one, were performed by one researcher only. While this is not an issue for the sessions that were audio-captured and transcribed, there may be a reliability issue with regard to data capture for the other sessions. Specifically, while contemporaneous notes were taken, it is possible that the interviewer missed some of the relevant observations made by the interviewee, or mis-interpreted them. A second researcher taking notes in the interview, immediately followed by discussion between the two researchers would have served to lessen this reliability concern.

With respect to the issue of 'interpretation' Reliability, a form of Construct Validity in this study, the notes and transcripts were individually analysed by individual researchers from the author list. However, this analysis/these interpretations were presented back to the group for detailed discussions, focusing on the traceability of the findings from the data and the aggregation of common themes. Consequently the interpretations were refined over a number of iterations, serving to raise confidence in both the reliability and meaning (Construct Validity) of the results.

7 Conclusion and Further Work

The findings here paint a picture of a community not totally convinced by the current AC approaches/tooling provided by the research community. Even though practitioners see potential for AC in auditing, architectural evolution and increasing architectural awareness, they noted that current tooling does not let them model their entire system (beyond the source code) and predominantly focuses on structural aspects of the architecture only, neglecting a wide range of relevant architectural concerns like scaling user-bases, scaling data sets, improved performance, and streamlining versioning and migration to services.

In addition the difficulty in quantifying the beneficial effects of architectural consistency and the perceived irrelevance of these benefits to customers also heightened participants' reluctance to adopt AC approaches. They do suggest, however, that it would help adoption if AC approaches were capable of capturing more dependency types in software systems and provide support for fixing any inconsistencies. Regardless, they felt that additional features would need to be added before AC could become common in companies' software development processes and that AC offerings would need to address other architectural concerns like porting, scaling and performance.

We plan on conducting longitudinal case studies to understand how practitioners' organizations and software systems benefit from AC tools and approaches. These longitudinal studies would look at the state of software systems, the practises of software organizations and the knowledge of software developers before and after the application of AC tools, for a duration of time. With these kinds of studies, we hope to provide evidence that assesses if applying AC tools can benefit organizations in terms of architectural awareness, decreasing the rate at which architectural inconsistencies are introduced and decreasing maintenance effort. In carrying out these studies, we also intend to derive measures for more quantitative, generalizable empirical studies on the topic.

Acknowledgements We acknowledge the anonymous participants of the study. This work has partly been funded by Enterprise Ireland under the Commercialisation Fund Programme, Grant-Ref No:CF20122719Y. Also, by University of Brighton under the Rising Star Scheme awarded to Nour Ali. This study was also supported in part by Science Foundation Ireland grant 12/IP/1351 and 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

References

Ali N, Rosik J, Buckley J (2012) Characterizing real-time reflexion-based architecture recovery: an in-vivo multi-case study. Proceedings of the 8th international ACM SIGSOFT conference on quality of software Architectures (QoSA '12). ACM, New York, pp. 23–32

Bachmann F, Bass L, Klein M, Shelton C (2005) Designing Software Architectures to Achieve Quality Attribute Requirements. IEEE Softw 152(4):153–165

Bang JY, Medvidovic N (2015) Proactive detection of higher-order software design conflicts. 12th working IEEE/IFIP conference on software Architecture (WICSA 2015), pp 155-164

Barbar MA (2009) An exploratory study of architectural practices and challenges in using agile software development approaches. In: Proceedings of the European Conference on Software Architecture, pp 81–90

Behnamghader P, Ming LD, Garcia J, Link D, Shahbazian A, Medvidovic N (2016) A large-scale study of architectural evolution in open-source software systems. Empir Softw Eng:1–48

Boehm B (2002) Get ready for Agile methods, with care. IEEE Computer 35(1):64–69

Brunet J, Bittencourt RA, Serey D, Figueiredo J (2012) On the evolutionary nature of architectural violations. Proceedings of working conference on reverse engineering (WCRE), pp 257–266

Brunet J, Murphy GC, Serey D, Figueiredo J (2015) Five years of software architecture checking: a case study of eclipse. IEEE Softw 32(5):30–36

Buchgeher G, Weinreich R (2011) Automatic tracing of decisions to architecture and implementation. In: Proceedings of the 2011 Ninth working IEEE/IFIP conference on software Architecture (WICSA '11). IEEE Computer Society, Washington, DC, pp 46–55

Buckley J, Ali N, English M, Rosik J, Herold S (2015) Real-Time Reflexion Modelling in Architecture Reconciliation. Inf Softw Technol 61:107–123

Buckley J, Le Gear A, Johnston T, Cadogan R, Looby B, Exton C, Koschke R (2008) Encapsulating targeted component abstractions using software reflexion modelling. International Journal Software Evolution and Maintenance: Research and Practice 20:107–134

Buckley J, Mooney S, Rosik J, Ali N (2013) JITTAC: a Just-In-Time Tool for Architectural Consistency. Proceedings of the 2013 international conference on software engineering (ICSE '13). IEEE Press, Piscataway, pp. 1291–1294

CoderGears CoderGears product webpage. (2016) http://www.codergears.com/home. Retrieved March 30, 2016

Coplien J, Bjornvig G (2010) Lean architecture of agile software development. Wiley, Chichester

Crucible: (2016) https://www.atlassian.com/software/crucible/overview. Last accessed 26 May 2016

De Silva L, Balasubramaniam D (2012) Controlling software architecture erosion: a survey. J Syst Softw 85(1): 132–151

De Silva L, Balasubramaniam D (2013) PANDArch: a pluggable automated non-intrusive dynamic architecture conformance checker. In: Drira K (ed) Software architecture. Lecture notes in Computer science 7957. Springer Berlin Heidelberg, pp 240–248

Ducasse S, Pollet D (2009) Software architecture reconstruction: a process-oriented taxonomy. IEEE Trans Softw Eng 35(4):573–591

Falessi D, Cantone G, Sarcia SA, Calavaro G, Subiaco P, D'Amore C (2010) Peaceful coexistence: agile developer perspectives on software architecture. IEEE Softw 2:23–25

Forster T, Keuler T, Knodel J, Becker MC (2013) Recovering component dependencies hidden by frameworks–experiences from analyzing OSGi and Qt. 17th European conference on, Genova software maintenance and reengineering (CSMR), pp 295-304

Ganesan D, Keuler T, and Nishimura, Y (2008) Architecture compliance checking at runtime: an industry experience report. In: The Eighth International Conference on Quality Software

Garlan D, Schmerl B (2004) Using architectural models at runtime: research challenges. First European Workshop on software Architecture, LNCS 3047. Springer, pp 200-205

Girard J, Koschke, R (1997) Finding components in a hierarchy of modules: a step towards architectural understanding. ICSM, pp 58–65

Haizer T, Zdun U (2012) 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

Hello2morrow Sonargraph product family webpage. (2016) https://www.hello2morrow.com/products/sonargraph. Retrieved March 30, 2016

Herold S, Buckley J (2015) Feature-oriented reflexion modelling. In: Proceedings of the 2015 European conference on software Architecture workshops (ECSAW '15). ACM, New York

Hochstein L, Lindvall M (2005) Combating architectural degeneration: a survey. Inf Softw Technol 47(10):643–656

Ihme T, Abrahamsson P (2005) The use of architectural patterns in the agile software development of mobile applications. International Conference on Agility, Helsinki

Javed MA, Zdun U (2014) A systematic literature review of traceability approaches between software architecture and source code. In: Proceedings of the 18th international conference on evaluation and assessment in software engineering (EASE '14). ACM, New York

Knodel J (2010) Sustainable structures in software implementations by live compliance checking. PhD Thesis, Fraunhofer Institute for Experimental Software Engineering

Knodel J, Muthig D, Haury U, Meier G (2008a) Architecture compliance checking-experiences from successful technology transfer to industry. Proceedings of 12th European conference on software maintenance and reengineering (CSMR 2008). IEEE, pp 43–52

Knodel J, Muthig D, Naab M, Lindvall M (2008b) Static evaluation of software architectures. Proceedings of European conference on software maintenance and reengineering, pp 279–294

Koschke R, Simon D (2003) Hierarchical reflexion models. Proceedings 10th working conference on reverse engineering, pp 36–45

Krutchen P (2010) Software Architecture and agile software development: a clash of two cultures?. 32nd international conference on software engineering, pp 497-498

Lago P, Malavolta I, Muccini H, Pelliccione P, Tang A (2015) The road ahead for architectural languages. IEEE Softw 32(1):98–105

Lattix Lattix Architect webpage. (2016) http://lattix.com/lattix-architect. Last Accessed 26 May 2016

Le Gear A, Buckley J, Collins JJ, O'Dea K (2005) Software reconnexion: understanding software using a variation on software reconnaissance and reflexion modelling. Proceedings of international symposium on empirical software engineering, pp 34-43

Lindvall M, Tesoriero, R, Costa P (2002) Avoiding architectural degeneration: an evaluation process for software architecture. In: Eighth IEEE Symposium on Software Metrics, pp 77–86

MacCormack A, Sturtevant DJ (2016) Technical debt and system architecture: the impact of coupling on defect-related activity. J Syst Softw 120:170–182

Madison J (2010) Agile Architecture Interactions. IEEE Softw 2:41–48

Malavolta I, Lago P, Muccini H, Pelliccione P, Tang A (2013) What industry needs from architectural languages: a survey. IEEE Trans Softw Eng 39(6):869–891

Mattsson A (2010) Automatic enforcement of architectural design rules. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering - volume 2 (ICSE '10), vol 2. ACM, New York, pp 369–372

Mattsson A (2012) Modelling and automatic enforcement of architectural design rules. University of Limerick

Medvidovic N, Egyed A, Gruenbacher P (2003) Stemming architectural erosion by coupling architectural discovery and recovery. Second international SofTware requirements to Architectures Workshop (STRAW'03), pp 61-68

Ming LD, Behnamghader P, Garcia J, Link D, Shahbazian A, Medvidovic N. (2015) An empirical study of architectural change in open-source software systems. Twelfth Working Conference on Mining Software Repositories, pp. 235–245

Mirakhorli M, Fakhry A, Grechko A, Wieloch M, Cleland-Huang J (2014) Archie: a tool for detecting, monitoring, and preserving architecturally significant code. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering (FSE 2014). ACM, New York, pp 39–742

Mirakhorli M, Shin Y, Cleland-Huang J, Cinar M (2012) A tactic-centric approach for automating traceability of quality concerns. In: Proceedings of the 34th international conference on software engineering (ICSE '12). IEEE Press, Piscataway, pp 639–649

Mitchel MI (2004) Research design explained, 5th edn Thomson-Wadsworth

Murphy GC, Notkin D (1997) Reengineering with Reflexion models: a case study. Computer 30(8):29–36

Murphy GC, Notkin D, Sullivan KJ (1995) Software reflexion models: bridging the gap between source and high-level models. Proc. of the 3rd ACM SIGSOFT symposium on foundations of software engineering, pp 18–23

Murta LGP, van der Hoek A, Werner CML (2008) Continuous and automated evolution of architecture-to-implementation traceability links. Autom Softw Eng 15(1):75–107

Nguyen TN, Munson EV, Thao C (2005) Object-oriented configuration management technology can improve software architectural traceability. Third ACIS Int'l conference on software engineering research, management and applications (SERA'05), pp 6-93

Nord RL, Ozkaya I, Kruchten P, Gonzalez-Rojas M (2012) In search of a metric for managing architectural technical debt, 2012 Joint working IEEE/IFIP conference on software Architecture and European conference on software Architecture, Helsinki, pp 91–100

Oates BJ (2005) Researching information systems and computing. Sage publications, London

Passos LT, Terra R, Diniz M, Valente R, Mendonça NC (2010) Static architecture-conformance checking: an illustrative overview. IEEE Softw 27(5):82–89

Perry D, Porter A, Votta LA (1997) Primer on empirical studies. Tutorial Presented at the International Conference on Software Maintenance, Boston, U.S.A., pp 657

Popescu D, Medvidovic N (2008) Ensuring architectural conformance in message-based systems. In: Workshop on architecting dependable systems (WADS)

Prechelt L (1999) The 28:1 Grant Sackman Legend is Misleading: How large is Interpersonal Variation Really?. Technical Report, 1999–18. Universitat Karlsruhe, Germany

Pruijt L, Köppe C, van der Werf JM, Brinkkemper S (2016) The accuracy of dependency analysis in static Architecture compliance checking. Software Practice and Experience 47(2):273-309

Qualyzer: (2016) http://qualyzer.bitbucket.org/. Last accessed 26 May 2016

Romano D, Pinzger M, Bouwers E (2011) Extracting dynamic dependencies between web services using vector clocks. 2011 I.E. International Conference on Service-Oriented Computing and Applications (SOCA), Irvine, CA, pp. 1–8

Rosik J (2014) A continuous approach for software architecture consistency. PhD Thesis, University of Limerick

Rosik J, Gear A, Buckley J, Babar M, Connolly D (2011) Assessing architectural drift in commercial software development: a case study. Softw Pract Exper 41(1):63–86

Rosik J, Le Gear A, Buckley J, Babar MA (2008) An industrial case study of architecture conformance. Proceedings of the second ACM-IEEE international symposium on empirical software engineering and measurement (ESEM '08). ACM, New York, pp. 80–89

Sangal N, Jordan E,Sinha V, Jackson, D (2005) Using dependency models to manage complex software architecture. Proceedings of 20th Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM Press, pp 67–176

Sarkar S, Ramachandran S, Kumar GS, Iyengar MK, Rangarajan K, Sivagnanam S (2009) Modularization of a large-scale business application: a case study. IEEE Softw 26(2):28–35

Schmit F, MacDonnel SG, Connor A (2011) An automatic architecture reconstruction and refactoring framework. Proceedings of the 9th ACIS international conference on software engineering research, management and applications (SERA2011). Sprnger, pp 95-111

Seaman C (1999) Qualitative Methods in Empirical Studies of Software Engineering. IEEE Trans Softw Eng 25(4):557–572

Sefika M, Sane A, Campbell RH (1996) Monitoring compliance of a software system with its high-level design models. Proceedings of the 18th international conference on software engineering, pp 387–396

SOA Dependency Analyser: (2016) https://www.openhub.net/p/SOA-Dependency-Analyzer. Last accessed 19 May 2016

SonarQube: (2014) http://www.sonarqube.org/. Last accessed 02 Sept 2014

Stoermer C, Rowe A, O'Brien L, Verhoef C (2006) Model-centric software architecture reconstruction. Software: Practice and Exper 36(4):333–363

Strauss A, Corbin J (1998) Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory (2nd ed.). Thousand oaks: sage publications, Inc xiii, 312

Structure 101 Structure 101 webpage. (2016) http://structure101.com. Retrieved March 30, 2016

Techbeacon Is Agile the new Norm: (2017) http://techbeacon.com/survey-agile-new-norm. Last accessed 16 Jan 2017

Tesoriero RT, Costa P, Lindvall M (2004) Evaluating software architectures. Adv Comput 61:1–43

Tran JB, Holt RC (1999) Forward and reverse repair of software architecture. Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research. IBM, pp. 12–20

Tran H, Zdun U, Dustdar S (2011) VbTrace: using view-based and model-driven development to support traceability in process-driven SOAs. Softw Syst Model 10(1):5–29

Verbaere M, Godfrey MW, Girba T (2008) Query technologies and applications for program comprehension. ICPC:285–288

West D, Grant T, Gerush M, DeSilva D (2010) Agile development: minstream adoption has changed agility. Forrester Research Inc

Yin R (2003) Case Study Research: Design and Methods. Sage Publications, Thousand Oaks

Zheng Y, Taylor RN (2011) Taming changes with 1.x-way architecture-mplementation mapping. 26th IEEE/ ACM International Conference on Automated Software Engineering (ASE), pp. 396–399

Zheng Y, Taylor RN (2003) Enhancing architecture-implementation conformance with change management and support for behavioural mapping. 34th International Conference on Software Engineering (ICSE), Zurich, pp. 628–638