Leveraging Inconsistency in Software Development

Bashar Nuseibeh
Imperial College
Steve Easterbrook
University of Toronto
Alessandra Russo
Imperial College

Maintaining consistency at all times is counterproductive. In many cases, it may be desirable to tolerate or even encourage inconsistency to facilitate distributed teamwork and prevent premature commitment to design decisions.

In 1995, Michael Jackson accurately described software engineering as a discipline of description.[1] Software engineers make use of many descriptions, including analysis models, specifications, designs, program code, user guides, test plans, change requests, style guides, schedules, and process models. But since different developers construct and update these descriptions at different times during development, establishing and maintaining consistency among descriptions presents several problems:

- descriptions vary greatly in their formality and precision;
- individual descriptions may themselves be ill-formed or self-contradictory;
- descriptions evolve throughout the life cycle at different rates; and
- checking consistency of a large, arbitrary set of descriptions is computationally expensive.

Checking the consistency of a large set of descriptions is a particularly difficult task. As our sets of descriptions grow, it very quickly becomes infeasible to test their consistency. Furthermore, incremental or localized consistency strategies do not guarantee global consistency. 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 truly intractable.

Existing approaches to this problem have been ad hoc or have only addressed a limited part of the life cycle. Tools exist to check the consistency of specific documents—object models, for example—but not for testing consistency between these documents and other development artifacts. To make matters worse, existing software development techniques assume consistency. And many software development environments attempt to enforce it. Most developers view inconsistency as undesirable, something to be avoided if at all possible. But most also recognize that their descriptions are frequently inconsistent and learn to live with these inconsistencies.

A systematic approach to managing inconsistency can help solve many of these problems. Inconsistency draws attention to problem areas, which means you can use inconsistency as a tool to

- improve the development team's shared understanding,
- direct the process of requirements elicitation, and
- assist with verification and validation.

To turn inconsistency into a tool, however, inconsistency management must become central to your development process.[2]

0018-9162/00/$10.00 © 2000 IEEE

WHAT IS INCONSISTENCY?

We use the term inconsistency to denote any situation in which a set of descriptions does not obey some relationship that should hold between them.[3] The relationship between descriptions can be expressed as a consistency rule against which the descriptions can be checked. In current practice, some rules may be captured in descriptions of the development process; others may be embedded in development tools. However, the majority of such rules are not captured anywhere.

Here are three examples of consistency rules expressed in English:

1. In a dataflow diagram, if a process is decomposed in a separate diagram, the input flows to the parent process must be the same as the input flows to the child dataflow diagram.
2. For a particular library system, the concept of an 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.
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 status of the SRS is changed to "approved."

The first rule applies to two descriptions written in the same notation. The second rule describes a consistency relationship that must be maintained between three different documents. The third rule expresses a relationship between the status of two descriptions to ensure that they are consistent with the stated development process.

The second and third rules reflect a common pattern: A consistency relationship exists between two descriptions because a third description says it should. Problems occur if the three-way relationship is untraceable or if you make changes to one of the three descriptions without cross-checking the others. The move toward better process modeling has helped to ensure that more of the process relationships, such as our third example, are documented.

Our definition of inconsistency is deliberately broad. It allows you to manage inconsistency systematically across many different types of document, without worrying about which notation has been used for the individual descriptions.

THE MANAGEMENT FRAMEWORK

To clarify our understanding of inconsistency, we developed the framework shown in Figure 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 evolving set of descriptions for inconsistencies. When inconsistencies are detected, some diagnosis is performed to locate and identify the cause.

At this point, you would choose from among several different inconsistency-handling strategies, includ-

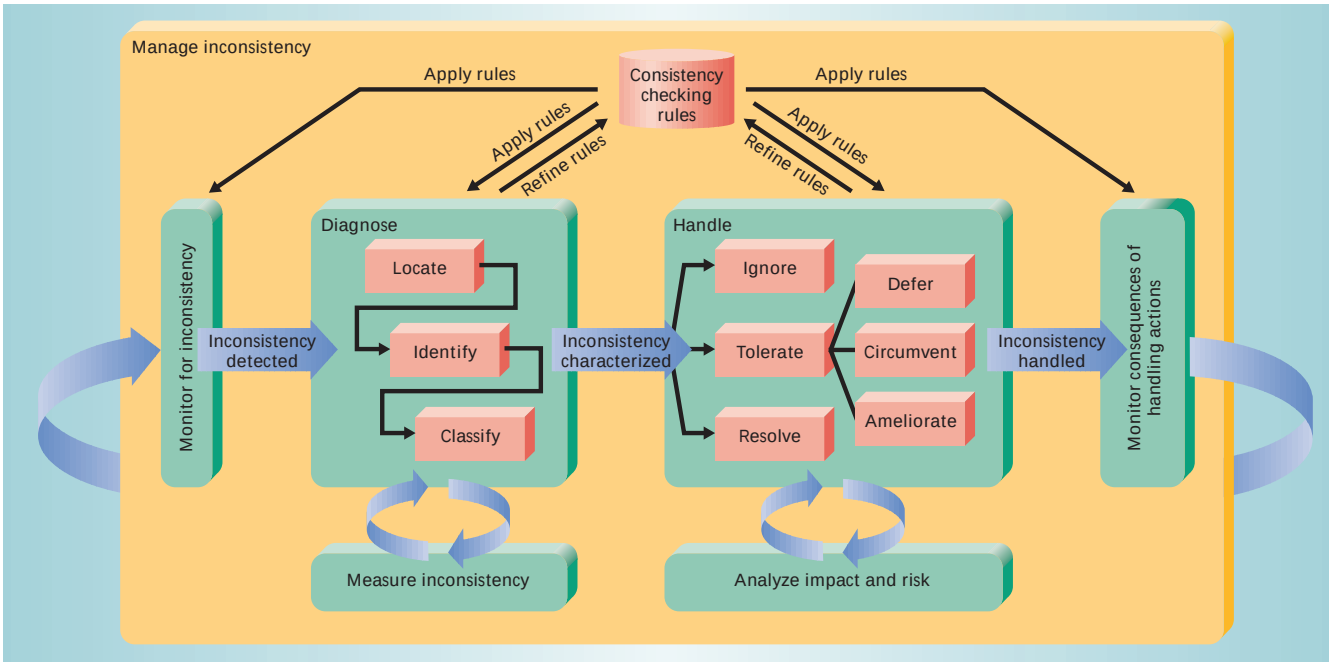

Figure 1. A framework for managing inconsistency.

The choice of an inconsistency-handling strategy depends on the context and the impact it has on other aspects of the development process.

ing resolving the inconsistency immediately, ignoring it completely, or tolerating it for a while. Whatever action you choose, the result needs to be monitored for undesirable consequences.

Checking consistency rules

In our framework, when you iterate through the consistency management process, you expand and refine the set of consistency rules. You will never obtain a complete set of rules covering all possible consistency relationships in a large project. However, the rule base acts as a repository for recording those rules that are known or discovered so that they can be tracked appropriately.

Consistency rules can emerge from several sources:

- *Notation definitions*. Many notations have well-defined syntactic integrity rules. For example, in a strongly typed programming language, the notation requires that the use of each variable be consistent with its declaration.
- *Development methods*. A method provides a set of notations, with guidance on how to use them together. For example, a method for designing distributed systems might require that for any pair of communicating subsystems, the data items to be communicated must be defined consistently in each subsystem interface.
- *Development process models*. A process model typically defines development steps, entry and exit conditions for those steps, and constraints on the products of each step.
- *Local contingencies*. Sometimes a consistency relationship occurs between descriptions, even though the notation, method, or process model does not predetermine this relationship. Examples include words used as synonyms, and relationships between timing values in parallel processes.
- *Application domains*. Many consistency rules arise from domain-specific constraints.

Monitoring and diagnosing 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—the descriptions can be checked at specific points during development, using a lazy consistency strategy.[4]

Our approach defines a scope for each rule, so that each edit action need be checked only against those rules that include in their scope the locus of the edit action.

When you find an inconsistency, the diagnosis process begins. Diagnosis includes

- locating the inconsistency by determining what parts of a description have broken a consistency rule;
- identifying the cause of an inconsistency, normally by tracing back from the manifestation to the cause; and
- classifying an inconsistency.

Classification is an especially important stage in the process of selecting a suitable handling strategy. Inconsistencies can be classified along a number of different dimensions, including the type of rule broken, the type of action that caused the inconsistency, and the impact of the inconsistency.

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. But it often relies on resolving fundamental conflicts or making important design decisions. In such cases, immediate resolution is not the best option. You can ignore, defer, circumvent, or ameliorate the inconsistency.

Sometimes the effort to fix an inconsistency is significantly greater than the risk that the inconsistency will have any adverse consequences. In such cases, you may choose to ignore the inconsistency. Good practice dictates that such decisions should be revisited as a project progresses or as a system evolves.

Deferring the decision until later may provide you with more time to elicit further information to facilitate resolution or to render the inconsistency unimportant. In such cases, flagging the affected parts of the descriptions is important.

Sometimes software developers won't regard a reported inconsistency as an inconsistency. This may be because the rule is incorrect or because the inconsistency represents an exception to the rule. In these cases, the inconsistency can be circumvented by modifying the rule or by disabling it for a specific context.

Sometimes, it may be more cost-effective to ameliorate an inconsistency by taking some steps toward a resolution without actually resolving it. This approach may include adding information to the description that alleviates some adverse effects of an inconsistency and resolves other inconsistencies as a side effect.

Measuring inconsistency

For several reasons, measurement is central to effective inconsistency management. Developers often need to know the number and severity of inconsistencies in their descriptions, and how various changes that they make affect these measures. Developers may also use

these measures to compare descriptions and assess, given a choice, which is preferred.

Sometimes developers need to prioritize inconsistencies in different ways to identify inconsistencies that need urgent attention. They may also need to assess their progress by measuring their conformance to some predefined development standard or process model.

The actions taken to handle inconsistency often depend on an assessment of the impact these actions have on the development project. Measuring the impact of inconsistency-handling actions is therefore a key to effective action in the presence of inconsistency. You also need to assess the risks involved in either leaving an inconsistency or handling it in a particular way.

PRACTICAL INCONSISTENCY MANAGEMENT

To refine the framework, we performed a number of practical case studies that provided some insights into the nature of inconsistency and its management. The first two studies dealt with parts of the requirements specifications for the command and control software for the International Space Station.[5,6] The third case study dealt with the design of a dual-redundant controller for a deep-space probe.[7] We based all three case studies on analysis of existing, evolving specifications expressed in a mixture of prose, tables, flowcharts, and other diagrams.

While each case study provided the opportunity to employ different techniques for analyzing the specifications, our approach in each case was to re-represent and restructure the specifications more precisely, more formally, and at different levels of abstraction. Our primary goal was to permit more detailed analysis than would otherwise have been possible.[8] These case studies provided us with several valuable insights.

Lingering inconsistencies

The observation that some inconsistencies never get fixed seems counterintuitive 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; eventually a consistent specification would be needed as a basis for an implementation. In practice, this is not true. Many local factors affect how you handle an inconsistency, including the cost of resolution, the cost of updating the documentation, and the developers' level of shared understanding. Ultimately, the decision to repair an inconsistency is risk-based. If the cost of fixing it outweighs the risk of ignoring it, then it makes no 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 to ameliorate this inconsistency by adding a disclaimer that the text should be regarded as definitive whenever it failed to match the diagram. Despite the inconsistency, developers still found the diagram useful because it provided an overview of the requirements expressed in the text.

On another occasion, we had a problem with the analysis models we abstracted from the original specifications. These were state machine models that captured the behavior 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 because they allowed partial checks of key properties. 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 cases, the inconsistency did not need to be resolved; it was sufficient just to be aware of its existence. In each case, we based the decision to ignore the inconsistency on a careful analysis of the risk involved. If we hadn't detected the inconsistency, we couldn't have performed the risk analysis.

The decision to repair an inconsistency is risk-based. If the cost of fixing it outweighs the risk of ignoring it, then it makes no sense to fix it.

Reevaluating risk

The decision to tolerate an inconsistency is a risk-based decision. Because risk factors change during the development process, you have to reevaluate risk periodically. Ideally, you reevaluate risk by monitoring each unresolved inconsistency for changes in the factors that affect the decision to tolerate it. In practice, such monitoring isn't feasible with current tools, so the usual approach is to identify key points in the future at which the decision must be reevaluated.

Consider the Ariane-5 rocket,[9] which reused much of the software from Ariane-4. The project team tolerated an inconsistency in Ariane-4 between the safety requirement that all exceptions be handled, and the implementation in which some floating-point exceptions were left unhandled, to meet memory and performance requirements. The analysis concluded, correctly, that particular exceptions would never occur, so the risk was minimal. Unfortunately, the Ariane-4 project team never repeated the risk analysis when they reused the software in Ariane-5. The decision to tolerate the inconsistency was not wrong for Ariane-4, but the lack of available tools to indicate that the risk needed to be reevaluated for Ariane-5 proved disastrous.

We once believed that as long as relationships are precisely defined, consistency could be determined objectively.

The Ariane-5 developers did not have a method to warn them they must revisit decisions when the design parameters changed. Knowing when and how to reevaluate these decisions is critical.

Consistency checks

Resolving an inconsistency has a cost associated with it that might make it not worth doing in some cases. And sometimes a consistency check is not worth performing, either.

In our 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 the consistency checking we had performed on our model became irrelevant.

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 may be project-specific, although process models may provide some guidance. As part of our research on guiding the inconsistency management process, we examined the conditions under which consistency checking should and should not be performed, and the mechanisms for guiding this process.[10]

Inconsistency is deniable

Our framework relies on a well-defined set of relationships between descriptions. We once believed that as long as relationships are precisely defined, consistency could be determined objectively. But we found that developers often debated whether a reported inconsistency really was an inconsistency.

Two factors are at work here. People generally don't 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 so that both parties feel they are working toward the common goal of creating a high-quality product. Despite this focus, inconsistency still carries a stigma that implies poor-quality work. If the V&V team declares an inconsistency publicly—at a formal review, for example—authors tend to become defensive. They may give an argument for why the inconsistency is not really an issue or claim that they are already aware of the problem and have fixed it or are in the process of doing so.

The second factor is a modeling issue. Descriptions can be inconsistent because one or more of them is inaccurate or vague. Although we can formalize a description so 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 uncertainty sometimes results in a useful discussion of the descriptions' nature, which may in turn lead to an improvement in how the descriptions 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.

Several available tools 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 description type. Such method-specific consistency checking is extremely useful but covers only a fraction of the range of consistency relationships that can affect software development.

Our next step is to develop our framework into a software development environment in which different techniques for inconsistency management[11] play a central role. ❖

Acknowledgments

We thank our colleagues Frank Schneider, John Hinkle, Dan McCaugherty, and Chuck Neppach, who all worked on the case studies. We also 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).

References

1. M. Jackson, *Software Requirements & Specifications: A Lexicon of Practice, Principles, and Prejudices*, Addison-Wesley, Wokingham, England, 1995.
2. B. Nuseibeh, "To Be and Not to Be: On Managing Inconsistency in Software Development," *Proc. Eighth Int'l Workshop on Software Specification and Design*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 164-169.
3. B. Nuseibeh, J. Kramer, and A.C.W. Finkelstein, "A Framework for Expressing the Relationships between Multiple Views in Requirements Specification," *IEEE Trans. Software Eng.*, 1984, pp. 760-773.
4. K. Narayanaswamy and N. Goldman, "Lazy Consistency: A Basis for Cooperative Software Development,"

Proc. 4th Int'l Conf. Computer Supported Cooperative Work, ACM Press, New York, 1992, pp.257-264.
5. S.M. Easterbrook and J. Callahan, "Formal Methods for Verification and Validation of Partial Specifications: A Case Study," *J. Systems and Software*, Vol. 40, No. 3, 1998, pp. 199-210.
6. A. Russo, B.A. Nuseibeh, and J. Kramer, "Restructuring Requirements Specifications for Inconsistency Analysis: A Case Study," *Third Int'l Conf. Requirements Engineering*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 51-60.
7. F. Schneider et al., "Validating Requirements for Fault Tolerant Systems Using Model Checking," *Third Int'l Conf. Requirements Engineering*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 4-13.
8. C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Trans. Software Engineering and Methodology*, ACM Press, New York, Vol. 5, No. 3, 1998, pp. 231-261.
9. B.A. Nuseibeh, "Ariane 5: Who Dunnit?" *IEEE Software*, Vol. 14, No. 2, 1997, pp. 15-16.
10. U. Leonhardt et al., "Decentralized Process Modeling in a Multi-Perspective Development Environment," *Proc. 17th Int'l Conf. Software Eng.*, ACM Press, New York, 1995, pp. 255-264.
11. C. Ghezzi and B. Nuseibeh, "Guest Editorial: Introduction to the Special Section on Managing Inconsistency in Software Development," *IEEE Trans. Software Eng.*, 1999, pp. 782-783.

Bashar Nuseibeh is a lecturer and head of the software engineering laboratory in the Department of Computing at Imperial College, London. His research interests include requirements engineering, software specification, and technology transfer, and he serves as editor-in-chief of the Automated Software Engineering Journal. *Nuseibeh received a PhD in computer science from Imperial College. Contact him at ban@doc.ic.ac.uk.*

Steve Easterbrook is an associate professor in the Department of Computer Science at the University of Toronto, and he is general chair for the Fifth IEEE International Symposium on Requirements Engineering, to be held in Toronto in 2001. His research interests include the problems associated with managing conflict and change in software requirements. Easterbrook received a PhD in computer science from Imperial College in 1991. Contact him at sme@cs.toronto.edu.

Alessandra Russo is research associate in the Department of Computing at Imperial College, London. Her research interests include mathematical logics and their applications in computer science and software engineering. Russo received a PhD in computer science from Imperial College. Contact her at ar3@doc.ic.ac.uk.