ELSEVIER

The Journal of Systems and Software 58 (2001) 171–180

The Journal of Systems and Software

www.elsevier.com/locate/jss

Making inconsistency respectable in software development ☆

Bashar Nuseibeh [a,*], Steve Easterbrook [b], Alessandra Russo [c]

[a] Computing Department, The Open University, Walton Hall, Milton Keynes MK7 6AA, UK

[b] Department of Computer Science, University of Toronto, 6 King's College Road, Toronto, Ont., Canada M5S 3H5

[c] Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK

Received 21 August 2000; accepted 8 September 2000

Abstract

The development of software systems inevitably involves the detection and handling of inconsistencies. These inconsistencies can arise in system requirements, design specifications and, quite often, in the descriptions that form the final implemented software product. A large proportion of software engineering research has been devoted to consistency maintenance, or geared towards eradicating inconsistencies as soon as they are detected. Software practitioners, on the other hand, live with inconsistency as a matter of course. Depending on the nature of an inconsistency, its causes and its impact, they sometimes choose to tolerate its presence, rather than resolve it immediately, if at all. This paper argues for "making inconsistency respectable" [A phrase first used by D. Gabbay and A. Hunter (in: Proceedings of Fundamentals of Artificial Intelligence Research'91, Springer, Berlin, p. 19; in: Symbolic and Quantitative Approaches to Reasoning and Uncertainty, Lecture Notes in Computer Science, Springer, Berlin, 1992, p. 129) to describe the same sentiments that motivated our work.] – sometimes avoided or ignored, but more often used as a focus for learning and as a trigger for further (constructive) development actions. The paper presents a characterization of inconsistency in software development and a framework for managing it in this context. It draws upon practical experiences of dealing with inconsistency in large-scale software development projects and relates some lessons learned from these experiences. © 2001 Elsevier Science Inc. All rights reserved.

Keywords: Software specification; Requirements engineering; Inconsistency management; Inconsistency handling; Conflict

1. Introduction

Software Engineering has been described as a discipline of description (Jackson, 1995). Software engineers make use of a large number of different descriptions throughout the development process, including analysis models, specifications, designs, program code, user guides, test plans, change requests, style guides, schedules and process models. These descriptions are constructed and updated by different developers at different times during development. Establishing and maintaining consistency between these descriptions is a difficult problem for a number of reasons:

- The descriptions vary greatly in their formality and precision. Sometimes sketchy or imprecise descriptions are used just to gain an understanding; at other times, detailed and precise descriptions are crucial to ensure correctness, safety, security, etc.
- Individual descriptions may themselves be ill-formed or self-contradictory.
- The descriptions will continue to evolve throughout the software development lifecycle, at different rates.
- Checking consistency of a large, arbitrary set of descriptions is computationally expensive. For example, imagine we could translate all the descriptions into a formal predicate logic. The problem of determining consistency of a set of logic formulae is NP-hard – in practice, this means that as our sets of descriptions grow, it very quickly becomes infeasible to test their consistency. Furthermore, incremental or localized consistency strategies will not guarantee global consistency (see Appendix A). In practice, it may be possible to find fast consistency checking techniques for specific types of description, but in the general case, the problem is intractable.

Existing approaches to this problem have been ad hoc, or have only addressed a limited part of the software development lifecycle. For example, tools exist to check

☆ This paper is a revised and expanded version of the paper that appeared in Nuseibeh et al. (2000).

* Corresponding author.

E-mail addresses: B.A.Nuseibeh@open.ac.uk (B. Nuseibeh), sme@cs.toronto.edu (S. Easterbrook), ar3@doc.ic.ac.uk (A. Russo).

0164-1212/01/$ - see front matter © 2001 Elsevier Science Inc. All rights reserved.
PII: S0164-1212(01)00036-X

the consistency of specific documents (e.g., that a set of object models are consistent), but not for testing consistency between these and other software development artifacts. Many existing software development techniques assume consistency, and many software development environments attempt to enforce it. Inconsistency is viewed as undesirable, to be avoided if at all possible.

However, inconsistency is a pervasive problem throughout the software development lifecycle. Practitioners recognize that their descriptions are frequently inconsistent, and they 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. We have found that a systematic approach to handling inconsistency is helpful, and we view inconsistency management as a central activity throughout software development. We argue that maintaining consistency at all times is counterproductive. In many cases, it may be desirable to tolerate or even encourage inconsistency, to facilitate distributed collaborative working, 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 learning, for directing the requirements elicitation process, and as a validation and verification (V&V) tool for analysis and testing.

This paper presents a characterization of inconsistency in software development and a framework for managing it in this context. It draws upon our practical experiences of dealing with inconsistency in large-scale software development projects and relates some lessons learned from these experiences.

2. What is inconsistency?

We use the term inconsistency to denote *any situation in which two descriptions do not obey some relationship that is prescribed to hold between them* (Nuseibeh et al., 1994). A precondition for inconsistency is that the descriptions in question have some area of *overlap* (Spanoudakis et al., 1999). A relationship between descriptions can be expressed as a *consistency rule*, against which descriptions can be checked. In current practice, some such consistency rules are captured in various project documents, others are embedded in tools, and some are not captured anywhere.

Table 1 gives some example consistency rules, expressed in natural language. Rule 1 is a simple rule for two descriptions written in the same notation (in this case DFDs). Rule 2 is a consistency rule between three different documents; the area of overlap is the use of the terms "user" and "borrower". Rule 3 is an example of a rule about the development process. If program code has been entered before the sign-off has occurred, then the project is inconsistent with a process policy (which, presumably, is documented somewhere). Note that in this last example, the area of overlap concerns the status, rather than the contents, of the descriptions.

Rules 2 and 3 reflect a common pattern: a consistency relationship exists between two descriptions because some third description says it should. Problems occur if changes are made to one of the three descriptions without cross checking the others, or if cases the third description is not properly documented. The move towards better modeling of software processes has helped to ensure that more such relationships are documented, but note that not all such relationships are process oriented (e.g. Rule 2).

This definition of inconsistency is deliberately broad: it encompasses many types of inconsistency that occur in software development. In particular, the notion of logical inconsistency is subsumed within our definition, where the relationship that should hold is that it should not be possible to derive a contradiction from a set of propositions. [1] Defining inconsistency in this way provides some flexibility, as it does not tie us to any particular notation and allows us to consider many different forms of inconsistency throughout the development process. So, for example, checking descriptions against consistency rules may reveal stakeholder *conflicts* (Robinson, 1990), *divergent goals* (van Lamsweerde and Letier, 1998), *faults* in a running system (Littlewood, 1994), and *deviations* from documented development processes (Cugola et al., 1996).

3. Tolerating inconsistency

Inconsistency is a problem in software engineering if it leads to misunderstandings and errors. However, the problem is not with inconsistency per se, but with inconsistency that remains undetected. In many cases, we may wish to tolerate a known inconsistency. For example, a change to a code module may violate assumptions made in modules with which it interacts, cause several test scripts to be re-written because they were based on the structure of the module, and lead to a change in the user manuals. Inconsistency arises here because a change has been made while some of the consequences have not been addressed. Because it is not feasible to carry out a change and all the consequent alterations as a single atomic action, we have to accept that there will be times during development when the set of software descriptions is inconsistent. We would expect such inconsistencies to be temporary: they reflect a

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

Table 1
Informal examples of consistency rules

Example consistency rules	
Rule 1	In a data flow diagram, if a process is decomposed in a separate diagram, then the input flows into the parent process must be the same as the input flows into child data flow diagram
Rule 2	For a particular Library System, the concept of operations document states that "User" and "Borrower" are synonyms. Hence, the list of user actions described in the help manuals must correspond to the list of borrower actions in the requirements specification
Rule 3	Coding should not begin until the Systems Requirement Specification has been signed off by the Project Review Board. Hence, the program code repository should be empty until the SRS has the status "approved by PRB"

period of uncertainty, as a document evolves. In such cases, we would prefer an environment that permits the inconsistency and tracks it for us, rather than an environment that prohibits it.

There may be times during development when tolerating inconsistency is beneficial:

- Inconsistency may indicate deviations from a process model. The rationale for process modelling is that it facilitates process improvement. Cugola et al. (1996) argues that this can be achieved by allowing deviations from the prescribed process, and by providing support for dealing with the resulting inconsistencies.
- Inconsistency may facilitate flexible collaborative working. Schwanke and Kaiser (1988) consider the problem of incremental development in the presence of inconsistency. They argue that attempting to enforce total consistency can be difficult, and it is therefore preferable to allow inconsistencies to occur, and to resolve them periodically rather than prevent them. Narayanaswamy and Goldman (1992) also present an incremental inconsistency handling solution based on announcing and interleaving "proposed changes", while Balzer introduces the notion of "pollution markers" to flag portions of program code that contain inconsistencies which can be circumvented in order to continue development.
- Inconsistency can be used to identify areas of uncertainty. If a team of developers work together on a set of descriptions, then inconsistencies in those descriptions can indicate areas where the developers' shared understanding has broken down. Easterbrook (1996) describes how these cases can be used to improve understanding within a team. Gabbay and Hunter (1991, 1992) and Finkelstein et al. (1994) also suggest that inconsistency can be used to trigger further development actions, while Hunter and Nuseibeh (1998) describe a "Quasi-Classical" logic that allows reasoning in the presence of inconsistency.

The view that inconsistency is prevalent in and a driver of software development is the subject of increasing research in the software engineering community (Ghezzi and Nuseibeh, 1998, 1999).

4. A framework for managing inconsistency

To clarify our understanding of inconsistency management research and practice, we have developed the framework shown in Fig. 1. Central to this framework is the explicit use of a set of consistency rules, which provide a basis for most inconsistency management activities. The consistency rules are used to monitor an

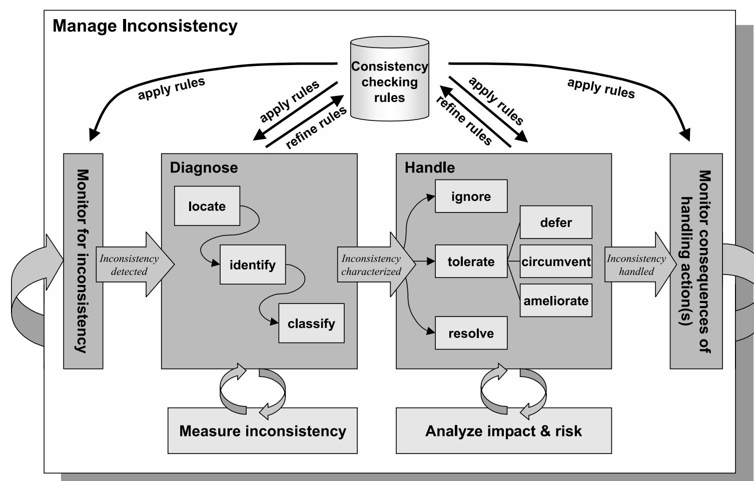

Fig. 1. A framework for managing inconsistency.

evolving set of descriptions for inconsistencies. When inconsistencies are detected, some diagnosis is performed to locate and identify their cause. One of a number of different inconsistency handling strategies can then be chosen, including resolving it immediately, ignoring it completely, or tolerating it for a while. Whatever action is chosen, the result is monitored for undesirable consequences.

4.1. Consistency checking rules

As developers iterate through the consistency management process, the set of consistency rules is expanded and refined. Hence, we do not expect to ever obtain a complete set of rules covering all possible consistency relationships in a large project. Rather, we regard the rulebase as a repository for recording those rules that are known or discovered, so that they can be tracked appropriately.

Consistency rules can arise from:

- *The definitions of notations*. For example, for a strongly typed programming language, the notation requires that the use of each variable should be consistent with its declaration.
- *The development method(s)*. A method provides a set of notations, together with guidance on how to use them. This guidance includes many 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.
- *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. Rule 3 in Table 1 is an example.
- *Local contingencies*. Sometimes, a specific consistency relationship occurs between specific descriptions, even though the notation, method or process model does not predetermine this relationship. For example, if a particular timeout in one specification must be greater than a timing constraint in another specification, this needs to be recorded as a consistency relationship.
- *The application domain*. Many consistency rules arise from domain-specific constraints. 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.

4.2. Monitoring for inconsistency

With an explicit set of consistency rules, monitoring can be automatic and unobtrusive. If certain rules have a high computational overhead for checking, the monitoring need not be continuous; but instead can be checked at specific points during development, using a lazy consistency strategy such as that outlined in Narayanaswamy and Goldman, 1992. In either case, our approach is to define a scope for each rule, so that each edit action need only be checked against those rules that include in their scope the locus of the edit action.

4.3. Diagnosing inconsistency

There may be many reasons why a particular consistency rule is broken. The diagnosis process begins whenever an inconsistency is detected. Diagnosis includes:

- *Locating* the inconsistency – that is, determining what parts of a description have broken a consistency rule.
- *Identifying the cause* of an inconsistency – normally by tracing back from the manifestation (i.e. a consistency rule broken) to the cause (e.g., missing information, misunderstanding, coordination breakdown, etc). If the history of all edit actions is available, with an owner or source for each action, this becomes a process of identifying the sequence of the actions that led to the inconsistency.
- *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, the type of action that caused the inconsistency, and the impact of the inconsistency.

4.4. Handling inconsistency

The choice of an inconsistency handling strategy depends on the context and the impact it has on other aspects of the development process. Resolving the inconsistency may be as simple as adding or deleting information from a software description. However, it often relies on resolving fundamental conflicts, or taking important design decisions. In such cases, immediate resolution is not the best option, and a number of choices are available:

- *Ignore* – it is sometimes the case that the effort of fixing an inconsistency is too great relative to the (low) risk that the inconsistency will have any adverse consequences. In such cases, developers may choose to ignore the existence of the inconsistency in their descriptions. Good practice dictates that such decisions should be revisited as a project progresses or as a system evolves.
- *Defer* – this may provide developers with more time to elicit further information to facilitate resolution or to render the inconsistency unimportant. In such cases, it is important to flag the parts of the descrip-

tions that are affected, as development will continue while the inconsistency is tolerated.

- *Circumvent* – 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 disabling it for a specific context.
- *Ameliorate* – 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.

4.5. Measuring inconsistency and analyzing impact and risk

Measurement is central to effective inconsistency management in a number of ways. For example:

- Developers often need to know the number and severity of inconsistencies in their descriptions, and how these numbers are affected by various changes they make. For example, they may use these measures to compare descriptions to assess which choice is "more consistent".
- Developers often need to prioritize inconsistencies in different ways, for example, to identify those inconsistencies that need urgent attention.
- Developers may need to assess their progress, for example, by measuring their conformance to a predefined development standard or process model.

Often, the actions taken to handle inconsistency depend 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 is also crucial.

5. Inconsistency management in practice

In our research into inconsistency, we have performed a number of case studies of NASA software development projects. These have helped us to refine the framework described above, and have provided some insights into the nature of inconsistency and its management. The first two studies concern the requirements specifications of parts of the International Space Station (ISS) Command & Control software (Easterbrook and Callahan, 1997; Russo et al., 1998). The third case study concerns the design of a dual redundant controller for a deep space probe (Schneider et al., 1998). All three case studies were based on analysis of existing, evolving, specifications expressed in a mixture of prose, tables, flowcharts and other diagrams.

While each case study applied different techniques for analyzing the specifications, our approach in each case was to re-represent and re-structure the specifications more precisely, more formally and at different levels of abstraction, in order to permit more detailed analysis than would otherwise have been possible. In the first study, we used two formal notations, SCR (Heitmeyer et al., 1996) and PROMELA (Holzmann, 1997) for verifying portions of the Fault Detection, Isolation & Recovery (FDIR) requirements for the control of a communications bus. This study demonstrated that translating the informal specifications into a formal notation helps identify ambiguities and inconsistencies, and allows some of the analysis to be automated. The second study also concentrated on the FDIR requirements for the space station. Fragments of the original FDIR specification were restructured using "viewpoints" (Nuseibeh et al., 1994), revealing both explicit and implicit relationships between these fragments. The third case study concentrated on the verification of an existing design for a mark and rollback scheme that allows a "hot backup" processor to take control of the spacecraft at an appropriate point in a sequence of operations when a fault occurs in the main processor. The study addressed the question of whether the proposed design met the requirements for fault tolerance. The design was modeled as a state machine in PROMELA. The fault tolerance requirements were expressed as temporal logic formulae, and tested against the model using the SPIN model checker (Holzmann, 1997). The analysis revealed three potential errors, which were then turned into test cases to check whether they occurred in the implementation.

The three case studies provided us with a number of insights that we discuss below.

5.1. Some inconsistencies never get fixed

This observation seems counter-intuitive at first. Although we have argued that inconsistencies can and should be tolerated during the process of developing specifications, we had always assumed that inconsistencies are temporary. For example, we assumed that eventually a consistent specification would be needed as a basis for an implementation. In practice, this is not true. Many local factors affect how an inconsistency is handled, including the cost of resolving an inconsistency, the cost of updating the documentation, and the shared understanding of the developers. Ultimately, the decision to repair an inconsistency is risk based. If

the cost of fixing it outweighs the risk of ignoring it, then it does not make sense to fix it.

In our first case study, one section of the specification contained a flowchart and some corresponding textual requirements. The flowchart was intended as a graphical representation of the text, but as the specification had evolved, the diagram and text had diverged. Due to the cost of updating the documents, the developers chose just to ameliorate this inconsistency by adding a disclaimer that the text should be regarded as definitive whenever it is inconsistent with the diagram. Despite the inconsistency, the diagram was still useful as it provided an overview of the requirements expressed in the text.

It is worth examining the risk analysis underlying the decision to leave this inconsistency unresolved. The document in question was a large (~300 pages) baselined specification. Each new version of the document is passed through a formal technical review. The participants in the review write an issue report for each problem found in the document. In a typical review, several hundred issues may be raised. A review panel examines the issue reports, and decides which ones need to be addressed. Addressing all the issues would be prohibitively expensive, so a prioritization is performed. Some issues are rejected, while for others, quick fixes are accepted. The example above is typical of a quick fix: it reduces the risk without eliminating it, and allows the issue to be closed immediately, without requiring it to be tracked and re-reviewed. There is a small risk that the document will be misunderstood if someone looks at the diagram and does not realize it is inconsistent with the text, but this risk is relatively minor in relation to other issues arising from the review. One of the incentives for our work on tolerating inconsistency is to develop techniques that reduce this residual risk by clearly flagging unresolved inconsistencies when they are detected, but without imposing any additional documentation overhead.

A second example concerns the analysis models we abstracted from the original specifications. Typically, these were state machine models, where the behavior of the model captured the behaviors described in the original specification. Sometimes, our analysis models were inconsistent with either the specification or the implementation, because they did not cover the same set of behaviors. Despite these inconsistencies, however, the analysis models were still extremely useful, as partial checks of key properties were still possible. In some cases the inconsistency could not be fixed because the formal notation would not capture certain aspects of the specifications. In other cases, fixing the formal model would introduce complexities that could interfere with the analysis.

In both these cases, the inconsistency did not need to be resolved; it was sufficient just to be aware of its existence. In each case, the decision to ignore the inconsistency was based on a careful analysis of the risk involved. If the inconsistency was not detected, no such risk analysis can be performed. Undetected inconsistencies can be dangerous, but some inconsistencies can be safely ignored *after* they have been detected.

5.2. Living with inconsistency means continuously re-evaluating the risk

The decision to tolerate an inconsistency is a risk-based decision. Because the risk factors change during the development process, it is necessary to re-evaluate the risk periodically. Ideally, this is done by monitoring each unresolved inconsistency for changes in the factors that affect the decision to tolerate it. In practice, such monitoring is not feasible with current tools. Hence, the usual approach is to identify key points in the future at which the decision needs to be re-evaluated.

As an example, consider the Ariane-5 disaster (Nuseibeh, 1997). Ariane-5 reused much of the software from Ariane-4. In Ariane-4 an inconsistency had been tolerated between the safety requirement that all exceptions be handled, and the implementation in which floating-point exception handling was turned off for some exceptions to meet memory and performance requirements. In Ariane-4, the risk of tolerating this inconsistency was thoroughly analyzed. The analysis concluded, correctly, that the floating-point overflow would never occur, and so the inconsistency could be tolerated. Ariane-5 experienced a larger horizontal bias because its trajectory differs from that of Ariane-4, such that the floating-point overflow did occur. Unfortunately, the risk analysis was never repeated when the software was reused in Ariane-5. The problem was not that the decision was wrong for Ariane-4, but that there were no tools available to indicate that the risk needed to be re-evaluated for Ariane-5.

We observed a similar example in our third case study. There was an inconsistency concerning the behavior of the two processors on the spacecraft at the end of a critical sequence. To allow for hardware delays, the fault processing logic requires a three-second delay before an operation is considered successfully completed. If a fault occurs within three seconds of the end of the critical sequence, the main processor will rollback and repeat the last section of the sequence. The backup processor should do likewise, but has already suspended itself. However, this inconsistency may not matter, depending on how the critical sequences are written. For example, if the sequences are written in such a way as to include a delay at the end, then the problem disappears. When we performed the analysis, none of the critical sequences had been designed. Hence, our recommendation was to defer resolution until the critical sequences were written. At this point the risk analysis would have to be repeated.

Knowing when and how to re-evaluate these decisions is critical. In the Ariane-5 example, the developers did not have a method to warn them to revisit decisions when the design parameters changed. In our spacecraft example, the need to revisit the decision was explicitly recorded.

5.3. Some consistency checks are not worth performing

We argued above that the act of resolving an inconsistency has a cost associated with it, and that it might not always be worth doing. An observation from the case studies was that the *application* of a consistency check also has a cost associated with it, and that it might not always be worth performing.

For example, in the first case study, we discovered an error with the sequencing of the fault diagnosis steps in the original specification. The need to apply the steps in a specific order had not been described in the text, and without this sequencing they would not work as intended. We discovered this problem while building a formal model, which we had planned to use to check that the fault handling design was consistent with the high level requirements. We made some assumptions about the correct sequencing and continued to build the model and perform further consistency checks. In the meantime, the authors of the original specification updated it to correct the problem. Their changes were so major that none of the consistency checking we had performed on our model was relevant anymore. Hence, the effort we expended to perform these checks was wasted.

This observation raises an important question: how do you know *when* to apply each consistency check, and how do you know when to stop checking consistency? The answers to these questions are often problem-specific, but they may also lie in project process models. As part of our research into guiding the inconsistency management process, we have examined the conditions under which consistency checking should and should not be performed, and the mechanisms for guiding this process (Leonhardt et al., 1995).

5.4. Inconsistency is deniable

Our framework relies on a well-defined set of relationships between descriptions. As long as these relationships are precisely defined, determination of consistency is an objective process. However, we have found that in practice, developers often debate whether a reported inconsistency really was an inconsistency. Example reactions to reported inconsistencies in our case studies included: “that’s not inconsistent because you’ve assumed...”, “that inconsistency doesn’t matter because...”; “oh, your model’s wrong...”; or, quite often “yes, we already fixed that...”.

There are two factors at work here. The first is a face saving device. People generally do not like other people finding fault with their work. The V&V teams we worked with at NASA strive to maintain a collaborative relationship with the developers, such that both parties feel they are working towards the common goal of a high quality product. However, inconsistency still carries a stigma, implying poor quality work. If an inconsistency is reported in a public manner (e.g., at a formal review), there is a tendency for authors to become defensive. We observed two common face saving devices: the author may give an argument for why the inconsistency is not really an issue, or the author might claim that he or she is already aware of the problem and has fixed it (or is in the process of fixing it).

The second factor is a modelling issue. It is possible for descriptions to be inconsistent because one or more of them is inaccurate or vague. Although we can formalize a description such that we can say objectively whether it is inconsistent at the syntactic and semantic levels, it is often possible to deny the inconsistency at the pragmatic level. In effect, such a denial questions the formalization of either the description itself, or the consistency rules. This sometimes results in a useful discussion of the nature of the descriptions, which may in turn lead to an improvement in how the descriptions (models) are expressed. On the other hand, such denials are sometimes merely obfuscation, and it is often hard to tell whether the ensuing debate will lead to anything useful.

6. Conclusions

Inconsistency arises throughout the software development lifecycle, as various development documents evolve. It arises because of interdependencies within and between documents. Whilst general principles of modularity and information hiding help to reduce such dependencies, they do not eliminate them. Moreover, any single change to a document may affect many different parts of a project, and it can be difficult to identify all the dependencies in that project. Some such dependencies are captured in traceability matrices and interface documents; however, these typically capture the existence of a relationship only, without its semantic content. Hence, reasoning about the impact of a change still has to be done by hand, and can be tedious and error prone.

In this paper, we have argued that the problems of establishing and maintaining consistency are endemic to software development. We argued that it is not always possible to avoid inconsistency, and that tools that tolerate and manage inconsistency provide more flexibility. We also argued that it is *undetected* inconsistency that

causes the most problems. Known inconsistencies can be tolerated, provided they are managed carefully.

Determining whether a set of descriptions is inconsistent depends on knowing what relationships should hold between them. We capture these relationships as a set of well-defined consistency rules, and use these as the basis for tracking both resolved and unresolved inconsistencies. There are a number of tools available that detect inconsistency in different phases of software development. In general, each tool concentrates on one particular type of description, and defines consistency narrowly in terms of integrity rules for that type of description. For example, the formal specification tool SCR has a built in set of consistency checks that establish the well-formedness of an SCR specification, according to the semantics of SCR. Such method-specific consistency checking is extremely useful, but covers only a fraction of the range of consistency relationships that are of interest during software development. So, as well as checking the internal consistency of an SCR specification, one might also wish to check the consistency between the SCR specification and a design model, or between the SCR specification and a set of test cases. Further work is needed to develop our framework into a software development environment in which inconsistency management becomes a core activity throughout development.

Acknowledgements

We would like to thank our colleagues Frank Schneider, John Hinkle, Dan McCaugherty and Chuck Neppach, who all worked on the case studies. We would also like to thank the participants at the ICSE-97 workshop on "Living with Inconsistency" for lively discussions of these ideas. Nuseibeh and Russo acknowledge the financial support of the UK EPSRC for the projects MISE (GR/L 55964) and VOICI (GR/M 38582).

Appendix A. On local vs global consistency checking

In principle, global consistency cannot be proved through local consistency checking. For example, it is possible to add a statement to a set of descriptions that is consistent with each description, but which makes the whole set inconsistent. Consider the two viewpoints shown in Fig. 2. Each is expressed in a simple propositional logic, and we will assume that the terms have the same designations in each viewpoint (i.e. that a in T_1 means the same as a in T_2). Let us also suppose that there is a simple consistency relationship between them, that when composed, it should not be possible to derive a contradiction.

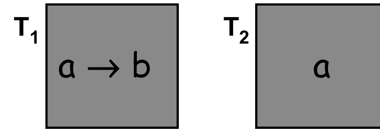

Fig. 2. Two consistent viewpoints.

Now consider the proposition $\neg b$. This proposition can be added to either viewpoint without making them internally inconsistent. However, adding it to either viewpoint will break the consistency rule that holds between them, as it will be possible to derive both b and $\neg b$. In general, elaborating a viewpoint, even if it preserves local consistency, may introduce inconsistencies with other viewpoints.

Furthermore, it is possible to establish the consistency of set of descriptions just by making local comparisons, even if you exhaustively compare, say, every pair of sentences. Consider the three viewpoints shown in Fig. 3. It is not hard to show that the three are consistent. We can merge them to form T:

$$T = T_1 \cup T_2 \cup T_3.$$

Note that $T \vdash d$.

Consider a new piece of information, Ψ, such that

$$\Psi : d \rightarrow \neg a.$$

Ψ is consistent with each of T_1, T_2 and T_3 individually, but is not consistent with T

$$T \cup \Psi \vdash \neg a \quad \text{and} \quad T \cup \Psi \vdash a.$$

Pairwise consistency checking between the viewpoints is not sufficient to reveal this. We could add Ψ to each of the viewpoints in turn and each pairwise union of the viewpoints would be consistent; i.e. each of:

$$\Psi \cup T_1 \cup T_2,$$
$$\Psi \cup T_1 \cup T_3,$$
$$\Psi \cup T_2 \cup T_3$$

is consistent. Finally, note that $T \vdash a$ and $T \cup \Psi \vdash a$. Imagine a is an important safety property to be checked in the specification composed from the three viewpoints. We could use a theorem prover to prove that a holds in T. After adding Ψ to one of the viewpoints, we can repeat the same proof to demonstrate that a still holds. Unless we check global consistency, we will not notice that in the latter case we can also derive $\neg a$.

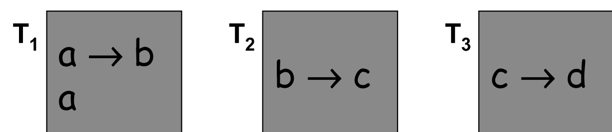

Fig. 3. Three consistent viewpoints.

Hence, we have demonstrated that a set of descriptions can be consistent when compared pairwise, but inconsistent when taken together. Furthermore, adding a new statement that is consistent with each description individually can still make the whole set of descriptions inconsistent. This has important implications throughout software engineering. For example, it is well known that testing pairwise integration of a set of software modules is not sufficient; system level testing is still necessary.

References

Balzer, R. Tolerating inconsistency. In: Proceedings of 13th International Conference on Software Engineering (ICSE-13), Austin, TX, USA. IEEE Computer Society Press, Silver Spring, MD, 1991, pp. 158–165.

Cugola, G., Nitto, E., Fuggetta, A., Ghezzi, C., 1996. A framework for formalizing inconsistencies and deviations in human-centered systems. Trans. Software Eng. Methodol. 5 (3), 191–230.

Easterbrook, S., Callahan, J., 1997. Formal methods for V&V of partial specifications: An experience report. In: Proceedings of 3rd International Symposium on Requirements Engineering (RE'97), Annapolis, USA. IEEE Computer Society Press, Silver Spring, MD, January 5–8, pp. 160–168.

Easterbrook, S.M., 1996. Learning from inconsistency. In: Proceedings of 8th International Workshop on Software Specification and Design (IWSSD-8), Paderborn, Germany, March 22–23, 1996. IEEE Computer Society Press, Silver Spring, MD, March 22–23, pp. 136–140.

Finkelstein, A.C.W., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B., 1994. Inconsistency handling in multi-perspective specifications. Trans. Software Eng. 20 (8), 569–578.

Gabbay, D., Hunter, A., 1991. Making inconsistency respectable: a logical framework for inconsistency in reasoning, Part 1 – A position paper. In: Proceedings of Fundamentals of Artificial Intelligence Research'91. Springer, Berlin, pp. 19–32.

Gabbay, D., Hunter, A., 1992. Making inconsistency respectable: a logical framework for inconsistency in reasoning, Part 2. In: Symbolic and Quantitative Approaches to Reasoning and Uncertainty, Lecture Notes in Computer Science. Springer, Berlin, pp. 129–136.

Ghezzi, C., Nuseibeh, B.A., 1998. Special issue on managing inconsistency in software development (1). Trans. Software Eng. 24 (11), 906–1001.

Ghezzi, C., Nuseibeh, B.A., 1999. Special issue on managing inconsistency in software development (2). Trans. Software Eng. 25 (11), 782–869.

Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G., 1996. Automated consistency checking of requirements specifications. Trans. Software Eng. Methodol. 5 (3), 231–261.

Holzmann, G.J., 1997. The model checker spin. Trans. Software Eng. 23 (5), 279–295.

Hunter, A., Nuseibeh, B., 1998. Managing inconsistent specifications: reasoning, analysis and action. Trans. Software Eng. Methodol. 7 (4), 335–367.

Jackson, M., 1995. Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices. Addison-Wesley, Reading, MA.

Leonhardt, U., Finkelstein, A., Kramer, J., Nuseibeh, B., 1995. Decentralised process modelling in a multi-perspective development environment. In: Proceedings of 17th International Conference on Software Engineering, Seattle, Washington, USA. ACM, New York, April 23–30, pp. 255–264.

Littlewood, B., 1994. Learning to live with uncertainty in our software. In: Proceedings of 2nd International Symposium on Software Metrics, vols. 2–8, London. IEEE Computer Society Press, Silver Spring, MD, October 24–26.

Narayanaswamy, K., Goldman, N., 1992. Lazy consistency: a basis for cooperative software development. In: Proceedings of 4th International Conference on Computer Supported Cooperative Work (CSCW'92), Toronto, Canada. ACM SIGCHI & SIGOIS, pp. 257–264.

Nuseibeh, B., 1997. Ariane 5: Who Dunnit?. IEEE Software 14 (3), 15–16.

Nuseibeh, B., Easterbrook, E., Russo, A., 2000. Leveraging inconsistency in software development. Computer 33 (4), 24–29.

Nuseibeh, B., Kramer, J., Finkelstein, A.C.W., 1994. A framework for expressing the relationships between multiple views in requirements specification. Trans. Software Eng. 20 (10), 760–773.

Robinson, W.N., 1990. Negotiation behaviour during multiple agent specification: a need for automated conflict resolution. In: Proceedings of 12th International Conference on Software Engineering (ICSE-12), Nice, France. IEEE Computer Society Press, Silver Spring, MD, March, pp. 268–276.

Russo, A., Nuseibeh, B.A., Kramer, J., 1998. Restructuring requirements specifications for inconsistency analysis: a case study. In: Proceedings of 3rd International Conference on Requirements Engineering (ICRE98), Colorado Springs, USA. IEEE Computer Society Press, Silver Spring, MD, pp. 51–60.

Schneider, F., Easterbrook, S.M., Callahan, J.R., Holzmann, G.J., 1998. Validating requirements for fault tolerant systems using model checking. In: Proceedings of 3rd International Conference on Requirements Engineering (ICRE-98), Colorado Springs, USA. IEEE Computer Society Press, Silver Spring, MD, April 6–10, pp. 4–13.

Schwanke, R.W., Kaiser, G.E., 1988. Living with inconsistency in large systems. In: Proceedings of the International Workshop on Software Version and Configuration Control, Grassau, Germany. Teubner, Stuttgart, January 27–29, pp. 98–118.

Spanoudakis, G., Finkelstein, A.C.W., Till, D., 1999. Overlaps in requirements engineering. Automated Software Eng. 6 (2), 171–198.

van Lamsweerde, A., Letier, E., 1998. Integrating obstacles in goal-driven requirements engineering. In: Proceedings of 20th International Conference on Software Engineering (ICSE-20), Kyoto, Japan. IEEE Computer Society Press, Silver Spring, MD, April 19–25, pp. 53–62.

Bashar Nuseibeh is a Professor of Computing at The Open University and Director of the Centre for Systems Requirements Engineering at Imperial College, London, UK. Previously, he was a Reader at the Department of Computing, Imperial College, London, and Head of its Software Engineering Laboratory. His research interests are in Requirements Engineering, Software Process Technology, Software Design, and Technology Transfer. He is Editor-in-Chief of the Automated Software Engineering Journal, and current Program Chair the 5th IEEE International Symposium on Requirements Engineering (RE'01), which will be held in Toronto, Canada, in 27–31st August 2001. More information at: http://mcs.open.ac.uk/ban25.

Steve Easterbrook is an Associate Professor in Department of Computer Science at the University of Toronto. Previously, he was a faculty member of the School of Cognitive and Computing Science at the University of Sussex, UK, from 1990 to 1995, and a Research Associate Professor at the NASA Independent Software Verification & Validation facility in West Virginia, from 1995 to 1999, where he led the facility's research team. His research interests focus on the problems of managing conflict and change in software requirements. Dr. Easterbrook regularly serves on program committees for conferences in requirements engineering and automated software engineering, and is General Chair for the International Symposium on Requirements Engineering (RE'01).

Alessandra Russo is a Lecturer in the Department of Computing, Imperial College, London, UK. Previously she was a Research Associate and Ph.D. student also at Imperial College. Her research interests are in both mathematical logic and software engineering in general, and in the applications of logic and automated reasoning in requirements engineering. She serves on a number of international program committees and as an investigator on a UK EPSRC project on "Handling Inconsistency and Change in Evolving Requirements Specifications".