Computer-Aided Inconsistency Management in Software Development

Bashar Nuseibeh

Department of Computing
Imperial College
London SW7 2BZ
Email: ban@doc.ic.ac.uk

Technical report DoC 95/4

0. Abstract

The incremental development of software systems involves the detection and handling of inconsistencies. These inconsistencies arise in system requirements, design specifications and, quite often, in the final implemented software product. In this paper we explore different kinds of inconsistency that arise during different stages of software development, and examine the scope and role of computer-based tool support for managing inconsistency in this setting. In addition to detecting and removing inconsistencies , managing inconsistency also includes a wide range of activities that facilitate continued development in the presence of inconsistency. These include procedures for controlled amelioration and avoidance of inconsistencies. The paper uses the ViewPoints framework for multi-perspective software development as a vehicle for the discussion, and as a test bed for tool support. The framework facilitates the development and composition of multiple partial specifications (ViewPoints), and is itself supported by automated tools that check and handle inconsistencies (The Viewer).

The paper makes a contribution towards a better understanding of the way in which complex software systems are developed, and consequently, the kind of automated tool support that needs to be provided in this setting.

Keywords: *inconsistency management, handling, specification, process modelling, CASE.*

1. Introduction

The development of a large and complex software system inevitably involves the management of inconsistencies. Inconsistencies may arise in the early stages of development if, for example, contradictory requirements are specified. They may also arise in design specifications as developers explore alternative solutions; and in implementations if programmers, for example, fail to consider particular exceptions.

A large proportion of software engineering research has been devoted to consistency maintenance, or at the very least, has been geared towards eradicating inconsistencies as soon as they are detected.

In our previous work [10], we have proposed an approach to software development that is tolerant of inconsistencies, and which attempts to provide techniques for handling them explicitly. We have also explored the consequences of taking a radically decentralised approach to software

development in which multiple development participants hold multiple - often inconsistent - views [12]. In this paper, we examine the implications that such an approach has on the provision and nature of automated tool support. In particular, we compare and contrast the scope and role of tools that support inconsistency management, with that of tools that attempt to adopt consistency maintenance through strict enforcement or immediate resolution.

The paper is organised as follows. We begin by outlining the origins and kinds of inconsistencies that arise in software development (section 2), and how they may be detected and identified (section 3). With reference to related work, we then discuss strategies for handling inconsistencies (section 4), and examine computer-based tool support for inconsistency management in this setting (section 5). We conclude with a classification of current and desirable tools for supporting such activities, discuss of outstanding research issues, and outline an agenda for future work (section 6).

2. Causes of Inconsistencies

Inconsistency is an inevitable part of a complex, incremental software development process. Even in an idealised process, system requirements are often uncertain or contradictory, alternative design solutions exist, and errors in implementation arise.

The requirements engineering stage of development is particularly illustrative of such inconsistencies. During requirements acquisition, customer requirements are often sketchy and uncertain. For large projects in particular, a number of "client authorities" may exist who have conflicting, even contradictory requirements. In many instances customers may not even be certain of their own needs, and a requirements engineer's job is partly to elicit and clarify these needs. The requirements specification produced as a result of such a specification and analysis process however is not static: it continues to evolve as new requirements are added and conflicts identified are resolved. In fact, even with strict project management practices in place, requirements specifications - and subsequent design specifications - continue to evolve.

Thus, there is a wide range of possible causes of inconsistencies and conflicts in software development. Many of these are due to the heterogeneity of the products being developed (e.g., systems deploying different technologies) and the multiplicity of stakeholders and/or development participants involved in the development process. Inconsistencies arise between multiple development participants because of:

- the different views they hold,
- the different languages they speak,
- the different development strategies (methods) they deploy,
- the different stages of development they address,
- the partially, totally or non-overlapping areas of concern they have, and

- the different technical, economic and/or political objectives they want to achieve.

While inconsistencies can occur in software development processes and products for a variety of reasons, we adopt a simple definition of what actually constitutes an inconsistency:

An inconsistency occurs if and only if a (consistency) rule has been broken.

Such a rule explicitly describes some form of relationship or fact that is required to hold. In previous work, we have examined three uses of such consistency rules. They may describe syntactic relationships between development artefacts prescribed by a development method, which is also a way of describing semantic relationships between artefacts produced by that method [21]. They may also be used to prescribe relationships between the sub-processes in an overall development process, which is also a way of coordinating the activities of developers deploying different development strategies [6]. Finally, consistency rules can be used to describe user-defined relationships that emerge as development of a software specification proceeds [7]. This is useful for capturing ontological relationships between the products of a development process (for example, two developers specifying a library system may use the term "user" and "borrower" to refer to the same person).

Reducing an inconsistency to the breaking of a rule facilitates the identification of inconsistencies in specifications, and is a useful tool for managing other "problems" that arise during software development. For example, if we treat *conflict* as the interference of the goals of one party caused by the actions of another party [8], then we can use inconsistency as a tool for detecting many conflicts[1]. Similarly, if we define a *mistake* as an action that would be acknowledged as an error by its perpetrator (e.g., a typo), then we can detect mistakes that manifest themselves as inconsistencies.

Hagensen and Kristensen have also explicitly explored the consistency perspective in software development [14]. The focus of their work is on the structures for representing information ("descriptions") and the relations between these structures. Consistency of descriptions is defined as relations between interpretations of descriptions. Consistency handling techniques in software systems modelled in terms descriptions, interpretations and relations, are also proposed.

3. Detecting and Identifying Inconsistencies

Detecting an inconsistency that breaks an explicit rule is relatively straight forward. For example, a type checker can check whether or not an instance or variable conforms to its type definition. Similarly, a parser can check whether or not a sentence conforms to the syntactic rules specified by its grammar.

---

1 Of course, not all conflicts will manifest themselves as inconsistencies, particularly if the conflict is caused by a "conceptual disagreement" such as a difference in personal values.

Simple inferences in classical logic can also be used to detect logical inconsistencies resulting from too much or too little information. For example, a *contradiction* (where a rule of the form $X \vee \neg X$ has been broken) may be detected in this way.

Other kinds of inconsistency are more difficult to detect. A conflict between two development participants may not manifest itself as an inconsistency until further development has taken place (making the original source of the inconsistency difficult to identify). Furthermore, what actually constitutes an inconsistency from one participant's perspective may not be the case from another perspective. An example of this is an "inconsistency" in a person's tax return. Such an inconsistency may actually be a "desirable" piece of information from a tax inspector's point of view!

One of the difficulties in handling inconsistencies effectively, even after they have been successfully detected, is that the kind of inconsistency detected also has to be identified. The CONMAN (*con*figuration *man*agement) project [26] for example, attempts to classify consistency in programs into one of six kinds in order to facilitate inconsistency handling later on:

- Full consistency - where a system satisfies the rules that a programming language specifies for legal programs (insofar as they can be checked prior to execution).
- Type consistency - where a system satisfies the static type checking rules of the programming language.
- Version consistency - where a system is built using exactly one version of each logical source code file.
- Derivation consistency - where a system is operationally equivalent to some version consistent system.
- Link consistency - where each compilation unit is free of static type errors, and each symbolic reference between compilation units is type safe according to the rules of the programming language.
- Reachable consistency - where all code and data that could be accessed or executed by invoking the system through one of its entry points are safe.

The CONMAN system checks for all six kinds of consistency automatically, and then reacts differently depending on the kind of inconsistency detected. It does however appear appropriate for configuration management applications only, and it is therefore desirable to identify a more general set of inconsistencies that arise during software development in-the-large.

4. Handling Inconsistencies

Many approaches to handling inconsistency attempt to maintain and enforce consistency, usually by adopting simple procedures for inconsistency detection followed by immediate resolution. We now examine alternative approaches to inconsistency handling that "tolerate inconsistency" [2] in a variety of ways.

We believe that these approaches represent more realistic attempts at supporting software development, and we therefore discuss some discuss some general techniques for acting and reasoning in the presence of inconsistency.

4.1. Related work

Schwanke and Kaiser suggest that during large systems development, programmers often circumvent strict consistency enforcement mechanisms in order to get their jobs done [26]. They propose an approach to "living with inconsistency" during development (implemented in the CONMAN programming environment described in section 4). CONMAN helps programmers handle inconsistency by:

- identifying and tracking the six different kinds of inconsistencies described above (without requiring them to be removed),
- reducing the cost of restoring type safety after a change (using a technique called "smarter recompilation"), and,
- protecting programmers from inconsistent code (by supplying debugging and testing tools with inconsistency information).

Balzer proposes the notion of "tolerating inconsistency" by relaxing consistency constraints during development [2]. The approach suggests that inconsistent data be marked by guards ("pollution markers") that have two uses: (1) to identify the inconsistent data to code segments or human agents that may then help resolve the inconsistency, and (2) to screen the inconsistent data from other segments that are sensitive to the inconsistencies. The approach does not however provide any mechanism for specifying actions that need to be performed in order to handle these inconsistencies.

Gabbay and Hunter suggest "making inconsistency respectable" by proposing that inconsistencies be viewed as signals to take external actions (such as "asking the user" or "invoking a truth maintenance system"), or as signals for taking internal actions that activate or deactivate other rules [13]. Again, the suggestion is that "resolving" inconsistency is not necessarily done by eradicating it, but by supplying rules that specify how to act in the presence of such inconsistency. Gabbay and Hunter further propose the use of temporal logic to specify these meta-level rules. We have adapted this approach to a multi-perspective software development [10], in which logical inconsistencies between partial specifications (ViewPoints) are detected by translating them into classical logic, and then using an action-based temporal logic to specify inconsistency handling rules.

Narayanaswamy and Goldman propose "lazy" consistency as the basis for cooperative software development [18]. This approach favours software development architectures where impending or proposed changes - as well as changes that have already occurred - are "announced". This allows the consistency requirements of a system to be "lazily" maintained as it evolves.

The approach is a compromise between the optimistic view in which inconsistencies are assumed to occur infrequently and can thus be handled individually when they arise, and a pessimistic approach in which inconsistencies are prevented from ever occurring. A compromise approach is particularly realistic in a distributed development setting where conflicts or "collisions" of changes made by different developers may occur. Lazy consistency maintenance supports activities such as negotiation and other organisational protocols that support the resolution of conflicts and collisions.

Finally, Feather has recently proposed an approach to modularised exception handling [9], in which programs accessing a shared database of information impose their own assumptions on the database, and to treat exceptions to those assumptions differently. The assumptions made by each program together with their respective exception handlers are used to provide each program with its own individual view of the database. Alternative - possibly inconsistent - views of the same information can therefore be used to support different users or developers of a software system.

Table 1 summarises the various approaches to inconsistency handling described above.

Approach	Mechanism	Scope
Living with inconsistency [26]	Smarter recompilation	Programming (Configuration Management)
Tolerating inconsistency [2]	Pollution markers	Programming
Making inconsistency respectable [13] *Application: Inconsistency handling between multiple perspectives [10]*	Meta-level temporal rules	(Logic) Databases *Multiple ViewPoints in software development*
Lazy inconsistency [18]	Announce proposed changes	Cooperative software development
Modularised exception handling [9]	Multiple exception handlers	Programming views

Table 1: *Inconsistency handling in software development.*

4.2. Acting in the presence of inconsistency

The inconsistency handling approaches described above address inconsistencies in different ways. What they have in common however, is the goal of allowing continued development in the presence of inconsistency. A number of strategies for achieving this may be adopted.

- *Ignoring* the inconsistency completely and continuing development regardless. This may be appropriate in certain circumstances where the inconsistency is isolated and does not prevent further development from taking place.
- *Circumventing* the inconsistent parts of the system being developed and continuing development. This may be achieved by marking inconsistent

portions of the system (e.g., using Balzer's "pollution markers") or by continuing development in certain directions depending on the kind of inconsistency identified (e.g., as in CONMAN).

- *Removing* the inconsistency altogether by correcting any mistakes or resolving conflicts. This depends on a clear identification of the inconsistency and assumes that the actions required to fix it are known. Restoring consistency completely can be difficult to achieve, and is quite often impossible to automate completely without human intervention.
- *Ameliorating* inconsistent situations by performing actions that "improve" these situations and increase the possibility of future resolution. This is an attractive approach in situations where complete and immediate resolution is not possible (perhaps because further information is required from another development participant), but where some steps can be taken "fix" part or some of the inconsistent information.

Logic-based approaches offer promising contributions to inconsistency handling by providing techniques that compute "minimal inconsistent subsets" of an inconsistent database or specification. This allows developers to continue development (by avoiding the inconsistent information), and to isolate the inconsistent information which can then be analysed at leisure. Logic-based approaches may also facilitate reasoning in the presence of inconsistency, and we are currently examining the use of extensions of classical logic for this purpose [3].

Promising contributions are also offered by work on fault-tolerant distributed systems [4] where continued operation of these systems is still possible in the presence of failure. Failure in this context is analogous to inconsistency in software development, and includes problems caused by omissions (e.g., server not responding to input), timing (e.g., server response is too early or too late), response (e.g., server response is incorrect) and crashes. What fault-tolerant systems have in common in such failure scenarios is their ability to react to these failures and continue operating. In fact, analysis of many such failures often produces information that identifies hitherto undetected errors which can then also be repaired.

Finally, it is worth noting that what we have been discussing thus far are inconsistencies that arise in the artefacts of software development (e.g., specifications, programs, systems, etc.). Inconsistencies can also arise in software development processes themselves. An interesting example of these is an inconsistency which occurs between a software development process definition and the actual (enacted) process instance [5]. Such an inconsistency between "enactment state" and "performance state" is often avoided by blocking further development activities until some precondition is made to hold. Since this policy is overly restrictive, many developers attempt to fake conformance to the process definition (for example, by fooling a tool into thinking that a certain task has been performed in order to continue development). What is therefore needed is a software development process

which is flexible enough to tolerate development that diverges from its definition, or a process that can be dynamically corrected, changed and/or improved as it is being enacted.

5. Computer-Aided Inconsistency Management

Whatever the cause or kind of inconsistency that exists in a software development process or product, there is a need for automated tools that detect, identify, record, track and handle such inconsistencies in this setting. We now discuss the scope and kinds of tools that support such inconsistency management.

5.1. Scope

Figure 1 outlines three broad areas of inconsistency management that benefit from computer-based tool support, and what follows identifies their scope.

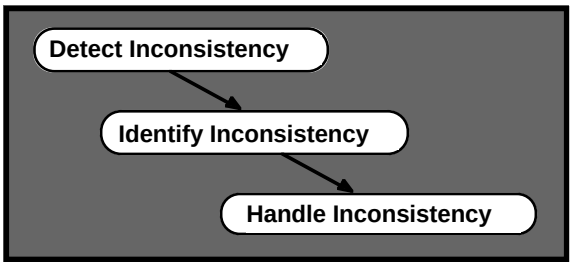

Figure 1: *Three activities of inconsistency management.*

Detecting Inconsistency. This includes a wide range of tools that check consistency rules, such as type checkers and parsers. Detecting inconsistency can be automated if the appropriate consistency rules can be defined precisely. Conflicts or mistakes that do not manifest themselves as inconsistencies (because no pre-defined rule was prescribed), cannot be detected automatically and normally require human involvement.

Identifying Inconsistency. Once an inconsistency has been detected, the next step is to identify the kind of inconsistency it is (perhaps by comparing it against some pre-defined classification of inconsistencies). Identifying inconsistency automatically can be difficult, particularly if there are multiple sources/causes of the inconsistency. However, once an inconsistency is identified, then removing it is often also simplified. Tools that detect inconsistency usually also attempt to identify or suggest its possible cause.

Handling Inconsistency. Reacting to inconsistencies in a system is a particularly challenging area for the provision of tool support. Many tools allow the inconsistency to be ignored or require actions to resolve it. Some of the tools described in section 5.1 also allow controlled development to continue in the presence of inconsistency. More tools are needed however for tracking inconsistencies in software systems, as well as tools that use this monitoring information to remove inconsistencies, or to ameliorate inconsistent information.

5.2. Tool support

The broad definition of inconsistency as the breaking of an consistency rule means that there is an equally broad range of tools that support inconsistency management. For example, most CASE tools [24, 27] feature syntactic consistency checkers that check well-formedness of diagrams, conformance to software engineering methods and so on. Process-centred environments on the other hand, check not only the artefacts of development, but also the process by which these artefacts are developed. In general however, most of these tools have limited inconsistency handling capabilities, concentrating instead on inconsistency detection and identification, and leaving inconsistency handling to be performed by the user of these tools. Nevertheless, some scope for conflict resolution is provided by negotiation-support tools [25].

A class of research tools known as theorem provers [16] also offer some scope for inconsistency handling in that they attempt to *prove* that a description (e.g., a specification) satisfies a set of properties or contains no contradictions. Therefore, these tools have the capability of reasoning about *why* an inconsistency exists when a proof cannot be produced.

A wide range of consistency checkers are also available for supporting programming activities in particular. These include interpreters and compilers which themselves deploy tools such as parsers and syntactic and semantic checkers [1]. Many integrated programming environments also provide automated support of inconsistency handling via tools such as static analysers and debuggers, which guide programmers through traces of inconsistent information in the process of trying to remove these inconsistencies.

5.3. Inconsistency handling in multi-perspective specifications

For a number of years, we have been investigating multi-perspective software development; that is, software development in which multiple development participants hold multiple views on a problem and/or solution domain. We have used the notion of a ViewPoint [12] to capture partial specification knowledge about an area of concern, together with partial knowledge about the representation scheme and process by which that partial specification is produced. We have used the separation of concerns offered by ViewPoints as a means for reducing software development complexity, and have deployed inter-ViewPoint rules as a means for integrating ViewPoints (and the software engineering methods upon which they are based) [22].

We have further developed *The Viewer* environment [19] to support the ViewPoints framework, and have used it as a vehicle for demonstrating the feasibility of our approach. In the area inconsistency management, *The Viewer*, provides a range of complementary tools. From a method designer or engineer's point of view, *The Viewer* facilitates the expression of consistency rules (both within and across ViewPoints). During actual development of

ViewPoints (method use), *The Viewer* provides tools for detecting, identifying and handling inconsistencies. The ConsistencyChecker shown in figure 2 for example is used to detect (selected) inconsistencies (in- and inter-ViewPoint), and can be used to invoke inconsistency handling tools as appropriate.

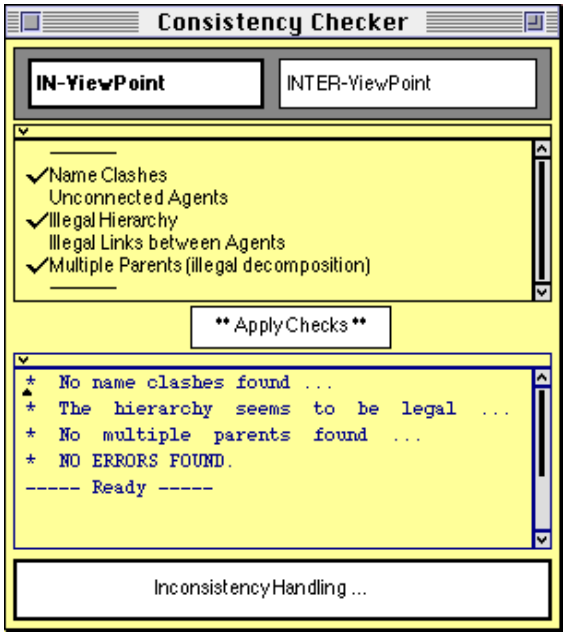

Figure 2: *A sample consistency checker in the Viewer environment. This tool can be used to check consistency of partial specifications (ViewPoints) internally (in-ViewPoint) and against other ViewPoints (inter-ViewPoint). The particular rules that the developer wishes to check may be selected, and executed by clicking on the "Apply Checks" button. If one or more inconsistencies are detected, then clicking on the "Inconsistency Handling" button invokes the appropriate inconsistency management tool (such as that shown in figures 3, 4 and 5).*

Three screen dumps from our prototype (inter-ViewPoint) inconsistency handler are shown in figures 3, 4 and 5. Basically, they illustrate the three different inconsistency handling activities that take place in a multi-perspective development environment. In figure 3, the developer is attempting to handle an inconsistency between two ViewPoints in his local ViewPoint, and is offered the option of editing his local ViewPoint specification and/or performing some further local consistency checks.

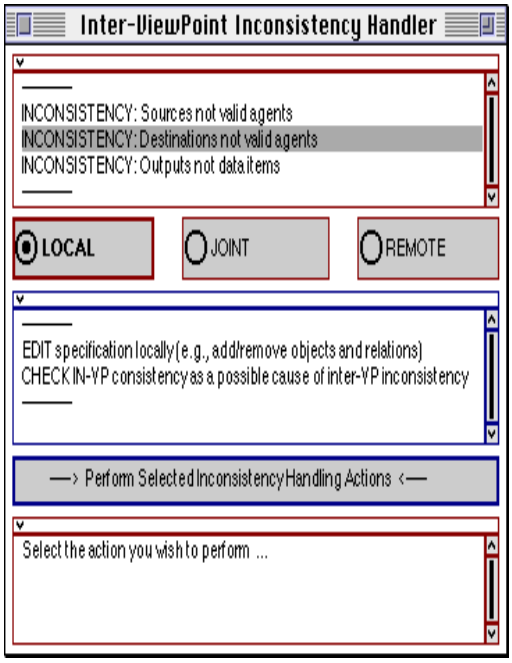

Figure 3: *A sample inconsistency handler in the Viewer environment. For the selected inconsistency (top), local handling of the inconsistency has been selected (left button clicked). In this case, possible handling actions include further editing of the ViewPoint specification or the application of some local checks.*

In figure 4, the developer has chosen to handle the selected inconsistency by transferring the relevant information to the other ("destination") ViewPoint with which the inconsistency arose. In other words, responsibility for handling the inconsistency is being transferred to another ViewPoint developer. The information transfer may include direct transfer of partial specification information or the "posting" of some appropriate "message".

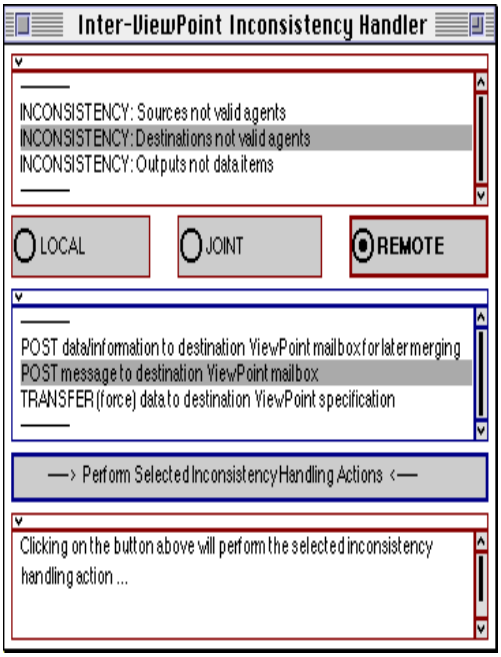

Figure 4: *A sample inconsistency handler in the Viewer environment. For the selected inconsistency (top), remote handling of the inconsistency has been selected (right button clicked). In this case, possible handling actions include transferring/posting information to another ViewPoint specification for remote handling.*

In figure 5, the inconsistency is to be handled jointly by the two ViewPoint developers involved in the failed consistency check. Options provided include further negotiation in order to understand the inconsistency better or resolve the conflict, or a declaration of deadlock (and presumably seeking a third party to arbitrate or provide more information.

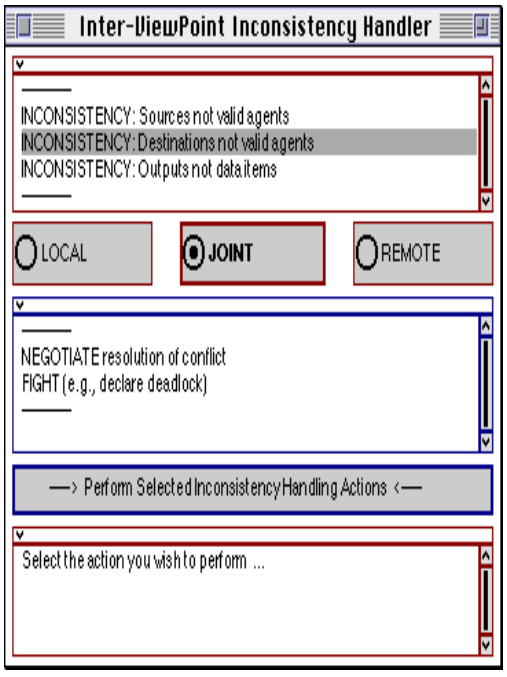

Figure 5: *A sample inconsistency handler in the Viewer environment. For the selected inconsistency (top), joint handling of the inconsistency has been selected (middle button clicked). In this case, possible handling actions include negotiation between the developers or war (i.e., external arbitration is required)!*

6. Discussion, Conclusions and Future Work

This paper has argued that software development processes must explicitly support the evolving nature of software systems (specifications and programs), and must therefore be capable of managing the inevitable inconsistencies and conflicts that arise in such systems. Managing inconsistencies in this setting does not necessarily mean removing them, although in many cases this may be desirable, rather, it involves: (a) detecting and identifying the kinds of inconsistencies, and possibly their source, and (b) continuing development in the presence of such inconsistencies, with a view to removing them later on down the line (or in some situations deferring resolution indefinitely). Often, intermediate steps that ameliorate the state of a specification or simply make progress towards removing inconsistencies in it, are also useful.

Of course, depending on the application for which a software system is being developed, different "levels of reliability" may also be acceptable. For non-safety-critical systems for example, some degree of uncertainty or inconsistency may be tolerated - even in the final product. In such cases however, there is a need to *measure* (or at least estimate) both the likely consequences and frequency of failures, in order to assess system reliability, and then devise ways of handling such failures [17].

A clearer understanding of the nature of software development is also needed in order to help identify and prioritise inconsistencies. An illustrative example of this is the distinction between an inconsistency reflecting an error in development, and an inconsistency that only exists temporarily because certain development steps have not been performed yet. The latter inconsistency is a part of every development and, is less important than the former, which reflects a more fundamental failure in development. If at all, current process modelling technology [11] provides guidance for "normal" development (e.g., "what should I do next?"), whereas we are attempting to handle inconsistencies that are usually labelled as "undesirable" in such a development process (e.g., "how do I get out of the mess I'm now in?") [20]. Moreover, because some inconsistencies can only be identified as a development process unfolds, we have been exploring process-guided approaches to inconsistency handling that analyse explicitly recorded development actions, then act according to the context of any inconsistencies detected [15, 23].

Another interesting temporal consideration in this setting is what we might call the "age" of an unresolved inconsistency. This is a measure of, say, the number of development actions that were performed since the last time an inconsistency was introduced by an action. It may be useful to explore the correlation, if any, between the age of an unresolved inconsistency and the degree of difficulty by which it may be handled or resolved. Intuitively, one would expect that the greater the age of the last consistency check, the higher the risk becomes, and that there is a trade-off between the cost of consistency checking and the cost of resolution (we measure risk in this context as the

likelihood of consistency failures multiplied by the cost of resolving them).

Finally, in this paper we have not explicitly addressed software development process considerations that determine *when* consistency checks should be performed, *how* these checks should be performed, and *what* should be done as result of performing these checks. Broadly speaking, deciding when to perform a consistency check should be determined by the process prescribed by a software development method. This should be designed to be "non-intrusive" since continuous reminders to perform checks are irritating and undesirable. The way in which checks are performed on the other hand, is an interaction issue and is determined by the context in which the check is applied. For example, in a cooperative development setting a negotiation protocol may be suitable, whereas in a distributed systems setting low-level communication protocols may be more appropriate. Finally, determining how to act once a consistency check has been performed is largely an inconsistency management issue that has been discussed in some detail in this paper.

We believe that many of the inconsistency management issues raised in this paper lie at the heart of software development. At all stages of a development life cycle, inconsistencies may arise and can be used to provide valuable input into a software development process. In fact, identifying and handling inconsistency in this context is a vehicle for monitoring and guiding development, and can be used as a tool for measuring many attributes of software development processes and products. We believe that there is a need to address issues of inconsistency management explicitly, and to provide computer-based tools that support this activity. These tools need to go beyond inconsistency avoidance or consistency maintenance, allow reasoning in the presence of inconsistency, and provide support for inconsistency handling which does not always involve immediate inconsistency resolution.

7. Acknowledgements

This work benefited greatly from discussions with Steve Easterbrook, Anthony Finkelstein, Tony Hunter and Jeff Kramer. It was partially funded by the UK Department of Trade and Industry as part of the Eureka Software Factory (ESF) project (grant reference number IED4/410/36/002), the UK EPSRC as part o the VOILA project, and the European Union as part of the Basic Research Action PROMOTER and the Information Systems Interoperability (ISI) projects (ECAUS003).

8. References

[1] Aho, A. V. and J. D. Ullman (1977); *Principles of Compiler Design*; Addison-Wesley Publishing Company, Reading Massachusetts, USA.

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

[3] Besnard, P. and A. Hunter (1994); "Quasi-classical Logic: Non-trvializable classical reasoning from inconsistent information"; *Technical report*, Department of Computing, Imperial College, London, UK, December 1994.

[4] Christian, F. (1991); "Basic Concepts and Issues in Fault-Tolerant Distributed Systems"; *Proceedings of International Workshop on Operating Systems of the 90s and Beyond*, Dagstuhl Castle, Germany, 8-12th July 1991, 119-149; LNCS 563, Springer-Verlag.

[5] Dowson, M. (1993); "Consistency Maintenance in Process Sensitive Environments"; *Proceedings of Workshop on Process Sensitive Environments Architectures*, Boulder, Colorado, USA, Rocky Mountain Institute of Software Engineering (RMISE).

[6] Easterbrook, S., A. Finkelstein, J. Kramer and B. Nuseibeh (1994); "Coordinating Distributed ViewPoints: The Anatomy of a Consistency Check"; *Concurrent Engineering: Research and Applications*, 2(3): CERA Institute, West Bloomfield, USA.

[7] Easterbrook, S. and B. Nuseibeh (1995); "Inconsistency Management in an Evolving Specification"; *(to appear in) Proceedings of 2nd International Symposium on Requirements Engineering (RE 95)*, York, UK, 27-29th March 1995, IEEE Computer Society Press.

[8] Easterbrook, S. M., E. E. Beck, J. S. Goodlet, L. Plowman, M. Sharples and C. C. Wood (1993); "A Survey of Empirical Studies of Conflict"; *(In) CSCW: Cooperation or Conflict?*; S. M. Easterbrook (Ed.); 1-68; Springer-Verlag, London.

[9] Feather, M. (1994); "Modularized Exception Handling"; *Draft technical report*, USC/Information Sciences Institute, Marina del Rey, California, USA, 25th October 1994.

[10] Finkelstein, A., D. Gabbay, A. Hunter, J. Kramer and B. Nuseibeh (1994); "Inconsistency Handling in Multi-Perspective Specifications"; *Transactions on Software Engineering*, 20(8): 569-578, August 1994; IEEE Computer Society Press.

[11] Finkelstein, A., J. Kramer and B. Nuseibeh (Eds.) (1994); *Software Process Modelling and Technology*, Advanced Software Development Series, Research Studies Press Ltd. (Wiley), Somerset, UK.

[12] Finkelstein, A., J. Kramer, B. Nuseibeh, L. Finkelstein and M. Goedicke (1992); "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development"; *International Journal of Software Engineering and Knowledge Engineering*, 2(1): 31-58, March 1992; World Scientific Publishing Co.

[13] Gabbay, D. and A. Hunter (1992); "Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning: Part 2"; *(In) Symbolic and Quantitative Approaches to Reasoning and Uncertainty;* 129-136; LNCS, Springer-Verlag.

[14] Hagensen, T. M. and B. B. Kristensen (1992); "Consistency in Software System Development: Framework, Model, Techniques & Tools"; *Software Engineering Notes (Proceedings of ACM SIGSOFT Symposium on Software Development Environments)*, 17(5): 58-67, 9-11th December 1992; SIGSOFT & ACM Press.

[15] Leonhardt, U., A. Finkelstein, J. Kramer and B. Nuseibeh (1995); "Decentralised Process Modelling in a Multi-Perspective Development Environment"; *(to appear in) Proceedings of 17th International Conference of Software Engineering*, Seattle, Washington, USA, 14-18th April 1995, IEEE Computer Society Press.

[16] Lindsay, P. A. (1988); "A Survey of Mechanical Support for Formal Reasoning"; *Software Engineering Journal (special issue on mechanical support for formal reasoning)*, 3(1): 3-27, January 1988; IEE, UK.

[17] Littlewood, B. (1994); "Learning to Live with Uncertainty in Our Software"; *Proceedings of 2nd International Symposium on Software Metrics*, London, UK, 24-26th October 1994, 2-8; IEEE Computer Society Press.

[18] Narayanaswamy, K. and N. Goldman (1992); ""Lazy" Consistency: A Basis for Cooperative Software Development"; *Proceedings of International Conference on Computer-Supported Cooperative Work (CSCW '92)*, Toronto, Ontario, Canada, 31st October - 4th November, 257-264; ACM SIGCHI & SIGOIS.

[19] Nuseibeh, B. and A. Finkelstein (1992); "ViewPoints: A Vehicle for Method and Tool Integration"; *Proceedings of 5th International Workshop on Computer-Aided Software Engineering (CASE '92)*, Montreal, Canada, 6-10th July 1992, 50-60; IEEE Computer Society Press.

[20] Nuseibeh, B., A. Finkelstein and J. Kramer (1993); "Fine-Grain Process Modelling"; *Proceedings of 7th International Workshop on Software Specification and Design (IWSSD-7)*, Redondo Beach, California, USA, 6-7 December 1993, 42-46; IEEE Computer Society Press.

[21] Nuseibeh, B., J. Kramer and A. Finkelstein (1993); "Expressing the Relationships Between Multiple Views in Requirements Specification"; *Proceedings of 15th International Conference on Software Engineering (ICSE-15)*, Baltimore, Maryland, USA, 17-21 May 1993, 187-200; IEEE Computer Society Press.

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

[23] Nuseibeh, B., J. Kramer, A. Finkelstein and U. Leonhardt (1995); "Decentralised Process Modelling"; *(to appear in) Proceedings of 4th European Workshop on Software Process Technology (EWSPT '95)*, Noordwijkerhout, 5-7th April 1995, Springer-Verlag.

[24] Rational (1992); "Rose: Rational Object-Oriented Software Engineering"; *Product Overview*, D-66B; Rational Technology Ltd., Brighton, UK, October 1992.

[25] Robinson, W. N. (1992); "Negotiation Behaviour During Requirements Specification: A need for automated conflict resolution"; *Proceedings of 12th International Conference on Software Engineering*, Nice, France, 26-30th March 1990, 268-276; IEEE Computer Society Press.

[26] Schwanke, R. W. and G. E. Kaiser (1988); "Living With Inconsistency in Large Systems"; *Proceedings of the International Workshop on Software Version and Configuration Control*, Grassau, Germany, 27-29 January 1988, 98-118; B. G. Teubner, Stuttgart.

[27] Wasserman, A. I. and P. A. Pircher (1987); "A Graphical, Extensible Integrated Environment for Software Development"; *SIGPLAN Notices (Proceedings of ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments)*, 22(1): 131-142, ACM Press.