Managing Inconsistencies in an Evolving Specification

Steve Easterbrook

School of Cognitive and Computing Sciences,
University of Sussex, Brighton, BN1 9QH, UK.
easterbrook@cogs.susx.ac.uk

Bashar Nuseibeh

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

Abstract

In an evolving specification, considerable effort is spent handling recurrent inconsistencies. Detecting and resolving inconsistencies is only part of the problem: a resolved inconsistency might not stay resolved. Frameworks in which inconsistency is tolerated help by allowing resolution to be delayed. However, evolution of a specification may affect both resolved and unresolved inconsistencies. We address these problems by explicitly recording relationships between partial specifications (ViewPoints), representing both resolved and unresolved inconsistencies. We assume that ViewPoints will often be inconsistent with one another, and we ensure that a complete work record is kept, detailing any inconsistencies that have been detected, and what actions, if any, have been taken to resolve them. The work record is then used to reason about the effects of subsequent changes to ViewPoints, without constraining the development process.

1. Introduction

In an evolving specification, considerable development time and effort is spent handling recurrent inconsistencies. Such inconsistencies are particularly prevalent during requirements engineering, when conflicting and contradictory objectives are often required by different stakeholders. Tools and techniques for detecting and resolving inconsistencies only address part of the problem: they do not ensure that a resolution generated at a particular stage will apply at all subsequent stages of the process.

In this paper, we propose an approach for managing inconsistencies that arise during the development of multi-perspective specifications, by explicitly recording consistency relationships between partial specifications, and by representing both resolved and unresolved inconsistencies. We use the ViewPoints framework for multi-perspective software development as a vehicle for demonstrating our approach, and illustrate our techniques by working through an example drawn from the behavioural specification of a telephone.

2. The ViewPoints Framework

We base this 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. [9] provides a fuller account of the framework, and [7] gives an introduction to the issues of inconsistency management.

A ViewPoint can be thought of as an 'actor', 'role', or 'knowledge source' in the development process, combined with a 'view' or 'perspective' which an actor maintains. In software terms, ViewPoints are loosely coupled, locally managed, distributable objects which encapsulate partial knowledge about a system and its domain, specified in a particular, suitable representation scheme, and partial knowledge of the process of development.

Each ViewPoint has the following slots:

- a representation *style*, the scheme and notation by which the ViewPoint expresses what it can see;
- a *domain*, which defines the area of concern addressed by the ViewPoint;
- a *specification*, the statements expressed in the ViewPoint's style describing the domain;
- a *work plan*, which comprises the set of actions by which the specification can be built, and a process model to guide application of these actions;
- a *work record*, which contains an annotated history of actions performed on the ViewPoint.

The development participant associated with a ViewPoint is the ViewPoint 'owner'. The owner is responsible for developing a specification using the notation defined in the style slot, following the strategy defined by the work plan, for a particular problem domain. A development history is maintained in the work record.

This framework encourages multiple representations, and is a deliberate move away from attempts to develop monolithic specification languages. It is independent from any particular software development method. In general, a method comprises of a number of different notations, with rules about when and how to use each notation. A method can be implemented in the framework by defining a set of

ViewPoint templates, which together describe the set of notations provided by the method, and the rules by which they are used independently and together.

The notion of a viewpoint was first introduced as part of requirements specification methods such as Structured Analysis [22] and CORE [17], and more recently deployed for validating requirements [16], domain modelling [5] and service-oriented specification [12, 14]. In our framework, we use ViewPoints to organise multi-perspective software development in general, and to manage inconsistency.

3. Inconsistency Management

In our framework, there is no requirement for changes to one ViewPoint to be consistent with other ViewPoints [8]. Hence, inconsistencies are tolerated throughout the software development process. This contrasts with many existing support environments which enforce consistency, for example by disallowing changes to a specification that lead to inconsistencies.

We view strict enforcement of consistency throughout the requirements process as unnecessarily restrictive. Partly this view arises from a consideration of the distributed nature of software development: it may not always be possible to check that particular changes are consistent with work in progress at another site. Consistency enforcement can also stifle innovation, causing premature commitment and preventing exploration of alternatives [15]. Finally, development participants are likely to have conflicting views about many aspects of the requirements, and exploration of these conflicts are greatly facilitated by the ability to express the alternative views.

The ability to express and reason with inconsistent specifications during software development overcomes many of these problems. However, we assume that eventually a consistent specification will be required as the basis for an implementation[1]. We therefore focus on the management of inconsistencies, so that the specification process remains a coordinated effort. Consistency checking and resolution can be delayed until the appropriate point in the process. As there is no requirement for inconsistencies to be resolved as soon as they are discovered, consistency checking can be separated from resolution.

In order to manage inconsistencies, the relationships between ViewPoints need to be clearly defined. In general, the relationships arise from deploying the software development method. For example, if a method involves hierarchical decomposition of a particular type of diagram, then two diagrams that are hierarchically related should obey certain rules. Similarly, a method which provides several notations will specify how those notations inter-relate. Thus, the possible relationships between ViewPoints are determined by the method.

Consistency checking is performed by applying rules, defined by the method, which express the relationships that should hold between particular ViewPoints [21]. The 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 [20].

The need to tolerate inconsistency has been recognised in a variety of areas, including configuration management [23], programming [3], logical databases [10] and collaborative development [18]. In [7], we discuss how co-ordination between ViewPoints can be supported without requiring consistency to be maintained. A key problem is to support resolution of inconsistencies in an incremental fashion, so that resolutions are not lost when the ViewPoints continue to evolve. We now present a scenario to illustrate how this process is supported.

4. Scenario

Our scenario involves the behavioural specification of a telephone. We assume the existence of a method which allows such specifications to be partitioned into separate ViewPoints. We begin by outlining the salient features of the method, before introducing the scenario.

4.1. The method

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 [11], and requirements complexity in particular [2]. It does, however, require corresponding techniques to combine resultant partial specifications, such as composition [25] and amalgamation [1].

By describing the behaviour of a telephone as two separate partial specifications, we can concentrate on different subsets of behaviours, and hence clarify how those subsets interact. In this way, we can, for example, analyse problems such as “feature interaction” in telephone systems [24].

The scenario concentrates on two analysts co-operating to build a description of the various states that a single telephone handset can be in. The analysts choose to partition this task such that one of them describes the

1 We will ignore the question of whether inconsistencies in a final specification or an eventual product are acceptable under some circumstances.

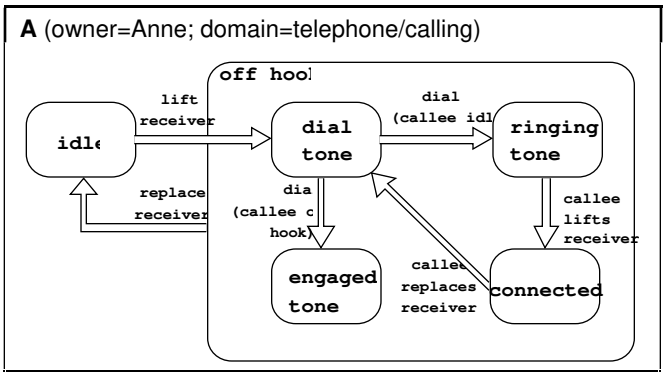

Figure 1: Anne's initial ViewPoint specification for making a call.

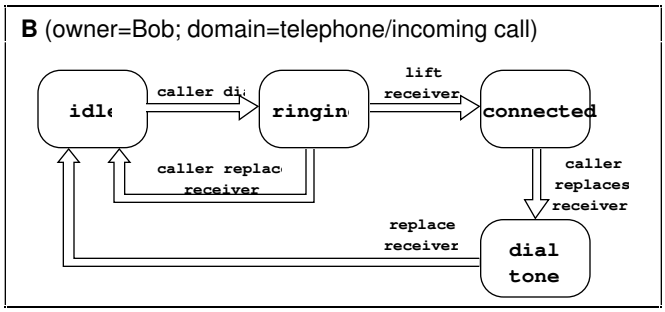

Figure 2: Bob's initial ViewPoint specification for receiving a call. Note that this specification is incomplete, as Bob has not yet considered what would happen if the callee replaces the receiver when in the connected state.

states involved when the handset is being used to make a call, and the other describes the states involved when the handset is receiving a call. There is an implicit 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 notation for expressing states and transitions diagrammatically. The state transition notation includes extensions for expressing super-states and sub-states[2].
- A partitioning step which 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.
- 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 problems; even though the checking process does not require such a merge to take place.

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.

4.2. Preliminary 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, a number of consistency relationships must hold between their ViewPoints.

2 We use Harel's extended state transition diagrams [13]. The extensions include the use of super and sub-ordinate states, as illustrated in figure 1. Transitions out of super-states are available from all sub-ordinate states. The notation also allows transitions to be a function not just of a stimulus, but of the truth of a condition. Conditions are shown in brackets after the name of the stimulus.

At this stage, Anne and Bob may wish to check whether their ViewPoints are consistent with one another. They do not yet attempt to analyse the interaction between a calling and receiving telephone: they only wish to check that the subset of behaviours described in each ViewPoint are consistent. In particular, the ViewPoints are likely to have some overlap, and these overlaps need checking. In this scenario, both ViewPoints include states such as 'idle', 'connected' and 'dial tone'.

We will consider two consistency rules in more detail:

(i) "If a transition between two states is described in one ViewPoint, and both states are described in the second ViewPoint, then the transition should also be described in the second ViewPoint".

In the example above, there is a 'replace receiver' transition between 'connected' and 'idle' in Anne's ViewPoint (inherited from the super-state 'off hook'), but not in Bob's. Although the partitioning method allows states to be missed out in different ViewPoints, if two states are included, all possible transitions between them should be shown. In this example, Bob's ViewPoint implies that replacing the receiver while connected does not return the phone to idle. Indeed, this is the actual behaviour of many telephone systems for incoming calls.

(ii) "If a state is shown as belonging to a super-state in one ViewPoint, and the same state is included in the second ViewPoint, then the super-state must also be included in the second ViewPoint".

This rule is to ensure no ambiguity: the 'connected' state is part of the 'off hook' super-state as defined in Anne's ViewPoint, but it is not clear which other states of Bob's ViewPoint are also members of 'off hook'.

4.3. Support for consistency checking

The consistency checking process described above is supported in each ViewPoint by providing consistency rules which may be invoked by the ViewPoint owner. The rules are defined by the method designer. We have developed a notation (presented in a simplified form below)

for expressing the rules as relationships between objects in the specifications of a source ViewPoint VP_S, (from which the rule is invoked), and a destination ViewPoint, VP_D. For example, the first consistency rule above would be expressed in each ViewPoint as:

R_1: $\forall\, VP_D(STD, D_S)$
$\{ VP_S.transition(X, Y) \wedge VP_D.state(X) \wedge VP_D.state(Y) \rightarrow VP_D.transition(X, Y) \}$

Briefly, the rule has three parts: a *label* by which it can be referred (R_1); a *quantifier* defining the possible destination ViewPoints for which the relationship should hold (in this case, all ViewPoints containing state transition diagrams, STD, whose domain, D_S, is the same as the current ViewPoint); and a *relationship* (in this case the existence of a transition in the source ViewPoint and the two states to which it is connected in the destination ViewPoint *entails* the existence of the transition in the destination ViewPoint).

There also is an entry in the ViewPoint's process model, defining circumstances under which the rule is applicable, and the possible results of applying it. Entries in the process model are expressed in the form:

{preconditions} ⇒ [agent, action] {postconditions}

For rule R_1, the following entry has been defined:

$\{\ \} \Rightarrow [VP_S, R_1]$
$\{\Re_1(transition(X, Y), VP_D.transition(X, Y))\} \cup$
$\{missing(transition(X, Y), VP_D.transition(X, Y), R_1)\}$

In this case the preconditions are empty, indicating the rule can be applied at any time. The action is the application of rule R_1 by the source ViewPoint, VP_S. The result is a set of predicates describing the facts that have been established. Predicates of the form $\Re_i(\sigma, \psi.\delta)$ indicate that the relationship defined by the rule R_i holds for the partial specifications σ in the source ViewPoint and δ in the ViewPoint ψ. Predicates of the form $missing(\sigma, \psi.ps_D, R_i)$ indicate that no items matching partial specification ps_D in the destination ViewPoint ψ were found to meet the existence criteria associated with partial specification σ as required in rule R_i.

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

missing(transition(off hook, idle),
B.transition(connected, idle), R_1)

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 work record for Anne's ViewPoint. Normally, ViewPoint B is also notified of the results of the check.

3 We have assumed that inheritance of transitions from super-states, which is a part of the notation, is handled by the process of matching partial specifications given in a rule with the actual contents of a ViewPoint.

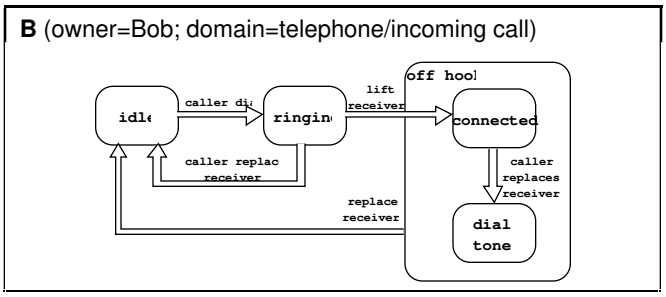

Figure 3: Bob's ViewPoint specification after resolution.

4.4. Resolution of inconsistencies

Anne and Bob consider the inconsistency resulting from the application of rule R_1 above. It reveals a conflict between their notion of the 'connected' state. Bob had assumed that if the callee replaces the receiver it does not sever the connection, and his ViewPoint is correct given that assumption. A possible resolution would be to distinguish the 'connected' state in each ViewPoint as different – being connected as a caller is different from being connected as callee. However, they cannot agree on this, and decide to delay resolution.

At a later point, they then consider the inconsistency resulting from the application of rule R_2. An obvious resolution is to add the 'off hook' super-state to Bob's ViewPoint. Bob does this, and his ViewPoint then contains the specification shown in figure 3.

This resolution accidentally resolves the inconsistency resulting from the application of rule R_1 as well, in that the connected state in Bob's ViewPoint now inherits the 'replace receiver' transition from the super-state. This side effect contradicts Bob's assumption that when the callee replaces the receiver it does not end the connection. However, he does not notice the side effect at this stage.

4.5. Support for resolving inconsistencies

When the consistency checking rules were invoked, the results were recorded as part of the ViewPoints' work records. This provides some basic historical information on which to base a resolution, and is available whenever ViewPoint owners wish to tackle the inconsistencies.

The method provides a number of actions for each consistency rule, which may be applied if the rule is broken. Some of the actions will repair the inconsistency, others may just take steps towards a resolution, for instance by eliciting further information or performing some analysis [operationally: the actions are available to the ViewPoint owners as a menu, and each action has a short text giving the rationale for that action].

Consider the inconsistency resulting from the application of rule R_1 above. Anne and Bob both have available the 'missing' predicate described in section 4.3.

ViewPoint A actions:
(1) delete transition(off hook, idle)
(2) move state(connected) so it is no longer part of state(off hook)
(3) move transition(off hook, idle) so it no longer connects from state(off hook)
(4) delete state(connected)
(5) delete state(idle)
(6) rename state(connected)
(7) rename state(idle)
(8) devolve transition(off hook, idle) to all sub-states of off hook
ViewPoint B actions:
(9) delete state(connected)
(10) delete state(idle)
(11) rename state(connected)
(12) rename state(idle)
Joint Actions:
(13) copy transition(off hook, idle) from ViewPoint A to ViewPoint B as transition(connected, idle)

Table 1: Possible resolution actions for rule R_1

They also have the list of suggested actions for tackling the resolution. The available actions are given in table 1.

Some actions are derived directly from the rule that failed. These include removing items that make the rule hold, or adding items required by the rule. For example, action (13) is suggested because under-specification may be the cause of the problem, which can be dealt with by transferring material from one ViewPoint to another.

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 a rule. They may also have resulted from the experience of method users in the past: methods evolve as lessons are learnt about their use.

Further suggested actions are derived using method-specific heuristics. For example, action (8) is derived from a heuristic which suggests that an alternative to deleting a transition from a super-state is to devolve the transition to the sub-states. This action will not resolve the inconsistency, but it may take ViewPoint owners a step closer to finding a resolution.

In addition to the suggested actions, ViewPoint owners always have the option of ignoring an inconsistency, or invoking a tool to analyse it further by, for example, displaying portions of the ViewPoints side-by-side and exploring the differences between them [4]. If they choose to ignore the inconsistency, they may wish to first perform some steps towards resolution, either by applying actions which don't quite resolve the inconsistency, or by eliminating some of the suggested actions as undesirable. Any such steps performed in the context of resolving a particular inconsistency are recorded in the appropriate ViewPoint work record, so that the process may be continued at a later point.

Each ViewPoint maintains a list of unresolved inconsistencies. The list only contains those that have been detected - there may always be others for which relevant rules have not been applied. Subsequent changes to a ViewPoint are checked to see if they affect any of the known inconsistencies. This process can be illustrated by considering what happens when an inconsistency resulting from the application of rule R_2 is resolved:

- Anne's ViewPoint represents the inconsistency as:
  missing(state(off hook), B. state(off hook), R_1)
- Among the actions suggested for its resolution are that 'off hook' be added to Bob's ViewPoint.
- Anne selects this action, as a suggested resolution for Bob to carry out. Bob agrees and so adds the new state.
- An entry is added to each ViewPoint's work record to record that the action resolved the inconsistency.
- As part of the resolution, the transition from 'off hook' to 'idle' is also copied to Bob's ViewPoint.
- The actions are checked for their effect on other inconsistencies. These checks are only performed locally: each ViewPoint only checks its own actions against its own list of consistency rules. In this case, the new transition in Bob's ViewPoint is likely to repair the inconsistency:
  A.missing(transition(off hook, idle),
  B. transition(connected, idle), A.R_1)
  This fact is recorded in Bob's work record, but it is not immediately flagged to Bob, as there may be a large number of such effects.
- Anne's rule R_2 is re-applied to check that the inconsistency is indeed resolved.

Note that the rule R_1 is not re-applied automatically, despite the evidence 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 only concerns the inconsistency from rule R_2. Any effect on other inconsistencies are dealt with when the ViewPoint owners specifically consider them.

4.6. 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.

Bob's changes are more complicated, as new states are needed 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

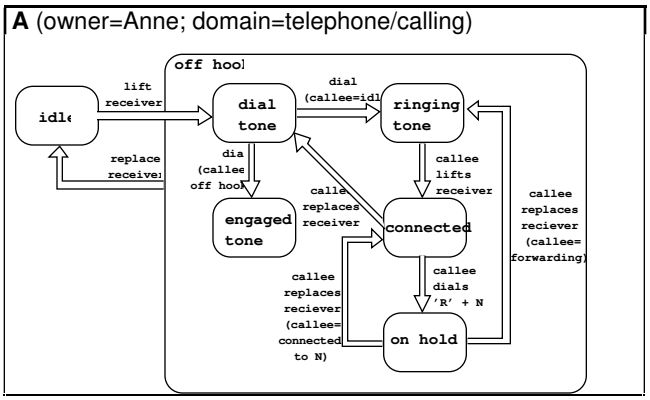

Figure 4: Adding an 'on hold' state to Anne's ViewPoint specification.

replaces the receiver after connecting to a third party, the original call is forwarded to the third party, leaving the callee's phone idle (figure 5).

At this point Bob realises that one of the reasons he has distinguished between 'connected' and 'connected to N' is because replacing the receiver has a different result in each case. In the 'connected' state, replacing the receiver does not disconnect the incoming call. In the 'connected to N' state, replacing the receiver completes the forward operation, leaving the phone idle. He notices that when he added the super-state 'off hook', he inadvertently gave all the off-hook states the transition to idle when the receiver is replaced. He now corrects this error (figure 6).

This now reintroduces an inconsistency from rule R_1, as Bob no longer has a transition from connected to idle. Because this latest change affects a previous resolution the support tools will suggest to Bob that he re-checks R_1 at some point. When Bob checks this rule he discovers his ViewPoint is inconsistent with Anne's. He realises that the only resolution he will be happy with is to rename his connected state to distinguish it from Anne's connected state. This resolves the inconsistency once more.

4.7. Support for monitoring inconsistencies

Throughout these elaborations, each action is checked for its effect on the known inconsistencies in each ViewPoint, whether or not they were resolved. In the scenario, only two inconsistencies were detected, as we only applied two consistency rules. Both were resolved, and annotated with the action that resolved them.

Although the list of unresolved inconsistencies is empty, this does not mean the ViewPoints are consistent. For example, if rule R_1 were applied after the elaboration above, an inconsistency between the states labelled 'ringing tone' in each ViewPoint would be detected: the transition 'replace receiver' has a different destination in each case. The same applies to 'engaged tone'. These inconsistencies will be detected next time R_1 is applied, but having applied a rule in the past is no guarantee that the relationship expressed in the rule still holds.

Now consider what happens when Bob deletes the transition from 'off hook' to 'idle'. As the addition of this transition resolved the inconsistency arising from rule R_1, its deletion may re-introduce the inconsistency. When the list of past inconsistencies is examined, this possibility is detected, and the ViewPoint owner, Bob, will be warned. He may ignore the warning (inconsistencies are tolerated), or he can choose to check whether or not the inconsistency has indeed re-appeared, by invoking rule R_1 again. If he does this, there are again two possibilities:

- The inconsistency does not re-appear. In this case some other action may have had an affect. The inconsistency is annotated to indicated that it was resolved by some

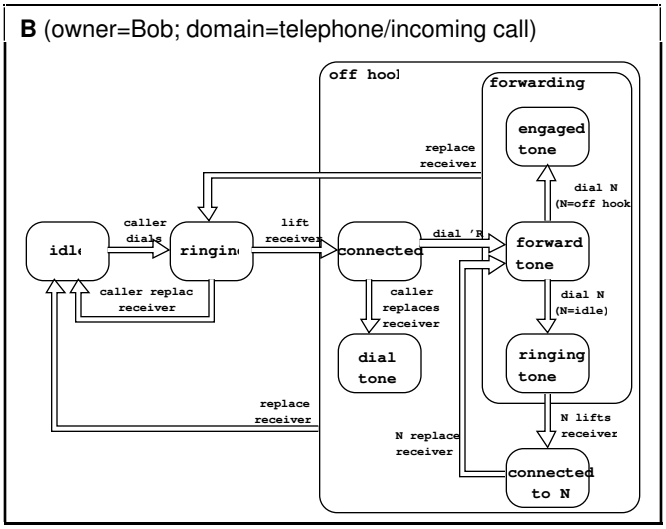

Figure 5: Extending Bob's ViewPoint specification to handle call forwarding.

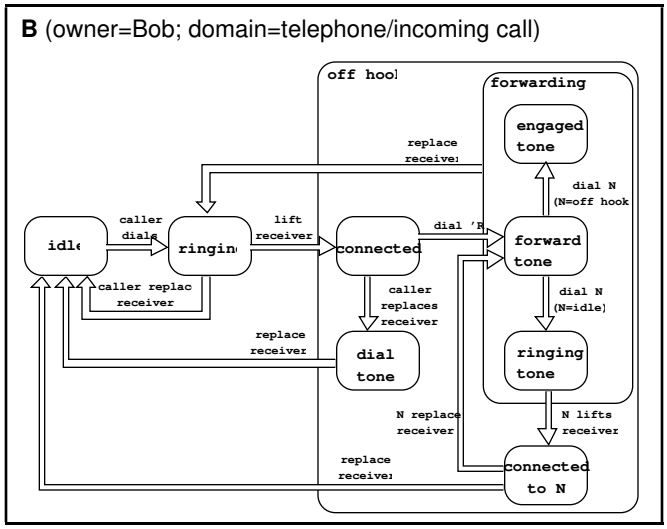

Figure 6: Replacing the receiver only returns the phone to an 'idle' state if there is a 'dial tone' or 'connected to N'.

unknown action between the original resolution and the current action.

- The inconsistency re-appears, as is the case in our scenario. Here, the inconsistency is marked as unresolved, and annotated to show which actions resolved and re-introduced it. This allows ViewPoint owners to further eliminate suggested resolution actions, if they have been tried and found to be unsatisfactory.

4.8. Discussion

Incremental exploration and resolution of the inconsistencies revealed an important mismatch between the conceptual models held by the two participants described in our scenario; namely about when connection are terminated, and whether there is a 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 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 help by bringing the specified behaviour to the attention of the analyst. Analysis of inconsistencies in the manner described here is clearly an additional help.

5. Undetected conflicts

We have demonstrated how conflicts between the conceptual models used by the two participants can be detected through 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 a method. Hence, what constitutes an inconsistency in any particular situation is entirely dependent on the rules defined during the method design. Such rules might 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 [6], typically where one person makes changes which interfere with the developments another person is making. This does not necessarily result in any consistency rules being 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, the scenario has shown how this in turn helps with the identification and resolution of conflicts, as well as mistakes. There remains the possibility that some conflicts and mistakes will not manifest themselves as inconsistencies.

The consistency rules arise from: consideration of the rationale and operation of the method; from consideration of examples and case studies of the use of the method; and from the experiences of method users. If it becomes clear that some types of mistakes and conflicts are not being detected, then new consistency rules can be added.

6. Implementation

A prototype development (*The Viewer*) has been constructed to support the framework [19]. *The Viewer* has two distinct modes of use: method design and method use. Method design involves the creation of ViewPoint templates which are ViewPoints for which only the representation style and work plan slots are filled. In method use, ViewPoints are instantiated from these templates, to represent the various perspectives. Each instantiated ViewPoint will inherit the knowledge necessary for building and manipulating a specification in the chosen notation, and cross checking consistency with other ViewPoints. Hence, each ViewPoint is a self-contained specification development tool.

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 in- and inter-ViewPoint consistency rules, and records the results of all such consistency checks in the appropriate ViewPoint's work record. A prototype Inconsistency Handler has also been implemented, to demonstrate the kind of inconsistency management we expect tool support to provide.

7. Conclusions

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 conflicts 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 suggested how conflicts can arise which do not give rise to inconsistencies. 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.

Acknowledgements

We would like to acknowledge the contributions and feedback of Anthony Finkelstein, Jeff Kramer, Martin Feather, and the anonymous reviewers. This work was partly funded by the UK DTI as part of the ESF project, and the UK EPSRC as part of the VOILA project.

References

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

[2] Alford, M. (1994) "Attacking Requirements Complexity Using a Separation of Concerns"; *Proc. 1st IEEE Conference on Requirements Engineering*, Colorado Springs, USA, 18-22nd April 1994, 2-5.

[3] Balzer, R. (1991) "Tolerating Inconsistency"; *Proc. 13th International Conference on Software Engineering (ICSE-13)*, Austin, Texas, 13-17th May 1991, 158-165; IEEE CS Press.

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

[5] Easterbrook, S. (1993) "Domain Modelling with Hierarchies of Alternative Viewpoints"; *Proc. IEEE Symposium on Requirements Engineering (RE '93)*, San Diego, USA, 4-6th Jan. 1993, 65-72.

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

[7] Easterbrook, S., A. Finkelstein, J. Kramer & B. Nuseibeh (1994) "Coordinating Distributed ViewPoints: The Anatomy of a Consistency Check"; *Concurrent Engineering: Research and Applications*, 2: 209-222.

[8] Finkelstein, A., D. Gabbay, A. Hunter, J. Kramer & B. Nuseibeh (1994) "Inconsistency Handling in Multi-Perspective Specifications"; *IEEE Trans. on Software Engineering*, 20(8): 569-578.

[9] Finkelstein, A., J. Kramer, B. Nuseibeh, L. Finkelstein & 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.

[10] Gabbay, D. & A. Hunter (1991) "Making Inconsistency Respectable: A Logical Framework for Inconsistency in Reasoning, Part 1 - A Position Paper"; *Proc. Fundamentals of Artificial Intelligence Research '91*, 19-32; LNCS 535, Springer-Verlag.

[11] Ghezzi, C., M. Jazayeri & D. Mandrioli (1991) *Fundamentals of Software Engineering*; Prentice-Hall.

[12] Greenspan, S. & M. Feblowitz (1993) "Requirements Engineering Using the SOS Paradigm"; *Proc. IEEE Symposium on Requirements Engineering (RE '93)*, San Diego, USA, 4-6th Jan. 1993, 260-263.

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

[14] Kotonya, G. & I. Sommerville (1992) "Viewpoints for Requirements Definition"; *IEE Software Engineering Journal*, 7(6): 375-387.

[15] Kramer, J. (1991) "CASE Support for the Software Process: A Research Viewpoint"; *Proc. Third European Software Engineering Conference*, Milan, Oct. 1991, 499-503; LNCS 550, Springer-Verlag.

[16] Leite, J. C. S. P. & P.A. Freeman (1991) "Requirements Validation Through Viewpoint Resolution"; *IEEE Trans. on Software Engineering*, 12(12): 1253-1269.

[17] Mullery, G. (1979) "CORE - a method for controlled requirements expression"; *Proc. 4th International Conference on Software Engineering (ICSE-4)*, 126-135; IEEE CS Press.

[18] Narayanaswamy, K. & N. Goldman (1992) ""Lazy" Consistency: A Basis for Cooperative Software Development"; *Proc. International Conference on Computer-Supported Cooperative Work (CSCW '92)*, Toronto, 31st Oct. - 4th Nov., 257-264.

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

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

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

[22] Ross, D. T. & K. E. Schoman (1977) "Structured Analysis for Requirements Definition"; *IEEE Transactions on Software Engineering*, 3(1): 6-15.

[23] Schwanke, R. W. & G. E. Kaiser (1988) "Living With Inconsistency in Large Systems"; *Proc. International Workshop on Software Version and Configuration Control*, Grassau, Germany, 27-29 Jan. 1988, 98-118.

[24] Zave, P. (1993) "Feature Interaction and Formal Specifications in Telecommunications"; *IEEE Computer*, 26(8): 20-30.

[25] Zave, P. & M. Jackson (1993) "Conjunction as Composition"; *ACM Trans. on Software Engineering and Methodology*, 2(4): 379-411.