The Process of Inconsistency Management: A framework for understanding

Bashar Nuseibeh
Department of Computing
Imperial College
180 Queen's Gate
London SW7 2BZ, UK
Email: ban@doc.ic.ac.uk

Steve Easterbrook
Department of Computer Science
University of Toronto
6 King's College Road
Toronto, Ontario M5S 3H5, Canada
Email: sme@cs.toronto.edu

Abstract

The process of managing inconsistency is a fundamental activity that permeates a large part of the systems development process. Failure to identify inconsistencies may lead to major problems in the operation of a system, while failure to support a range of inconsistency handling strategies can lead to a rigid and impractical development process. We argue, therefore, that the inconsistency management process needs to be made explicit, and should play a central role in defining the broader development process. In this paper we sketch out the constituent processes of inconsistency management in the context of an overall requirements engineering process. The framework is based on identification of an explicit set of consistency rules, which capture constraints on the evolving descriptions arising from process, method, notation, domain, etc. The set of consistency rules are refined as the development effort proceeds. The process model distinguishes four major steps: monitoring for inconsistency, diagnosis, handling, and monitoring the outcome. These are supported by the processes of measuring inconsistency, and analyzing the impact and risk associated with different inconsistency handling options. The framework provides a core process model for managing a large set of evolving descriptions during requirements engineering. Because the consistency rules are made explicit, the framework provides greater flexibility for selecting appropriate inconsistency handling actions. It also fully supports the need to adapt the inconsistency management strategy according to local contingencies in the development process.

1. Introduction

As an 'early' phase in the software development life cycle, requirements engineering (RE) involves the creation and analysis of a large number of artifacts, or *descriptions*. These descriptions reflect incomplete or partial stakeholder requirements that typically undergo many changes in the development lifetime of a software system. As a result, practitioners recognize that their descriptions are frequently inconsistent, but learn to live with these inconsistencies with a view to resolving them at some time in the future or because they judge that any adverse impact of these inconsistencies is tolerable. Unfortunately, many existing software development techniques assume consistency, and many software engineering support environments attempt to enforce it. Inconsistency is viewed as undesirable, to be avoided if at all possible. In this paper, we work on the premise that not only is inconsistency pervasive in requirements engineering, but that maintaining global consistency at all times can be counter-productive. We argue that it is often desirable to tolerate or even encourage inconsistency, to maximize design freedom, to prevent premature commitment to design decisions, and to ensure all stakeholder views are taken into account. Inconsistency also helps focus attention on problem areas, and as such may be used as a tool for directing the requirements elicitation process and as a validation and verification (V&V) tool for analysis and testing.

This position paper presents a characterization of inconsistency and then proposes a framework for managing inconsistency in this context. It draws upon our practical experiences of dealing with inconsistency in large-scale software development projects [4, 11]. For an account of related work in the area, readers are referred to [5, 6, 9].

2. What is inconsistency?

We use the term *inconsistency* to denote *any situation in which two descriptions do not obey some relationship that should hold between* [8]. Such a relationship can be expressed as a *consistency rule*, against which descriptions can be checked. The notion of logical inconsistency – a contradiction between two propositions – is subsumed within our definition, where the relationship that should hold is the absence of contradictory propositions[1].

[1] In logic, two propositions are contradictory if it is possible to derive both some fact, *X*, and its negation, *not X*.

A significant proportion of our research has focused on inconsistencies that occur during requirements engineering. Our motivation is based on our pragmatic experiences of dealing with large-scale, evolving requirements documents. Inconsistencies often arise in the requirements of multiple stakeholders and in the specifications that represent them. Inconsistencies also arise as systems evolve and new requirements are introduced. It is also our view that inconsistencies that occur during the early stages of development (e.g. during requirements elicitation) are more: (a) prevalent, (b) tolerable, and (c) in need of better management as their impact on later development stages can be critical.

3. A Framework for Managing Inconsistency

An outline of our inconsistency management framework is shown in below. Central to our approach is the use of a set of *consistency rules*, which provide a basis for most inconsistency management activities. These rules capture the set of relationships that should hold within a set of requirements descriptions. The consistency rules are derived from a number of sources, including constraints from a defined requirements engineering process, rules from specific methods about the use of requirements modelling schemes, and past experience of the application domain. As developers iterate through the inconsistency management cycle, the set of rules will be expanded and refined.

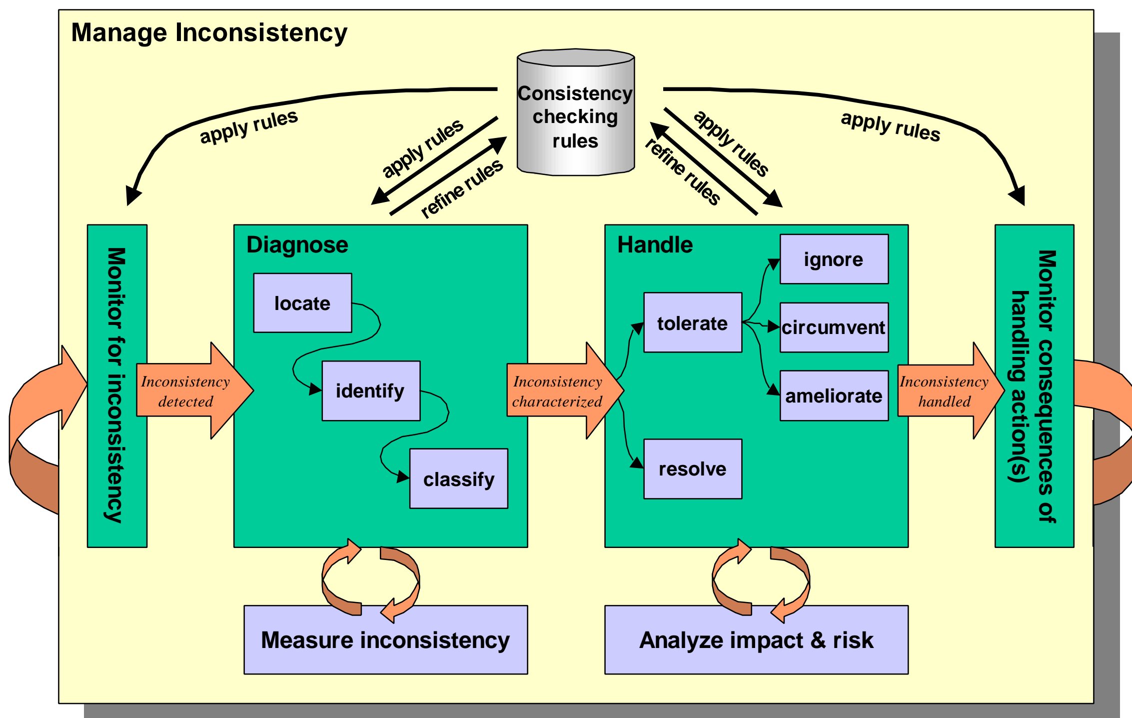
The inconsistency management process begins when inconsistencies are first detected (by *monitoring for inconsistency*). Each detected inconsistency is then *diagnosed* and *handled*. The *consequences* of the inconsistency handling actions are then monitored so that these actions can be continuously re-evaluated as development circumstances change.

Two cornerstones of inconsistency management are the activities of *measuring* (the degree of) inconsistency in software artifacts, and *analyzing* the impact and risk of performing particular inconsistency handling actions.

3.1. Monitoring for Inconsistency

Our inconsistency management process begins with the monitoring of an evolving set of descriptions for inconsistencies, according to a set of consistency rules. The key to effective management of inconsistency is the identification of consistency rules and the process of checking whether or not they hold for a particular description (specification). Consistency rules are the only means by which inconsistency can be detected (automatically). Therefore, these rules must be expressed precisely before they can be checked against the software artifacts in question. Consistency rules can arise from many sources, such as:

- from the definition of the notations used in a development process. For example, for a strongly typed programming language, the notation requires that the use of each variable be should be consistent with its declaration.
- from the development method(s) used. A method can be regarded as a set of notations, together with guidance on how these notations should be used together. This guidance is a rich source of consistency rules. For example, a method for designing distributed systems might require that for any pair of communicating subsystems, the



data items to be communicated are defined consistently in each subsystem's interface.

- from the development process model. A process model typically defines development steps, entry and exit conditions for those steps, and constraints on the products of each step. For example, a process model might require that every discrepancy report generated during testing must be counter-signed by the project manager before closure. Therefore, if a report is listed as closed, and there is no signature, there is an inconsistency.
- from local contingencies. In many cases, a consistency relationship might exist between particular instances of two objects, even though this relationship might not be predetermined by the notation, method or process model. For example, if a particular timing constraint in one requirement must be the same as the timing constraint in another requirement, this is a consistency relationship, even though it might not have been derived using any method.
- from the problem domain. Many consistency rules arise from domain-specific constraints or ontologies that are independent of the notation, method or process model used. Such rules may be specified prior to the start of development, or may be discovered, specified and refined as the development proceeds. For example, the telecommunications domain might impose constraints on the nature of a telephone call, to specify certain undesirable feature interactions. Such constraints can be specified as consistency rules to be checked during development.

Consistency rules only provide an indication of possible problems during the development of a set of descriptions. Thus, the breaking of a consistency rule may only be the manifestation of a problem elsewhere in the software description. However, at the very least, consistency rules provide an indication of the 'potential' inconsistencies in a description - that is, by specifying a rule, the developer is flagging the possibility that it may be broken, and therefore should be checked, some time during the development process. The diagnosis process described below includes the tracing back from the manifestation (i.e. a consistency rule broken) to the real problem (e.g. missing information, misunderstanding, coordination breakdown, etc).

3.2. Diagnosing Inconsistency

Like many problems in the real world, diagnosis is a pre-requisite to cure. The diagnosis process begins once an inconsistency is detected. Diagnosing inconsistency includes

- *locating the* inconsistency – that is, determining what parts of a description have broken a consistency rule (and, in logic terms, have led to a contradiction).
- *identifying the cause* of an inconsistency – normally by tracing back from the source of an inconsistency to one or more stakeholders who provided relevant information (from which the inconsistency was derived). Automation of this step relies on additional information being available for each description, including the history of edit actions performed on it, and an owner or source for each action. The process then becomes one of locating the *actions* that led to the inconsistency. In general, these are the edit actions in the history of each description that caused those descriptions to become inconsistent.
- *classifying* an inconsistency – classification is an important step toward selecting a suitable handling strategy. Inconsistencies can be classified along a number of different dimensions, including the type of rule that was broken (as listed in section 2.1); the type of action that caused the inconsistency; and the severity of the inconsistency in terms of its impact on the software artifact(s) in which it was detected.

3.3. Handling Inconsistency

The choice of what inconsistency handling strategy to follow depends on the stage of development and criticality of the artifact being produced. In other words, how an inconsistency is handled depends on when it arises and the impact it has on other aspects of the development process.

One strategy, which is generally undesirable in that it is rarely adhered to by practitioners, is inconsistency *avoidance*. This approach is strongly linked to inconsistency diagnosis in that inconsistencies need to be detected and then rejected immediately. In effect, this means that any action that leads to an inconsistency must be disallowed, requiring the consequences of such an action to be known before that action is actually taken. Determining the consequences of taking a particular course of action in this way can be very difficult.Assuming that some period of time is permitted to handle inconsistency, the strategic question then becomes "should one attempt to resolve or tolerate the existence of an inconsistency in one's specification"? The process of resolving inconsistency has received considerable attention in the literature, and can often be a difficult (and non-technical) activity. Although in some cases the act of removing an inconsistency may be a simple matter of adding or deleting information from a software

description, it often relies on resolving more fundamental conflicts between stakeholders. Such conflict resolution may require compromise by one or more of the stakeholders or else deadlocks can result.

An inconsistency handling strategy we favour in the context of requirements engineering is living with inconsistency. This is not to say that we favour allowing inconsistencies to exist in our descriptions without knowledge of their existence or impact. Rather, we believe that once an inconsistency has been characterized by the diagnosis process, it can be handled in ways other than direct resolution. Such strategies include:

- *Ignoring* inconsistency - it is sometimes the case that the effort of fixing an inconsistency is too great relative to the (low) risk that the inconsistency will result in any adverse consequences. In such cases, developers may choose to ignore the existence of the inconsistency in their descriptions - although good practice dictates that such decisions should be revisited as a project progresses or as a system evolves.

- *Circumventing* inconsistency - in some cases, what appears to be an inconsistency according to the consistency rules is not regarded as such by the software developers. This may be because the rule is wrong, or because the inconsistency represents an exception to the rule that had not been captured. In these cases, the inconsistency can be circumvented by modifying the rule, or by turning it off for a specific context (e.g. for a specific description, a specific process step, etc).

- *Deferring* inconsistency - deferring the resolution of an inconsistency may provide developers with more time to elicit further information that may facilitate the resolution process or render the inconsistency unimportant. In such cases, a useful inconsistency handling strategy might be to isolate the inconsistency (and any "polluted" parts of the descriptions in which it has been found [1]), and to continue the development until such time deemed appropriate to re-visit the inconsistency.

- *Ameliorating* inconsistency - it may be more cost-effective to 'improve' a description containing inconsistencies without necessarily resolving them all. This may include adding information to the description that alleviates some adverse effects of an inconsistency and/or resolves other inconsistencies as a side effect. In such cases, amelioration can be a useful inconsistency handling strategy in that it moves the development process in a 'desirable' direction in which inconsistencies and their adverse impact are reduced.

3.4. Measuring Inconsistency and Analyzing Impact and Risk

Clearly, the choice of inconsistency handling strategy depends the availability of measures of progress and impact, which in turn make descriptions and inconsistencies measurable; that is, represented in a way that allows analysis, reasoning and V&V. Paraphrasing Tom DeMarco "you can't control what you can't measure" [2]. Measurement is central to effective inconsistency management in a number of ways. For example:

- requirements engineers developing requirements specifications often need to know the number of inconsistencies in their specifications, and how these numbers change as they add or remove information to or from these specifications.

- requirements engineers often need to prioritize inconsistencies in different ways, for example, to identify those inconsistencies that need more urgent handling due to their adverse impact on a development project or system. Alternatively, they may use these measures to compare descriptions (specifications) to assess which description is 'more consistent' than the other.

- requirements engineers need to assess their progress in the development process, for example, by measuring their conformance to a pre-defined development standard or process model.

Often, the actions taken to handle inconsistency are very dependent on an assessment of the impact of these actions on the development project. Measuring the impact of inconsistency handling actions is therefore a key to effective action in the presence of inconsistency. An assessment of the risks involved in either leaving an inconsistency or handling it in a particular way are also crucial. A well publicized example of this is the Ariane 5 disaster [10]. One of the contributing factors for Ariane failing so spectacularly was the lack of re-evaluation of the risk involved in reusing the specifications of the earlier Ariane 4 launcher in the specifications of Ariane 5.

4. Conclusions

Managing inconsistency in requirements engineering processes and products is a broad area of research and practice. We have outlined some of the activities that are involved in inconsistency management, and highlighted inconsistency handling strategies that allow living with inconsistency. We have been working on a number of such strategies, and have developed techniques for analysis and reasoning in the presence of inconsistency [7], and the use of "ViewPoints" to capture domain-specific consistency relationships between partial requirements

descriptions. Further work is needed to investigate ways of identifying and assessing the impact of inconsistency handling actions, particularly in the context of evolving requirements specifications.The inconsistency management process we have described has a number of important features. The first of these is the explicit set of consistency rules, representing relationships between the various evolving descriptions used in a requirements engineering method. By making these rules explicit, we separate the inconsistency management strategy from the particular method and notations used. In addition, the rules themselves can be adapted and added to as the development proceeds, thus capturing important additional information and experience. As the consistency rules express constraints arising from the broader requirements process and methods, our framework supports a degree of process adaptation, by allowing the constraints of specific methods and notations to be adapted according to local contingencies.Secondly, the framework includes explicit steps for classifying the inconsistency and selecting an appropriate handling strategy. This allows greater flexibility in terms of choosing appropriate handling approach for each inconsistency, and ensures that risky decisions are explicitly captured and documented.

Thirdly, the framework is based on continuous monitoring, both for inconsistencies, and for undesired consequences of any inconsistency handling actions that have been taken in the past. This reduces the risk that major problems in an evolving set of descriptions will go undetected. It also ensures that important design decisions are continuously reassessed as the requirements evolve.

Finally, the framework can be tailored to specific risk management policies. The decision for how to handle each inconsistency is a risk-based decision, based on an assessment of the risk of leaving the inconsistency unresolved, versus the cost and residual risk of each possible handling option.

Acknowledgements. Bashar Nuseibeh gratefully acknowledges the financial support of the UK EPSRC (for projects GR/L 55964 and GR/M38582).

References

[1] R. Balzer (1991), "Tolerating Inconsistency", *Proceedings of 13th IEEE International Conference on Software Engineering (ICSE-13)*, 158-165, Austin, Texas, USA, 13-17th May 1991.

[2] T. De Marco, *Why Does Software Cost So Much?*, Dorset House, 1995.

[3] S. Easterbrook and B. Nuseibeh, "Using ViewPoints for Inconsistency Management", *BCS/IEE Software Engineering Journal,* January 1996.

[4] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo and D. Hamilton, "Experiences Using Lightweight Formal Methods for Requirements Modeling", *IEEE Transactions on Software Engineering,* 24(1), January 1998.

[5] C. Ghezzi and B. Nuseibeh (Eds.), Special issue on "Managing Inconsistency in Software Development", *IEEE Transactions on Software Engineering,* November 1998.

[6] C. Ghezzi and B. Nuseibeh (Eds.), Forthcoming special issue on "Managing Inconsistency in Software Development", *IEEE Transactions on Software Engineering,* November 1999.

[7] A. Hunter & B. Nuseibeh, "Managing Inconsistent Specifications: Reasoning, Analysis and Action", *ACM Transactions on Software Engineering and Methodology,* October 1998.

[8] B. Nuseibeh, J. Kramer and A. Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification", *IEEE Transactions on Software Engineering,* 20(10): 760-773, October 1994.

[9] B. Nuseibeh, "To Be and Not To Be: On Managing Inconsistency in Software Development", *Proceedings of 8th IEEE International Workshop on Software Specification and Design (IWSSD-8),* 164-169, Schloss Velen, Germany, 22-23 March 1996.

[10] B. Nuseibeh, "Ariane 5: who dunnit?", *IEEE Software,* 14(3): 15-16, May 1997.

[11] A. Russo, B. Nuseibeh, and Jeff Kramer, "Restructuring Requirements Specifications", (to appear in) *IEE Proceedings: Software Engineering,* 1999.