Consistent Inconsistency Management: a Concern-Driven Approach

Jasper Schenkhuizen[1], Jan Martijn E.M. van der Werf[1], Slinger Jansen[1], and Lambert Caljouw[2]

[1] Department of Information and Computing Science
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{j.schenkhuizen, j.m.e.m.vanderwerf, slinger.jansen}@uu.nl
[2] Unit4
Papendorpseweg 100
3528 BJ, Utrecht, The Netherlands
lcaljouw@unit4.com

Abstract. During the development of a software system, architects deal with a large number of stakeholders, each with differing concerns. This inevitably leads to inconsistency: goals, concerns, design decisions, and models are interrelated and overlapping. Existing approaches to support inconsistency management are limited in their applicability and usefulness in day to day practice due to the presence of incomplete, informal and heterogeneous models in software architecture. This paper presents a novel process in the form of a lightweight generic method, the Concern-Driven Inconsistency Management (CDIM) method, that is designed to address limitations of different related approaches. It aims to aid architects with management of intangible inconsistency in software architecture.

1 Introduction

Inconsistency is prevalent in software development and software architecture (SA) [7]. Although inconsistency in software architecture is not necessarily a bad thing [18], undiscovered inconsistency leads to all kinds of problems [17, 20]. Inconsistency is present if two or more statements made about a system or its architecture are not jointly satisfiable [9], mutually incompatible, or conflicting [3]. Examples of inconsistency are: failure of a syntactic equivalence test, non-conformance to a standard or constraint [9], or two developers implementing a non-relational and a relational database technology for the same database, to name a few.

In SA, inconsistency has a wide range of dimensions, such as inconsistency in code, inconsistent requirements , or model inconsistency. We refer to these types of inconsistency as 'tangible' inconsistency. On the contrary, an 'intangible' inconsistency is often denominated as a *conflict*, still being undocumented or unspecified: inconsistent design decisions or concerns. In architecture, a conflict between concerns occurs if their associated design decisions are mutually incompatible, or negatively affect each other. That is, a conflict (intangible inconsistency) can potentially manifest itself as a tangible inconsistency. Thus, if design decisions are *conflicting* (intangible inconsistency),

they are mutually incompatible [3], and can lead to tangible inconsistency. This corresponds with the view to see architectural design decisions as first-class entities [14]. Adopting the definition that a software system's architecture is the set of principal design decisions about the system [3], we see that inconsistency in SA – even though it may be intangible (undocumented) at early stages – already emerges if design decisions are inconsistent or contradictory. At early stages in the architecting process, architects deal with coarse-grained models and high-level design decisions, which are usually recorded and documented using more informal notations [3]. As a result, related formal and model-checking approaches for inconsistency management (IM) are less applicable.

Traditional approaches are based on logic or model-checking. The former uses formal inference techniques to detect model inconsistency, which makes them difficult to scale. The latter disposes model-verification algorithms that are sufficiently suited to detect specific inconsistencies, but do not fit well to other kinds of inconsistency [2]. Currently, no appropriate infrastructure is available that is capable of managing a broad class of inconsistency [9].

To address the limitations of related approaches and to support the architect in the difficult process of IM, this paper proposes a simple, lightweight method, enabling the architect to systematically manage inconsistency: the Concern-Driven Inconsistency Management (CDIM) method. CDIM identifies important concerns of different stakeholders, as these are a source of inconsistency [19], and utilizes a matrix to discuss overlapping concerns to find and manage diverse types of inconsistency. CDIM consists of a 7-step cyclic process, based on the Plan Do Check Act (PDCA) cycle [11], work of Nuseibeh [20], Spanoudakis [24], and related architecture evaluation methods. The reader is referred to [23] for a detailed overview of the construction of the CDIM and its design decisions.

The remainder of this paper is structured as follows: Section 2 provides a short overview of inconsistency management in practice. The CDIM method is briefly demonstrated in Sec. 3, followed by a conclusion and directions for future work in Sec. 4.

2 Inconsistency management in SA

An important task of the software architect is inconsistency management (IM): identifying inconsistency, preserving it when acceptable, and deferring or solving it when required [5, 20]. Spanoudakis and Zisman [24] propose a framework for IM, based upon [6] and [20], consisting of six activities: (1) *detection of overlaps*, (2) *detection of inconsistencies*, (3) *diagnosis of inconsistencies*, (4) *handling of inconsistencies*, (5) *tracking of findings*, and (6) *specification of an IM policy*. A critical component in IM is identifying overlap, as it is a precondition for inconsistency [6]. Concerns have overlap when associated design decisions influence each other. Overlap emerges due to different views, assumptions, and concerns all being interrelated because they are related to the same system [17]. The presence of such interrelations introduces the potential for inconsistency [24]. Techniques that focus on *detection of overlaps* do this based on for example representation conventions, shared ontologies, or similarity analysis [24]. Techniques for *detection of inconsistencies* are logic-based approaches,

model-checking approaches, specialized model analyses, and human-centered collaborative approaches [24]. *Inconsistency diagnosis* is concerned with the identification of the source, cause and impact of an inconsistency [24]. *Handling inconsistency* is concerned with the identification and execution of the possible actions for dealing with an inconsistency. *Tracking* refers to recording important information of the inconsistency in a certain knowledge base [24].

Applicable approaches in the context of informal models are stakeholder-centric methods for inconsistency management. These involve human inspection of overlap, and human-based collaborative exploration [24] (e.g. Synoptic [4] or DealScribe [21]). These techniques assume that detection of inconsistency is the result of a collaborative inspection of several models by stakeholders [24]. Approaches like Synoptic [4] and DealScribe [21] solely focus on model inconsistency, and therefore, cannot be used for other types of inconsistency, such as inconsistent design decisions. Synoptic requires stakeholders to specify conflicts in so-called 'conflict forms' to describe conflicts that exist in models. In DealScribe, stakeholders look for 'root-requirements' in their models. Root requirements are identified for concepts present in the models, and pairwise analysis of possible interactions between root requirements results in a list of conflicting requirements. A limitation of this approach is that pairwise sequential comparison is time-consuming and labour-intensive.

Inconsistency arises inevitably due to the fact that SA is concerned with heterogeneous, multi-actor, multi-view and multi-model activities [19]. Consequently, this heterogeneity and the diverse context of software architecture causes IM to be inherently difficult [20]. In addition, a lot of architectural knowledge is contained in the heads of involved architects and developers [13]. Though IM is needed, the possibilities for managing inconsistency in software architecture are limited [17], and architects thus benefit from methods that aid in management of inconsistency.

3 Concern-Driven Inconsistency Management

As a means to address several limitations of related approaches this paper presents the CDIM method, to systematically identify and keep track of intangible inconsistency, based on concerns and perspectives. CDIM is developed using the Method Association Approach (MAA) [16] together with input from experts through semi-structured interviews. MAA takes existing methods into account to methodically assemble a new method for use in new situations [16]. IM is a part of the process of verification and validation [9]. Verification and validation of an architecture is done with the use of architectural evaluation methods (AE) methods, which is why several AE methods are used as a basis for CDIM. Many different AE methods have been developed over the past decade, and many of them have proved to be mature [1]. Due to space limitations, we refer the reader to e.g. [1] for a discussion on and comparison of various AE methods, and to [23] for a technical report documenting the design and development of the CDIM method.

The method is based on the inconsistency management process as introduced by Spanoudakis [24] and Nuseibeh [20], combining several AE methods and IM techniques with the traditional iterative phases *Plan*, *Do*, *Check*, and *Act* (PCDA) cycle [11].

Phase:	Activity:	Roles:
Plan	**Initiate:** set goal, discuss situational factors, determine scope, prepare and involve stakeholders, decide on definitions	SA
	Construct: choose perspectives, gather, define and prioritize, specify concerns, select concerns, populate matrix	SA
Do	**Identify:** workshop: identify overlaps, identify and discuss possible conflicts	SA + involved stakeholders
	Discover: use the results of previous activities to discover inconsistency in the architecture	SA
Check	**Diagnose:** diagnose inconsistency type and cause, classify inconsistency	SA + involved stakeholders
Act	**Execute:** define action plan, execute actions	SA
	Follow up: monitor impact, add results to knowledge base, follow up, plan	SA

SA = software architect

Fig. 1. This figure describes the 4 phases of the CDIM, with the 7 corresponding activities. The activities are performed in cyclic manner. Each of the activities consists of different sub steps.

Fig. 1 depicts these 4 phases. They are divided in 7 activities: each of the activities contains multiple steps. Concerns form the drivers of the CDIM. The following section motivates the use of concerns as central elements, and the subsequent section briefly describes the four phases of the CDIM.

3.1 Concerns & concern-cards

It is cost-effective to discover inconsistency early in the process [9]. One of the first elements that an architect needs to consider, are *concerns* [10]. Furthermore, concerns are the driving force of building an architecture and designing the system [14]. Concerns express the aspects that should be relevant to a software system in the eyes of a software architect [14]. According to [22] *a concern about an architecture is a requirement, an objective, an intention, or an aspiration a stakeholder has for that architecture*". Software architects benefit from inconsistency discovery and management at an early stage, as principal decisions are often hard to reverse. Conflicts in these decisions can lead to tangible inconsistency, which emphasizes the value of focusing on concerns and design decisions.

The use of templates to capture information makes methods more consistent across evaluators [12]. To ease IM, we propose the use of concern-cards, a template that enables reasoning about a concern. Such a template makes methods more consistent across evaluators [12]. Additionally, the use of cards in software development is not unusual (such as planning poker [8] used by many SCRUM teams [15]). Concern-cards are collected and kept in a concern-card desk. A concern card consists of:

1. a unique identifier,
2. a short, concise name,
3. a comprehensive definition and explanation of the concern,
4. the concern's priority,
5. related stakeholders that have an interest in the concern,

6. the perspective or category to which the concern belongs, and
7. possible associated architectural requirements, which can be used during discussion.

During the execution of the CDIM, concern-cards are used to assist the architect in collecting and understanding the different concerns, by making these explicit.

3.2 Plan Phase

The Plan phase consists of two activities *initiate* and *construct*. During *initiate* the architect develops an action plan containing the goal of the CDIM cycle, the scope of the architecture under analysis, the organization's situational factors, and which stakeholders are needed during the CDIM cycle. During *construct*, the architect selects perspectives and collects concerns from stakeholders. A perspective enables the architect to categorize the concerns gathered and to analyze the architecture from a certain angle. For each perspective, the architect collects important and relevant concerns, and documents these as concern cards. The Plan phase results in an action plan, a concern-card desk, and a prioritized matrix of concerns.

3.3 Do Phase

The Do phase consists of two activities: *identify* and *discover*. In *identify* the architect tries to identify possible conflicts through a workshop with the relevant stakeholders. Input of the workshop is the previously constructed matrix. The principal idea behind the cells of the matrix is that these provide the hotspots: areas in the architecture where concerns could possibly overlap or conflict. Multiple overlaps or conflicts may be contained in each cell, as visualized in Fig. 2. The "hotspots" are discussed by the architect and stakeholders. Their role is to aid the evaluator with deciding on how conflicts affect the architecture or possibly could affect the architecture, and which conflicts could be problematic. The outcome is a completed matrix, presenting the architect the important areas where inconsistencies may arise.

In *discover*, the architects go through the existing architecture to discover whether these are actual inconsistencies, possibly together with relevant stakeholders. The architect uses a combination of his expertise and knowledge of the system, to systematically search for important inconsistencies, using the conflicts identified in the matrix. Given the deliberate simplicity of CDIM, and the complexity of a software architecture context, the steps in this phase are inevitably one of judgment rather than quantifiable assessment. The drawback is that this approach is less explicit and more based on subjective factors as intuition and experience. The advantage is that this is easier and more flexible. The main outcome of this activity is a list of important inconsistencies in the architecture.

3.4 Check Phase

Once the matrix is completed, and possible inconsistencies in the architecture have been discovered, the architect determines the type and cause of the inconsistency, and

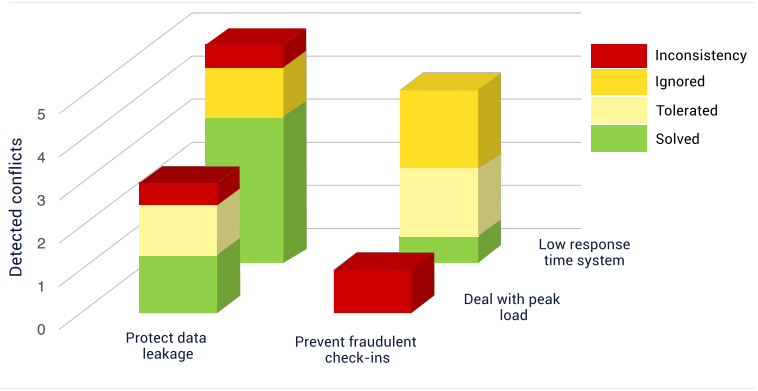

Fig. 2. A tool can be beneficial to the extent that the architect can use an overview of the amount of inconsistencies that are still open (red), that are carefully ignored (bright yellow), tolerated (cream yellow), or that have been solved (green) to indicate what needs to be done.

subsequently classifies the inconsistency. Classification is done on the basis of four aspects: *impact*, *business value*, *engineering effort*, and the individual *characteristics* of the inconsistency (such as the type and cause). *Impact* refers to an inconsistency's consequences, *business value* to whether the inconsistency is perceived as critical by the business, *engineering effort* addresses the effort of solving an inconsistency and the availability of design alternatives, and *characteristics* addresses factors related to the inconsistency itself such as the type or cause of the inconsistency. These factors are context-specific but should be considered as well [20]. The output is a *classification* in terms of these factors. Despite being still conceptual, results from the matrix could be visualized as presented in Fig. 2.

3.5 Act Phase

In the final phase of CDIM, the architect creates and executes handling actions for each inconsistency if needed (*execute*), and determines how to proceed (*follow up*). In *execute*, the architect handles discovered inconsistency based on five actions for settling inconsistencies: 'resolve', 'ignore', 'postpone', 'bypass', and 'improve'. It is important to note that handling an inconsistency is always context-specific and requires human insight. 'Resolving' the inconsistency is recommended if the impact is high, the business value is high, regardless of the engineering effort. Solving an inconsistency could be relatively simple (adding or deleting information from a description or view). However, in some cases resolving the inconsistency relies on making important design decisions (e.g. the introduction of a complete new database management technology) [6, 20]. 'Ignoring' the inconsistency is recommended if the potential impact is low, the business value is low, and the engineering effort is high. 'Postponing' the decision is recommended when both impact and business value are relatively low, and engineering effort is relatively high. Postponing provides the architect with more time to elicit further information [20]. 'Bypassing' is a strategy in which the inconsistency is circumvented

by adapting the architecture in such a way that the inconsistency itself still exists, but not touched upon. It is recommended when the current impact is low and the business value and engineering effort are relatively high, in which case continuity is important. 'Improving' on an inconsistency might be cost-effective in other situations, in which time pressure is high, and the risk is low. To improve on an inconsistency, inconsistent models can be annotated with explanations, in order to alleviate possible negative consequences of the inconsistent specification. The main outcome are the actions referring to how and when to settle the inconsistency, possibly with requests for change on the architecture, code or any other specifications that are important.

In *follow up*, the architect assesses the impact of the chosen actions, and adds gained knowledge to a knowledge base and determines how to proceed. The architect checks (a) whether a handling action intervenes with other actions; (b) whether a handling action affects existing concerns; and (c) whether a handling action results in new concerns. The architect decides whether a new cycle is needed, or iteration to previous steps is needed.

4 Conclusions & Future work

Managing inconsistencies in software architecture consistently and systematically is a difficult task. This paper presents CDIM, an inconsistency management method that aims to support software architects in managing and detecting intangible inconsistencies and conflicts in software architecture. Through concern cards, the architect can discover and document relevant concerns. Using the CDIM matrix, overlapping and conflicting concerns, and thus (undocumented) inconsistent design decisions can be detected as well as it helps architects to search for inconsistency in informal models. The method still needs careful and thorough evaluation. Initial results show that the method is simple to use, and offers the desired flexibility in combination with fast, practical results. As the method yields design solutions and concrete suggestions for architectural design, we envision the use of CDIM as a tool for communication and documenting design rationale as well. As the proof of the pudding is in the eating, we plan to build tool support to aid the architect, and evualate the method in large case studies.

References

1. M. A. Babar, L. Zhu, and R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. In *15th Australian Software Engineering Conference*, pages 309–319. IEEE Computer Society, 2004.
2. X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *30th International Conference on Software Engineering*, pages 511–520. ACM, 2008.
3. E. M. Dashofy and R. N. Taylor. *Supporting Stakeholder-driven, Multi-view Software Architecture Modeling*. PhD thesis, University of California, Irvine, 2007.
4. S. Easterbrook. Handling conflict between domain descriptions with computer-supported negotiation. *Knowledge Acquisition*, 3(3):255–289, 1991.
5. A. Finkelstein. A foolish consistency: Technical challenges in consistency management. In *Database and Expert Systems Applications*, volume 1873 of *LNCS*, pages 1–5. Springer, Berlin, 2000.

6. A. Finkelstein, G. Spanoudakis, and D. Till. Managing interference. *2nd International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development*, pages 172–174, 1996.
7. C. Ghezzi and B. Nuseibeh. Guest editorial: Introduction to the special section - managing inconsistency in software development. *IEEE Trans. Software Eng.*, 25(6):782–783, 1999.
8. J. Grenning. Planning poker or how to avoid analysis paralysis while release planning. *Hawthorn Woods: Renaissance Software Consulting*, 3, 2002.
9. S. J. I. Herzig and C. J. J. Paredis. A conceptual basis for inconsistency management in model-based systems engineering. *Procedia CIRP*, 21:52–57, 2014.
10. R. Hilliard. Chapter 10: Lessons from the Unity of Architecting. In *Software Engineering in the Systems Context*, pages 225–250. 2015.
11. C. N. N. Johnson. The benefits of PDCA. *Quality Progress*, 35(3), 2002.
12. R. Kazman, L. Bass, and M. Klein. The essential components of software architecture design and analysis. *Journal of Systems and Software*, 79(8):1207–1216, 2006.
13. P. Kruchten, P. Lago, and H. van Vliet. Building up and reasoning about architectural knowledge. In *Second International Conference on Quality of Software Architectures, QoSA 2006*, volume 4214 of *LNCS*, pages 43–58. Springer, Berlin, 2006.
14. P. Lago, P. Avgeriou, and R. Hilliard. Guest editors' introduction: Software architecture: Framing stakeholders' concerns. *IEEE Software*, 27(6):20–24, 2010.
15. Garm Lucassen, Fabiano Dalpiaz, Jan Martijn E. M. van der Werf, and Sjaak Brinkkemper. The use and effectiveness of user stories in practice. In *Requirements Engineering: Foundation for Software Quality*, volume 9619 of *LNCS*, pages 205–222. Springer, Berlin, 2016.
16. L. Luinenburg, S. Jansen, J. Souer, I. van de Weerd, and S. Brinkkemper. Designing web content management systems using the method association approach. *4th International Workshop on Model-Driven Web Engineering*, pages 106–120, 2008.
17. J. Muskens, R. J. Bril, and M. R. V. Chaudron. Generalizing consistency checking between software views. In *5th Working IEEE / IFIP Conference on Software Architecture*, pages 169–180. IEEE Computer Society, 2005.
18. C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Trans. Internet Techn.*, 2(2):151–185, 2002.
19. B. Nuseibeh. To be and not to be: On managing inconsistency in software development. In *8th International Workshop on Software Specification and Design*, page 164. IEEE Computer Society, 1996.
20. B. Nuseibeh, S. M. Easterbrook, and A. Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 58(2):171–180, 2001.
21. W. N. Robinson and S. D. Pawlowski. Managing requirements inconsistency with development goal monitors. *IEEE Trans. Software Eng.*, 25(6):816–835, 1999.
22. N. Rozanski and E. Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2012.
23. J. Schenkhuizen. Consistent Inconsistency Management: a Concern-Driven Approach. Technical report, Utrecht University, 2016. http://dspace.library.uu.nl/bitstream/handle/1874/334223/thesisv1_digitaal.pdf.
24. G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. *Handbook of software engineering and knowledge engineering*, 1:329–380, 2001.