Appears in BCS/IEE Software Engineering Journal, January 1996, pp31-43

sing iewPoints for Inconsistency Manage ent

STEVE EASTERBROOK[†]

School of Cognitive & Computing Sciences
University of Sussex, Falmer, Brighton, BN1 9QH
steve@cerc.wvu.edu

BASHAR NUSEIBEH

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

- Abstract

Large-scale software development is an evolutionary process. In an evolving specification, multiple development participants often hold multiple, inconsistent views on the system being developed, and considerable effort is spent handling recurrent inconsistencies. Detecting and resolving inconsistencies is only part of the problem: a resolved inconsistency might not stay resolved as a specification evolves. Frameworks in which inconsistency is tolerated help by allowing resolution to be delayed. However, the evolution of a specification may affect both resolved and unresolved inconsistencies.

We present and elaborate a framework in which software development knowledge is partitioned into multiple views called "ViewPoints". Inconsistencies between ViewPoints are managed by explicitly

Our work concentrates on requirements engineering. We take it for granted that requirements specifications evolve over a period of time. This evolution reflects a learning process, in that the specification is repeatedly updated and refined as more is learnt about the application domain. Furthermore, the learning process will not be monotonic: refinement involves retraction as much as adding detail. This learning process continues throughout the lifetime of a system, although we would expect the requirements specification to be frozen at some point.

The paper is organised as follows. We begin by discussing the notion of inconsistency in evolving specifications and motivating the need for inconsistency management (section 2). The ViewPoints framework within which we present our work is then briefly outlined, and the key issues of managing inconsistency in this framework are presented (section 3). The body of our work is presented by working through an example drawn from the behavioural specification of a telephone (section 4). We conclude the paper with a discussion of the issues arising from our example (section 5), a brief description of our prototype implementation (section 6), and present some conclusions and an agenda for future work (section 7).

- Inconsistencies in an Evolving Specification

If someone describes a specification as inconsistent, they usually mean that it contradicts itself, or that a logical contradiction can be derived directly from it. More generally, we regard an inconsistency as *any situation in which two parts of a specification do not obey some relationship that should hold between them*. This definition allows us to talk meaningfully about inconsistencies between partial specifications written in different notations. Notice that this definition subsumes logical contradiction, but does not require us to translate specifications into a formal notation to detect inconsistencies.

As we have defined inconsistency in terms of relationships that should hold, we cannot detect any such inconsistencies unless these relationships have been explicitly stated. The relationships may refer to both syntactic and semantic aspects of the specification. They may also be *process* relationships, in the sense that two parts of a specification may be inconsistent because they are at different stages of development.

Inconsistencies arise in an evolving specification for a number of reasons. They may be the result of mistakes, misunderstandings, or lack of information, especially where a specification is developed collaboratively. They may be the result of infeasible or impractical requirements, or because of conflicts between knowledge sources. Finally, because the notion of inconsistency is closely tied to the rules concerning correct use of a notation, inconsistencies may occur if a developer flouts the development method, perhaps because the method is too inflexible.

- - Tolerating inconsistency

A specification that contains an inconsistency is dangerous because it can be interpreted in more such a specification may see one or other of those versions, and may not realise there is an alternative version unless they are aware of the inconsistency. Any analysis of an inconsistent specification might be invalid, because it looked at the ‘wrong’ version. This problem is even more acute in a formal specification, where a logical contradiction (in the classical sense) allows any consequent to be derived by natural deduction[1].

For these reasons, maintenance of consistency has been given a high priority in software development environments, usually enforced through strict access control to a central database, and the use of a common data model or schema. However, maintaining global consistency at all times is expensive.

In many of the cases where a change to a specification would create an inconsistency, it is counter-productive to prevent the change being made. Enforcement of consistency means the change has to be delayed until the problem is sorted out, during which the desired change cannot be represented. It is often desirable to tolerate and even encourage inconsistency (Gabbay & Hunter, 1991), to maximise design freedom, to prevent premature commitment to design decisions, and to ensure all views are taken into account.

Inconsistencies can be tolerated during the development of a specification if we can overcome the versioning problem. For example, Balzer (1991) makes use of *pollution markers* to address this problem in a programming support environment. Pollution markers are used (a) to identify

- *Circumvent* - in some cases it may be sensible to circumvent the inconsistency by disabling or modifying the rule that was broken, for example because it represents a specific exception to a general consistency rule.
- *Ameliorate* - in many cases it will be possible to take steps that improve the situation, but which don't necessarily remove the inconsistency. We call this incremental resolution, and it is appropriate where resolution involves a number of actions by different parties, and where only some of these actions can be taken immediately. In this case, the actions need to be recorded, so that a record is available of the overall state of the resolution process.
- *Resolve* - to take actions that immediately repair the inconsistency. The actions may be as trivial as deleting elements of the specification that give rise to the inconsistency, or as complex as invoking a negotiation support tool to find a resolution.

- - Inconsistency i pl ies issing infor ation

We believe that inconsistency management is an invaluable tool for knowledge elicitation. In software development, one often talks about under- and over-specification (and, in fact, inconsistencies are often the result of either of these two cases). A requirements specification that

- Inconsistency and iewPoints

In order to manage inconsistencies and use them to support requirements elicitation in the manner we have described, a number of basic problems need to be addressed within a framework that is tolerant of inconsistency. We now outline these problems and describe the ViewPoints framework within which we address them.

- - Key probl e s of inconsistency anage ent

A key problem in managing inconsistencies in an evolving specification is *tracking* known inconsistencies and *recording* development information such as the circumstances that led to these inconsistencies. Recording such information facilitates detecting when inconsistencies accidentally get resolved, and support the choice of appropriate resolution actions. A record of detected inconsistencies may also be used to support incremental resolution; that is, actions which help but which do not necessarily resolve the inconsistency. Furthermore, the record may be used to keep track of actions which may potentially undo resolutions, as often happens during evolutionary software development.

Finally, developers must be allowed to define new relationships as part of the resolution process, or to modify or overrule default relationships. These relationships must also be recorded and tracked.

- - iewPoints

We base our work upon a framework for distributed software engineering, in which multiple perspectives are maintained separately as distributable objects, called “ViewPoints”. We will briefly describe the notion of a ViewPoint as it is used in this paper. The interested reader is referred to (Finkelstein, et al., 1992; Nuseibeh, 1994b) for a fuller account of the framework, and to

and how they inter-relate. Thus, the possible relationships between ViewPoints that are created during development are determined by the method.

Consistency checking is performed by applying a set of rules, defined by the method, which express the relationships that should hold between particular ViewPoints (Nuseibeh, et al., 1994). These rules define partial consistency relationships between the different representation schemes. This allows consistency to be checked incrementally between ViewPoints at particular stages rather than being enforced as a matter of course. A fine-grained process model in each ViewPoint provides guidance on when to apply a particular rule, and how resolution might be achieved if a rule is broken (Nuseibeh, et al., 1993).

- Scenario

Our scenario is drawn from the behavioural specification of a telephone, described using an extended state transition notation. A simplified version of this scenario was presented in Easterbrook and Nuseibeh (Easterbrook & Nuseibeh, 1995). Here we extend the scenario to show how our approach allows us to perform different kinds of consistency analysis on the same set of ViewPoints. In this case the method allows us to treat two statecharts firstly as partitioned behaviours of a single device, and secondly two separate (but interacting) instances of the device. In each case a different set of consistency relationships apply. Finally, we show how users can define their own consistency relationships to handle special cases.

We will begin by outlining the salient features of the method we use to elaborate the scenario, and then illustrate how we deploy the method to specify parts of our telephone system.

- - The ethod

Our method uses state transition diagrams to specify the required behaviour of a device, in this case a telephone. The method permits the partitioning of a state transition diagram describing a single device into separate ViewPoints, such that the union of the ViewPoints describes all the states and transitions of the device. Such separation of concerns is a powerful tool for reducing software development complexity in general (Ghezzi, et al., 1991), and requirements complexityin partic
but i consistcation of a t thoy c.211 Tc 05269 Tw 57tates s is now we ram d set
define
45is allorio to
0.98tates oan bees all thenariond trainvola

assumption that their descriptions could be merged at some point to give a complete state transition diagram for the handset.

The method provides the following:

a) A notation for expressing states and transitions diagrammatically. The state transition notation includes some of the extensions for expressing super-states and sub-states[2].

b) A partitioning step that allows a separate diagram to be created to represent a subset of the behaviours of a particular device. This may mean that on any particular diagram, not all the device's possible states are represented, and for some states, not all the transitions from them are represented.

c) A set of consistency checking rules which test whether partitioned diagrams representing the same device are consistent with one another. These rules test whether two diagrams describing the same device may be merged without any ambiguity; even though the checking process does not require such a merge to take place.

d) An analysis step that allows two ViewPoints to be treated as separate devices that interact. Some behaviours of one device will be associated with those of the other, so that for example a transition in one device causes a transition in another. Notice that in the example we use, these are the same two ViewPoints as in the previous steps; we will switch between treating them as partial descriptions of a single device, to partial descriptions of ***separate instances*** of a device.

e) A further set of consistency checking rules which test whether interacting devices whose transitions have been linked together exhibit consistent behaviours.

The method also includes guidance about when to use each of the steps, and when to apply the consistency rules. The scenario will illustrate each of these steps in turn.

- - Prel i inary specifications

At the start of our scenario, Anne has created a ViewPoint to represent the states involved in making a call (figure 1), and Bob has created a ViewPoint to represent the states involved in receiving a call (figure 2). As they are both describing states of the same device (that is, a telephone handset) [as permitted by step (b) of the method], a number of consistency relationships must hold between their ViewPoints [defined in step (c) of the method].

2 We usem BT 0.045 Tc /Rnr reCic) ofaween theiw1 7s Td cewPo.nr re methodos, anr9e j 0.067 T0 Tf 1t cR12 10 T

Hence, if Anne applies rule R_1, the result will be the predicate:

```
missing(transition(off hook, idle), B.transition(connected, idle), R1)
```

This states that according to rule R_1, the transition from ‘off hook’ to ‘idle’ in ViewPoint A requires that there be a transition from ‘connected’ to ‘idle’ in ViewPoint B, but it is missing[3]. This predicate is recorded as part of the history of Anne’s ViewPoint (in the ViewPoint’s work record slot). Normally, ViewPoint B is also notified of the results of the check.

Other actions are offered by the method designer. These are typically resolution actions that the method designer has identified after considering examples of the inconsistencies detected by the application of a rule. They may also have resulted from the experience of method users in the past: we assume that methods evolve as lessons are learnt about their use.

Eas erbrook, 1991) Tc /R108ns ar9 -229 67. Ihe

This fact is noted in Bob’s work record, but it is not immediately flagged to Bob, as there may be a large number of such effects.

- As initiator of the action, Anne’s ViewPoint re-applies rule R_2 to check that the inconsistency is indeed resolved.

Note that the rule R_1 is not re-applied automatically, despite the evidence in Bob’s ViewPoint that this too is resolved. There are two reasons for this: only Bob’s ViewPoint has the information about this side-effect, and the resolution process is only concerned with the inconsistency from rule R_2. Any effect on other inconsistencies can be dealt with when the ViewPoint owners specifically consider these.

- - Further elaboration

Anne and Bob now proceed to consider some additional features which will be made available on this phone. The first of these is the ability to forward a call to a third party. This requires Anne to add an ‘on hold’ state (figure 4). Note that her ‘connected’ state does not specify which party the phone is connected to.

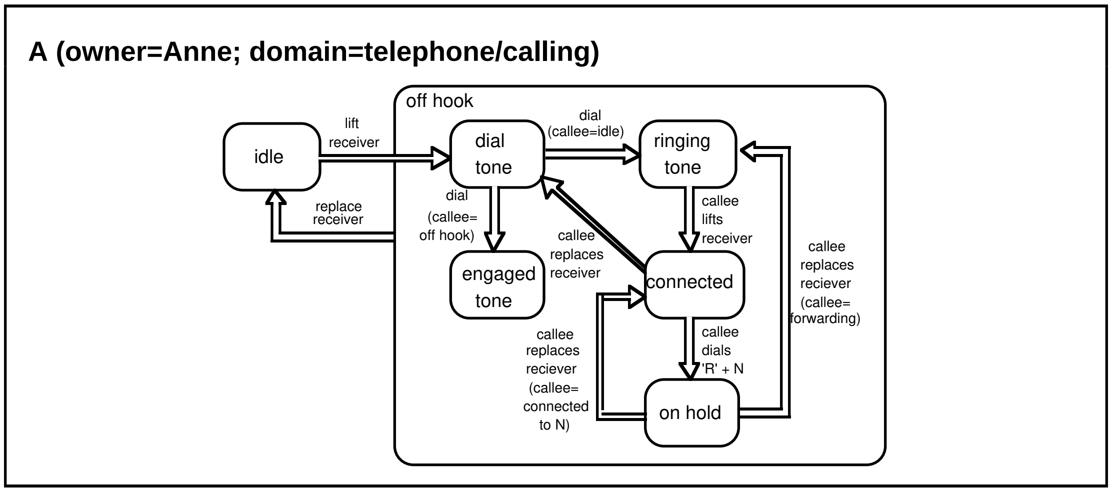
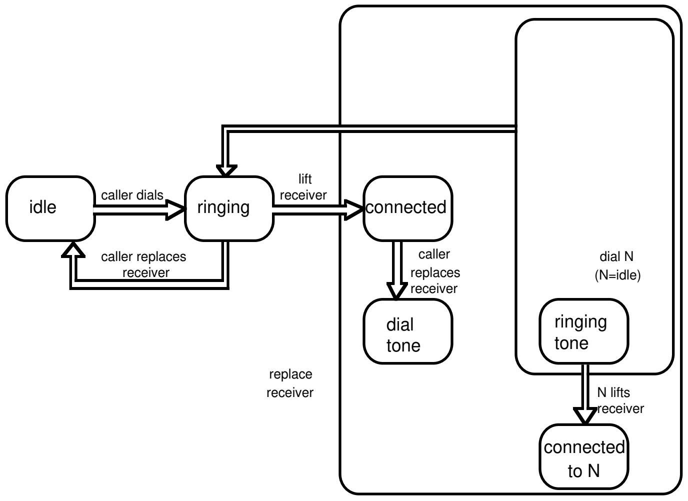


__Figure 4:__ Adding an ‘on hold’ state to Anne’s ViewPoint specification.

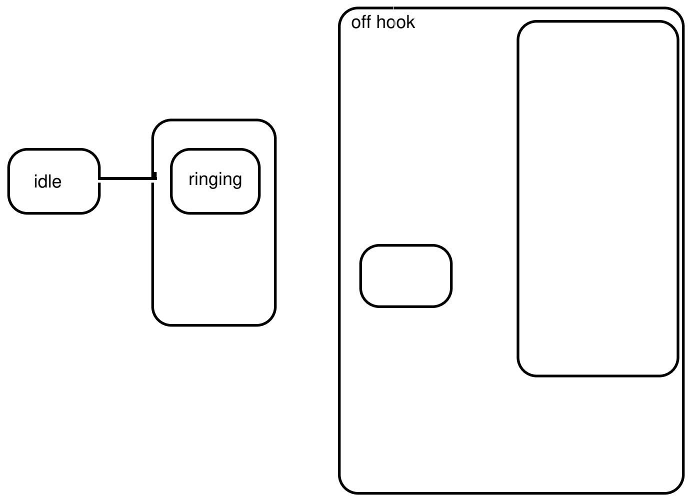
Bob’s changes are a little more complicated, as new states need to be added to represent the process of contacting the third party. The required behaviour for the callee is that pressing the ‘R’ button on the phone puts the calling party on hold, to enable the callee to dial and connect to the third party. If the callee replaces the receiver before a connection to a third party is established, the phone rings again; picking it up then reconnects to the original caller. If the callee replaces the receiver after connecting to a third party, the original call is forwarded to the third party, leaving the callee’s phone idle. This is shown in figure 5.

B (owner=Bob; domain=telephone/incoming call)



A further set of consistency rules will detect these conflicts at the next stage [step (d)] of the

B (owner=Bob; domain=telephone/incoming call)



R_3: $\forall\ VP_D(STD, D_a)$

$$\{\ (VP_S.state(X) \sim VP_D.state(Y)) \wedge VP_S.transition(X, \_) \rightarrow$$
$$VP_D.transition(Y, \_) \wedge corresponds(VP_S.transition(X, \_), VP_D.transition(Y, \_))\ \}$$

where this rule applies to two ViewPoints of any domain, D_a. The application of this rule will detect that Anne still has a ‘callee replaces receiver’ transition from ‘connected’, and add the predicate:

missing(transition(connected, dial tone), B.transition(connected, _), R_3)

to the list of inconsistencies in Anne’s ViewPoint. Should the inconsistency be explored, the suggested actions will include adding the missing transition to Bob’s ViewPoint, linking one of Bob’s existing transitions to Anne’s transition, or deleting Anne’s transition. Under normal circumstances, the default action would be to add the transition to Bob’s ViewPoint, due to the under-specification assumption mentioned earlier. However, in this case, there is more information available. A transition that matches the required pattern did once exist in Bob’s ViewPoint, but was deleted:

transition(connected, idle).name.‘replace receiver’

The implication, therefore, is that the default action should be to delete the corresponding transition in Anne’s ViewPoint. This is in fact the action that Anne chooses to perform.

5. Discussion

Incremental exploration and resolution of the inconsistencies reveals an important mismatch between the conceptual models held by the two participants described in our scenario: they had different conceptions of how telephone connections are terminated, and hence whether there is any difference in being connected as a caller and connected as a callee. Although it is entirely possible that this mismatch may have been detected anyway, the explicit conflict resolution process provides a focus for identifying these kinds of mismatch.

The process of defining the required behaviour of a device is crucial to requirements specification. Various tools exist for defining and analysing behavioural specifications, including, to some extent, determination of completeness and consistency. However, no such analysis can guarantee that the behaviour that gets specified is the intended one. Animating a behavioural specification can also help by bringing the specified behaviour to the attention of the analyst. Analysis of conflicts in the way described here is clearly an additional help.

We have demonstrated how conflicts between the conceptual models used by the two participants can be detected through the identification of inconsistencies. It is worthwhile clarifying the distinction between conflict and inconsistency. An *inconsistency* occurs if a rule has been broken. Such rules are defined by method designers, to specify the correct use of methods. Hence, what constitutes an inconsistency in any particular situation is entirely dependent on the rules defined during the method design. Rules will cover the correct use of a notation, and the relationships between different notations.

We define *conflict* as the interference in the goals of one party caused by the actions of another party (Easterbrook, et al., 1993). For example, if one person makes changes to a specification which interfere with the developments another person was planning to make, then there is a conflict. This does not necessarily imply that any consistency rules have been broken.

An inconsistency might equally well be the result of a mistake. We define a *mistake* as an action that would be acknowledged as an error by the perpetrator of the action; some effort may be required, however, to persuade the perpetrator to identify and acknowledge a mistake.

Although our approach is based on the management of *inconsistency*, our scenario has shown how this in turn helps with the identification and resolution of *conflicts* and *mistakes*. There remains the possibility that some conflicts and mistakes will not manifest themselves as inconsistencies.

Finally, there is at least one conflict between the ViewPoints in the scenario which has not been detected by the set of consistency rules we outlined. Consider what would happen in figures 7 and 8 if the callee is in any of the forwarding states, and the caller (who is on hold) replaces the receiver. Anne's ViewPoint is clear about the behaviour: the connection is terminated. However, Bob does not take account of this possibility. An obvious resolution would be for Bob to add a transition from 'forwarding' to 'dial tone' to account for this action, although it is not clear this is the desired behaviour once the callee has dialled a forwarding number. This conflict may require further consideration to find a satisfactory resolution.

That this conflict is not detected is a weakness in the set of consistency rules that we presented, rather than a problem with our approach. The consistency rules arise from: consideration of the rationale and operation of the method; consideration of examples and case studies of the use of the method; and from the experiences of the method in use. If it becomes clear that some types of mistakes and conflicts are not being detected, then new consistency rules should be added. In the example above, a new rule would need to be added to the set of rules for checking relationships between devices with associated behaviours. Moreover, the user-defined relationships described in sections 4.8 and 4.9 illustrate how domain-specific relationships may be recognised and defined dynamically as the method is used.

- I pl e entation

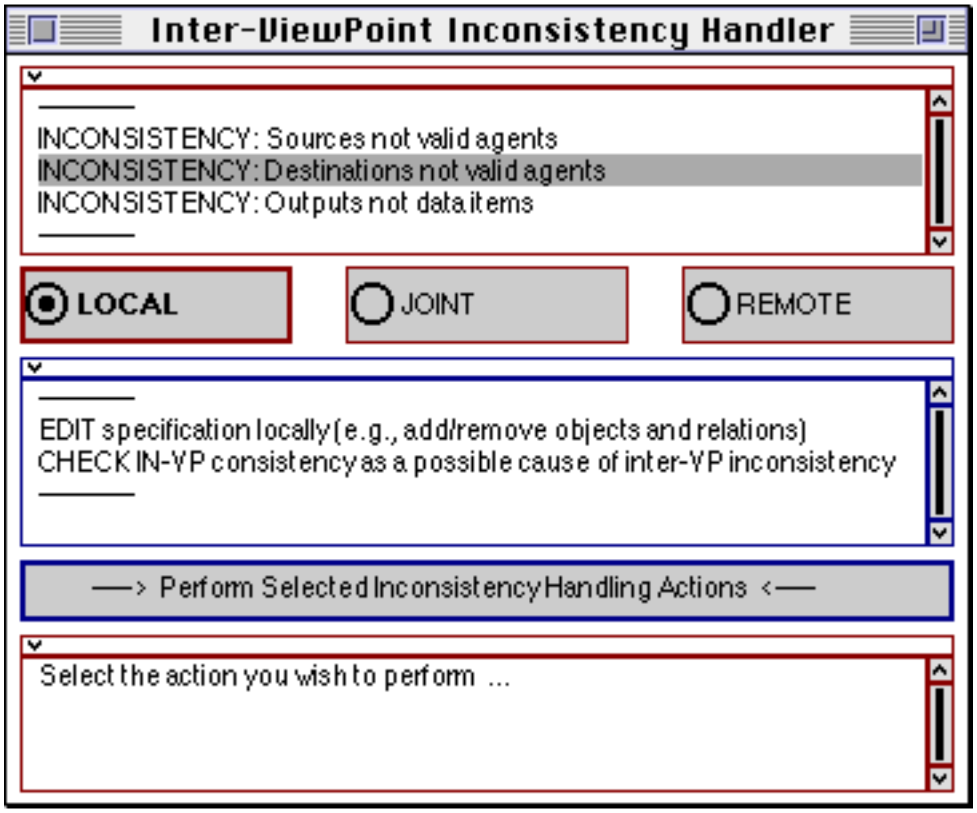
A prototype computer-based environment and associated tools (*The Viewer*

n006 T3- rule1this conflict i9

w'0.068ed, enario whicnshi 0.06

We have also extended *The Viewer* to support a subset of the inconsistency management tools described in this paper. A Consistency Checker allows users to invoke and apply selected in- and inter-ViewPoint consistency rules, and record the results of all such consistency checks in the appropriate ViewPoint's work record. A prototype Inconsistency Handler has also been implemented, to illustrate the kind and scope of inconsistency management we expect tool support to provide (figure 9).

A number of inter-ViewPoint consistency checking and inconsistency handling issues that arise from distributed and/or concurrent development in this setting have yet to be explored. Moreover, combining inconsistency handling with the notion of development guidance still requires further work. We plan to incorporate many of the conflict resolution strategies and actions within *The Viewer*, while tolerating inconsistency.


Inter-ViewPoint Inconsistency Handler
LOCAL
JOINT
REMOTE
—> Perform Selected Inconsistency Handling Actions <—

INCONSISTENCY: Sources not valid agents
INCONSISTENCY: Destinations not valid agents
INCONSISTENCY: Outputs not data items

EDIT specification locally (e.g., add/remove objects and relations)
CHECK IN-VP consistency as a possible cause of inter-VP inconsistency

Select the action you wish to perform ...

Figure 9: *The user-interface of an inter-ViewPoint Inconsistency Handler provided by The Viewer. A list of broken consistency rules is shown in the top pane, and a list of inconsistency handling actions for any selected rule is shown in the middle pane. These actions may be "local" to the source ViewPoint initiating the checks (e.g., local editing actions); "remote" actions performed by the destination ViewPoint; or "joint" actions (e.g., negotiation) performed by both ViewPoints involved in the check.*

- Conclusions and Future Work

ViewPoints facilitate separation of concerns and the partitioning of software development knowledge. Partitioning is only useful if relationships and dependencies between partitions can be defined. In this paper, we have shown how such relationships can be defined as part of a method. We have demonstrated how inconsistencies identified by checking these relationships may be resolved, and illustrated how subsequent evolution affects a resolution. Resolutions are recorded so that the effects of subsequent changes may be tracked.

We have also shown how re-negotiation may be supported. Analysis of inconsistency helps reveal the conceptual models used and assumptions made by development participants. In this way, the explicit resolution process acts as an elicitation tool. The ability to identify mismatches in conceptual models is an important benefit to requirements engineers adopting this approach.

The detection of conflicts and other problems (e.g., mistakes) depends on how well a method is defined. We have suggested how conflicts can arise that do not result in inconsistencies, as they do not break any of the defined relationships. Moreover, method design is an iterative process in which experience with method use can help improve the method. In this way, experience in using a method may lead to new types of consistency rules being added to the method.

Identifying consistency relationships, checking consistency and resolving conflicts are all important steps in managing inconsistency in an evolving specification. Our approach makes a contribution to multi-perspective software development in general, and requirements specification in particular by using inconsistency management to elicit knowledge about systems and their domain.

Further formalisation of the ViewPoints framework and notions of inconsistency is required in order to provide better tool support for inconsistency handling in general, and reasoning in the presence of inconsistency in particular. Moreover, the dependencies between elements of an evolving specification require further investigation (Pohl, 1994). While we have demonstrated how such domain-specific relationships may elicited, expressed, recorded and tracked, further examples and case studies are needed in order to validate our approach.

- Acknowledgements

We would like to acknowledge the contributions and feedback of Anthony Finkelstein, Tony Hunter and Jeff Kramer. Special thanks to Martin.084wed feeIro3reRs4slrEd review of earlid fdrafts of the paper. This work was partly funded by the UK DTI as part of the ESF project (IED4/410/36/002), the European Union as part of the PROMOTER BRA and the ISI project (ECAUS003), and the UK EPSRC as part of the VOILA project (GR/J15483). An earlid fversion of the paper appeared in (Easterbrook & Nuseibeh, 1995).

- References

Ainsworth, M., Cruickshank, A. H., Groves, L. G., & Wallis, P. J. L. (1994). Viewpoint Specification and Z. *Information and Software Technology*, *36*(1).

Alford, M. (1994). Attacking Requirements Complexity Using a Separation of Concerns. In *Proceedings of 1st International Conference on Requirements Engineering*, (pp. 2-5). Colorado Springs, Colorado, USA: IEEE Computer Society Press.

Balzer, R. (1991). Tolerating Inconsistency. In *Proceedings of 13th International Conference on Software Engineering (ICSE-13)*, (pp. 158-165). Austin, Texas, USA: IEEE Computer Society Press.

Besnard, P., & Hunter, A. (1994). *Quasi-Classical Logic: Non-trivializable classical reasoning from inconsistent information* (Technical Report No. Department of Computing, Imperial College, London, UK.

Easterbrook, S. (1991). Resolving Conflicts Between Domain Descriptions with Computer-Supported Negotiation. *Knowledge Acquisition: An International Journal*, *3*, 255-289.

Easterbrook, S. (1993). Domain Modelling with Hierarchies of Alternative Viewpoints. In *Proceedings of International Symposium on Requirements Engineering (RE '93)*, (pp. 65-72). San Diego, CA, USA: IEEE Computer Society Press.

Easterbrook, S., Beck, E. E., Goodlet, J. S., Plowman, L., Sharples, M., & Wood, C. C. (1993). A Survey of Empirical Studies of Conflict. In S. M. Easterbrook (Eds.), *CSCW: Co-operation or Conflict?* (pp. 1-68). London: Springer-Verlag.

Easterbrook, S., Finkelstein, A., Kramer, J., & Nuseibeh, B. (1994). Coordinating Distributed ViewPoints: The Anatomy of a Consistency Check. *Concurrent Engineering: Research and Applications*, *2*(3).

Easterbrook, S. M., & Nuseibeh, B. A. (1995). Managing Inconsistencies in an Evolving Specification. In *Second IEEE Symposium on Requirements Engineering*, . York, UK:

Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., & Nuseibeh, B. (1994a). Inconsistency Handling in Multi-Perspective Specifications. *Transactions on Software Engineering*, *20*(8), 569-578.

Finkelstein, A., Kramer, J., & Nuseibeh, B. (Ed.). (1994b). *Software Process Modelling and Technology*. Somerset, UK: Research Studies Press Ltd. (Wiley).

Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., & Goedicke, M. (1992). Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, *2*(1), 31-58.

Gabbay, D., & Hunter, A. (1991). Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning, Part 1 - A Position Paper. In *Proceedings of the Fundamentals of Artificial Intelligence Research '91*, 535 (pp. 19-32). Springer-Verlag.

Ghezzi, C., Jazayeri, M., & Mandrioli, D. (1991). *Fundamentals of Software Engineering*. Engelwood Cliffs, New Jersey, USA: Prentice-Hall, Inc.

Greenspan, S., & Feblowitz, M. (1993). Requirements Engineering Using the SOS Paradigm. In *Proceedings of International Symposium on Requirements Engineering (RE '93)*, (pp. 260-263). San Diego, CA, USA: IEEE Computer Society Press.

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, *8*, 231-74.

Kotonya, G., & Sommerville, I. (1992). Viewpoints for Requirements Definition. *Software Engineering Journal*, *7*(6), 375-387.

Leite, J. C. S. P., & Freeman, P. A. (1991). Requirements Validation Through Viewpoint Resolution. *Transactions on Software Engineering*, *12*(12), 1253-1269.