Managing Inconsistencies in an Evolving Specification

STEVE EASTERBROOK
School of Cognitive & Computing Sciences
University of Sussex, Falmer, Brighton, BN1 9QH
easterbrook@cogs.susx.ac.uk

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

Abstract

In an evolving specification, considerable development time and effort is spent handling recurrent inconsistencies. 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 specification process. Previously, we have advocated tolerance and management of inconsistency, rather than strict enforcement of consistency. The advantages of this approach include the ability to delay resolution, facilitation of concurrent development, and greater flexibility in development strategies. However, this approach does not prevent inconsistencies themselves from evolving, and it does not ensure that resolved inconsistencies remain resolved throughout subsequent developments. 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. This work record is then used to reason about the effects of subsequent changes to the ViewPoints, without constraining the development process. We illustrate the approach through a case study.

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 development 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 specification 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.

We begin the paper with a brief review of the ViewPoints framework (section 2), and discuss some inconsistency management issues in this context (section 3). We then demonstrate our approach to inconsistency management within the framework via a scenario (section 4), and this is followed by a discussion of outstanding issues of terminology and undetected conflicts (section 5). We conclude the paper with an outline of our prototype implementation constructed to support our approach (section 6), and draw some conclusions.

2. The ViewPoints Framework

The framework upon which we base this work supports 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 reader is referred to [Finkelstein et al. 1992] for a fuller account of the framework, and to [Easterbrook et al. 1994] for an introduction to the issues of inconsistency management.

A ViewPoint can be thought of as a combination of the notion of an 'actor', 'knowledge source', 'role' or 'agent' in the development process, and the notion of 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 any particular ViewPoint is known as the ViewPoint 'owner'. The owner is responsible for developing a ViewPoint 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 actively encourages multiple representations, and is a deliberate move away from attempts to develop monolithic specification languages. It is also independent from any particular software development method. In general, a method is composed of a number of different development techniques. Each technique has its own notation and rules about when and how to use that notation. A software development 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 use of viewpoints in requirements engineering has been advocated by a number of researchers. The notion of a viewpoint was first introduced as part of requirements specification methods such as Structured Analysis [Ross & Schoman 1977] and CORE [Mullery 1979], and more recently deployed for validating requirements [Leite & Freeman 1991], domain modelling [Easterbrook 1993] and service-oriented specification [Kotonya & Sommerville 1992; Greenspan & Feblowitz 1993]. In our framework, we use structured ViewPoints to organise multi-perspective software development in general, and to manage inconsistency as we elaborate briefly below.

3. Inconsistency Management

In our framework, there is no requirement for changes to one ViewPoint to be consistent with other ViewPoints [Finkelstein et al. 1994]. Hence, inconsistencies are tolerated throughout the software development process. This is in contrast with many existing support environments which enforce consistency maintenance, for example by disallowing changes to a specification that lead to inconsistencies.

We view the 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 [Kramer 1991]. 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 are still required, but they 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 also specify how those notations should be used in combination, and how they interrelate. 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, Kramer & Finkelstein 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, Finkelstein & Kramer 1993].

An increasing number of researchers have recognised the need to tolerate inconsistency in a variety of settings. These include configuration management [Schwanke & Kaiser 1988], programming [Balzer 1991], logical databases [Gabbay & Hunter 1991] and collaborative development [Narayanaswamy & Goldman 1992]. In [Easterbrook et al. 1994], we discuss how co-ordination between ViewPoints can be supported without requiring consistency to be maintained. A key problem with tolerance of inconsistency 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 is drawn from the behavioural specification of a telephone, described using an extended state transition notation. We 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.

4.1. The method

Our method is based on the use of state transition diagrams. This notation is used 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, Jazayeri & Mandrioli 1991], and requirements complexity in particular [Alford 1994]. It does, however, require corresponding techniques to combine resultant partial specifications, such as composition [Zave & Jackson 1993] and amalgamation [Ainsworth et al. 1994].

Our scenario, describing the behaviour of a telephone as two separate partial specifications, allows us to 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 telecom systems [Zave 1993]. The scenario concentrates on how two instances of the same device interact under various circumstances.

---

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

For the purposes of the scenario, we deploy a method that exploits the ViewPoint framework to model different aspects of the overall behaviour of a device separately. Thus, our method provides the following:

- 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].
- 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. Essentially, 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.
- An analysis step which allows behaviours of one device to be associated with those of another, so that for example a transition in one device causes a transition in another.
- A further set of consistency checking rules which test whether devices whose transitions have been linked together exhibit consistent behaviours.

Our method also includes some 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). This separation of views is useful, because it allows them to consider the interaction between the caller and the callee. However, as they are both describing states of a single device, a number of consistency relationships must hold between their ViewPoints.

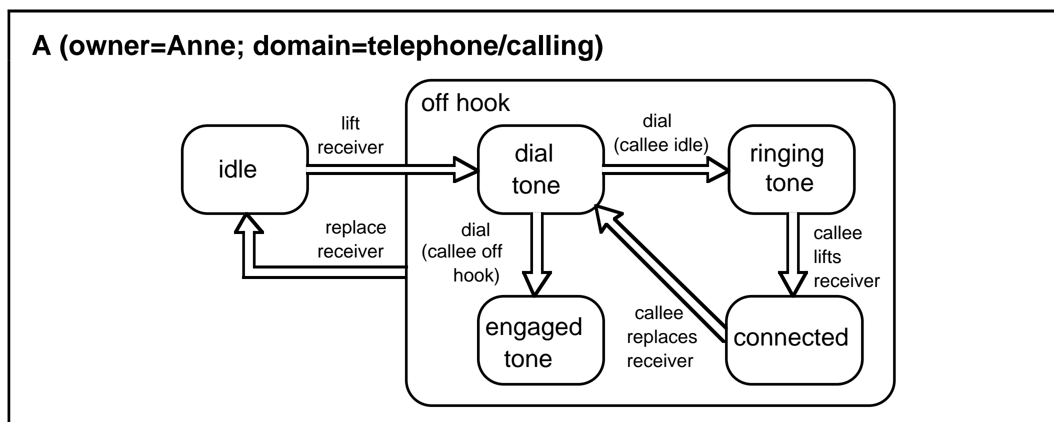

Figure 1: *Anne's initial ViewPoint specification for making a call.*

2 We use the extensions to state transition diagrams proposed by [Harel 1987]. These extensions include the use of super and sub-ordinate states, as illustrated in figure 1. Note that 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.

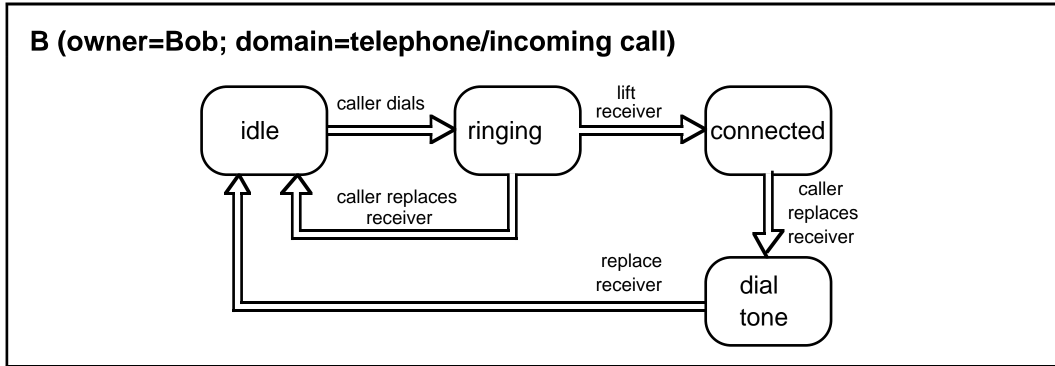

Figure 2: *Bob's initial ViewPoint specification for receiving a call.*

At this stage, Anne and Bob may wish to check whether or not their ViewPoints are consistent with one another. They have not yet attempted to analyse the interaction of the calling and receiving telephone: the only type of consistency they wish to check at present is that the subset of behaviours of a telephone that each ViewPoint represents are consistent. In particular the subsets of behaviour 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 of the 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, the "replace receiver" transition is shown between "connected" and "idle" in Anne's ViewPoint (because it is 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, and no transition is shown between them, then it is assumed that no such transition can occur. 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".*

Again 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 the consistency rules that may be invoked and applied by the ViewPoint owner. These rules are defined by the method designer. We have developed a notation for expressing the rules (presented in a simplified form below), which allows the method designer to express relationships between objects in the specifications of a source ViewPoint (from which the rule is invoked), VP_S, 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 above 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}

Hence, for rule R_1, the following entry has been defined:

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

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 $\text{missing}(\sigma, \psi.ps_D, R_i)$ 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 merely 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, which 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.

4.4. Resolution of inconsistencies

Anne and Bob consider the inconsistency resulting from the application of rule (i) 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. One possible resolution would be to distinguish the connected states 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 (ii). The most 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 5.

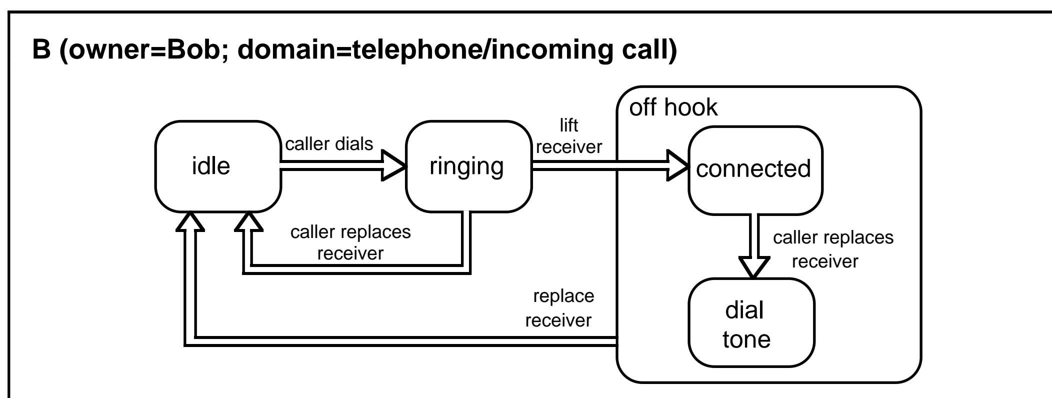

Figure 3: *Bob's ViewPoint specification after resolution.*

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.

Note that this resolution has accidentally resolved the inconsistency resulting from the application of rule (i) as well, in that the connected state in Bob's ViewPoint 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.

Anne now reapplies consistency rule (ii) to confirm that the problem is resolved. As she is group leader, she also re-applies all the other checks, and discovers, to her satisfaction, that the ViewPoints are completely consistent.

4.5. Support for resolution of inconsistencies

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

Support for resolution is in the form of a list of actions, defined as part of the method. Each consistency rule has a number of actions associated with it, 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 (i) above. Anne and Bob both have available the "missing" predicate described in section 4.3. They also have the list of suggested actions for tackling the resolution. In this case the available actions are:

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)

Note that some actions were derived directly from the rule that failed. These include removing items that make the rule hold, or adding items required by the rule. In particular, action (13) is offered as a likely suggestion, because it is normally assumed that under-specification is the cause of the problem, and it 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: we assume that methods evolve as lessons are learnt about their application.

One further source of suggested actions is from general heuristics defined as part of the method. For example, action (8) is derived from a heuristic for this type of state transition diagram, which suggests

that as an alternative to deleting a transition from a super-state, the transition could be devolved to the sub-states. In this case, action (8) does 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 [Easterbrook 1991]. 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 stored as such in the appropriate ViewPoint work record, so that the process may be continued at a later point.

Each ViewPoint always has available 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. All 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 off rule (ii) is resolved:

- The inconsistency is represented in Anne's ViewPoint 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 B.
- Anne selects this latter action, as a suggested resolution for Bob to carry out. Bob agrees and so adds the new state.
- An annotated entry is added to each ViewPoint's work record to record that this particular 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 any other inconsistencies. These are only performed locally; i.e., 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 noted 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 specifically concerned the inconsistency from rule R_2. Any effect on other inconsistencies can be dealt with when the ViewPoint owners specifically consider these.

4.6. Further elaboration

Anne and Bob now proceed to consider some additional features which will be made available on this phone system. 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:

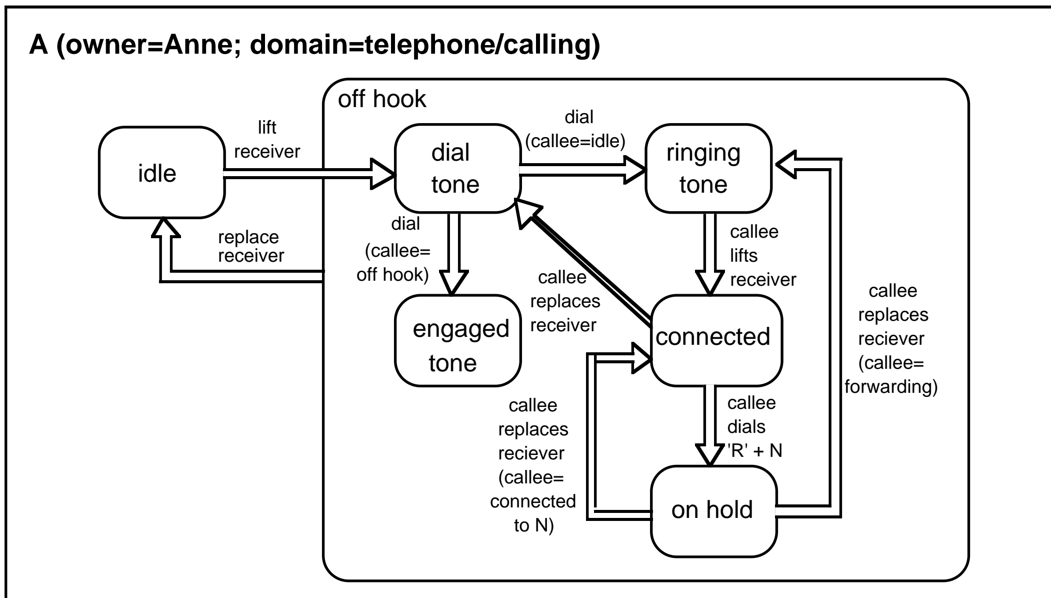

Figure 4: *Adding an "on hold" state to Anne's ViewPoint specification.*

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.

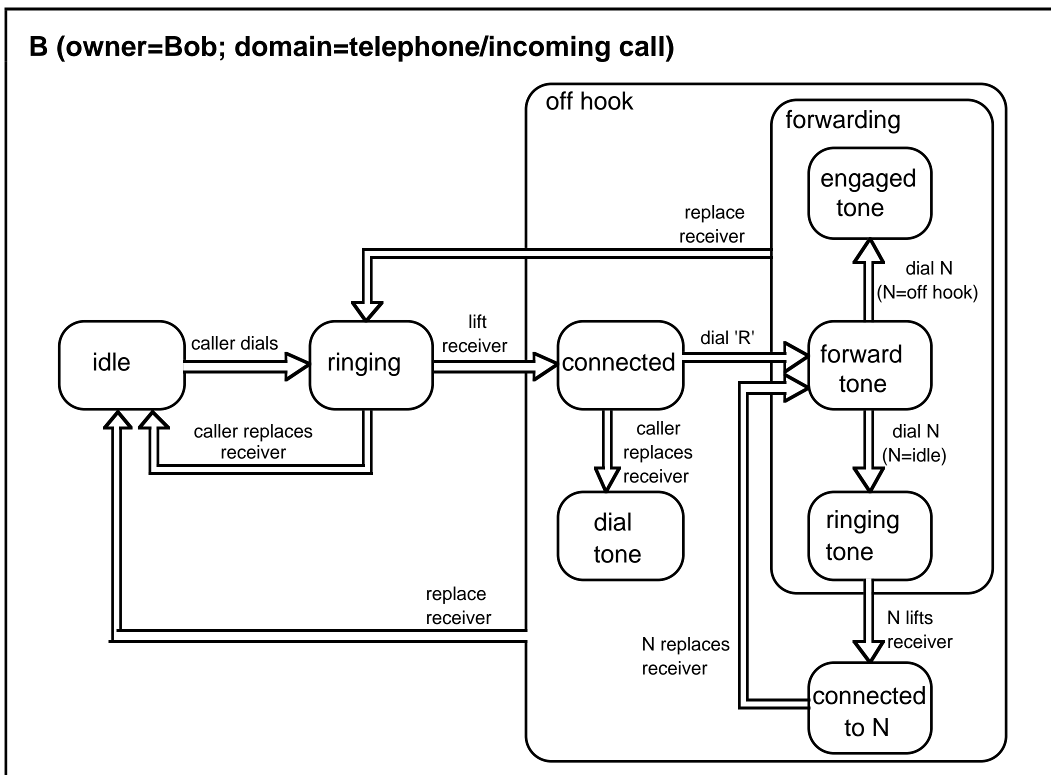

Figure 5: *Extending Bob's ViewPoint specification to handle call forwarding.*

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, it is an incoming call, and replacing the receiver does not disconnect. 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 as shown in figure 6.

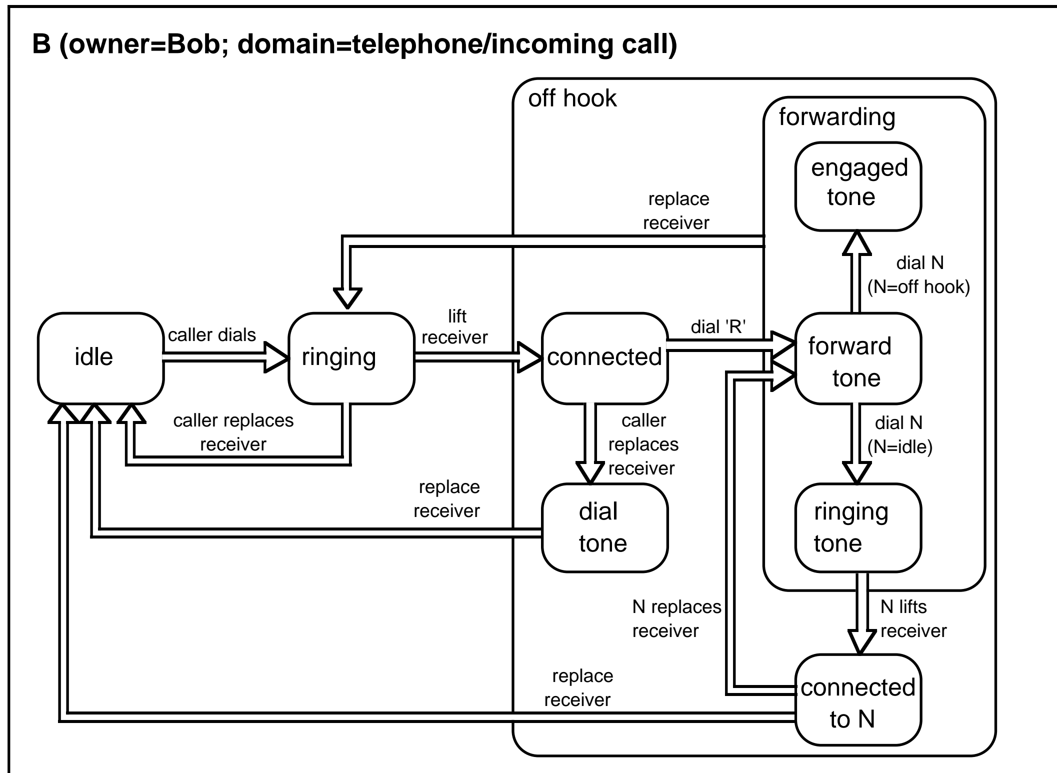

Figure 6: *Replacing the receiver only returns the phone to an “idle” state if there is a “dial tone” or “connected to N”.*

This now reintroduces an inconsistency according to rule (i), as Bob no longer has a transition from connected to idle. The fact that Bob has made a change that affects a previous resolution is noted, so that a suggestion to Bob that he re-checks rule (i) at some point can be made (this is particularly useful when constructing automated support for such a resolution process).

When Bob checks the 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.

4.7. Support for monitoring of inconsistencies

Throughout these further elaborations to ViewPoints, each action is checked for its effect on the known inconsistencies in each ViewPoint. In our scenario, only two kinds of inconsistency were detected, as we only applied two consistency rules. Both of these inconsistencies were resolved and annotated with the action that resolved them. The list of unresolved inconsistencies is empty, but this does not mean there are no other inconsistencies, as yet undetected. For example, if rule R_1 were applied after the elaboration above, a new inconsistency between the states labelled “ringing tone” in each ViewPoint would be detected - the transition “replace receiver” has a different destination in each case (and the same applies to “engaged tone”). This inconsistency will be detected the next time the rule 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 was the action that resolved the inconsistency resulting from the application of rule (i), its detection is likely to re-introduce the inconsistency. When the list of detected 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 R1 again. If he does this, there are again two possibilities:

- The inconsistency does not re-appear. In this case some unknown action since the inconsistency was first resolved has affected it. Thus, the inconsistency is annotated to indicated that some unknown action between the original resolution and the current action resolved it.
- The inconsistency re-appears, as is the case in our scenario. Here, the inconsistency is marked as unresolved, but still annotated with the sequences of actions that resolved, and then 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. User-defined relationships

There are still a number of conflicts between the two ViewPoints which have not been detected. For example, Anne still has a transition from her “connected” state, indicating that the callee can cause a disconnection by replacing the receiver. Bob’s ViewPoint assumes that the callee replacing the receiver has no effect on the connection, but according to his ViewPoint, the callee can be in the “connected as callee” state after replacing the receiver, even though “connected” is part of the “off hook” super-state!

A further set of consistency rules will detect these conflicts at the next stage of the method. This involves building relationships between the transitions of the caller and transitions of the callee, in order to model the dynamics of interaction between devices, in this case two telephones. The ViewPoints framework allows new relationships to be defined to represent such interactions.

For example, a connection between two phones must be synchronised such that if the connection terminates, both phones must move out of the “connected” state. This can be modelled by defining a relationship between “connected as caller” in Anne’s ViewPoint, and the state “connected as callee” in Bob’s. Similarly, Anne’s transition “callee replaces receiver” is the same stimulus as “replace receiver” in Bob’s ViewPoint.

As well as allowing ViewPoint owners to define such relationships between elements of their specifications, the method provides a further set of consistency rules. These check that the ViewPoints are consistent given the constraints imposed by any new relationships that are defined.

In the scenario, a number of further inconsistencies can be detected by applying these rules. For example, in Anne’s ViewPoint, the transition from “connected” to “dial tone”, labelled “callee replaces receiver”, is inconsistent with the agreed resolution of the connected state. However, this is not detected until the two “connected” states of the caller and callee are linked together as co-existent states. The most sensible resolution of this inconsistency is to delete the transition, not to transfer the corresponding transition into Bob’s ViewPoint, as might be expected in other circumstances. Figure 7 and 8 show the two ViewPoints after further inconsistencies have been resolved.

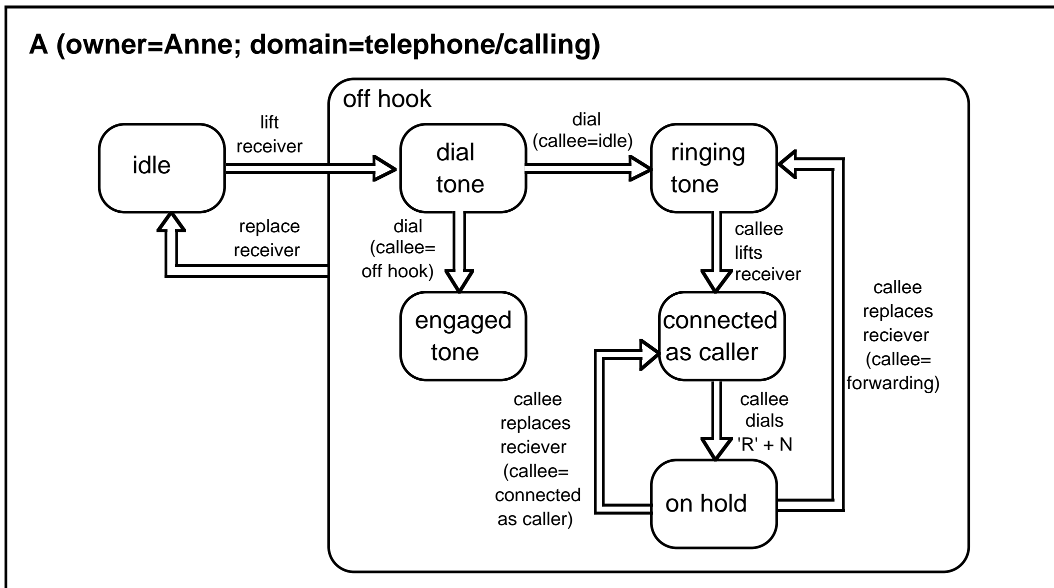

Figure 7: *Anne's ViewPoint specification after conflict resolution.*

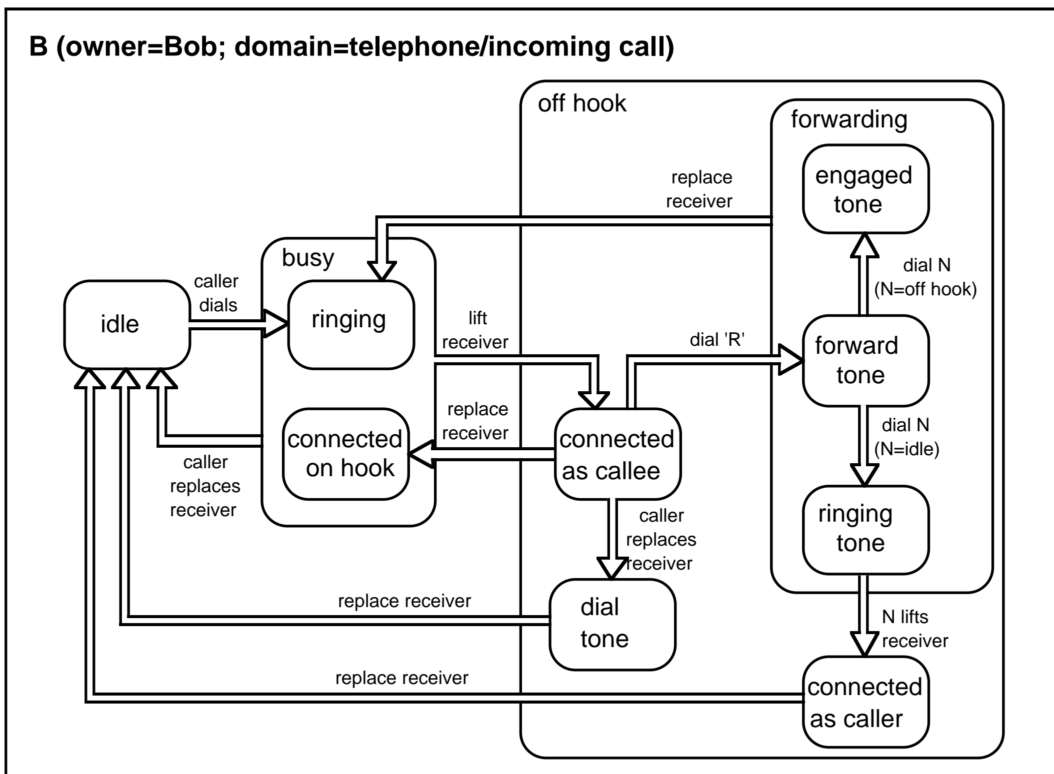

Figure 8: *Bob's ViewPoint specification after conflict resolution.*

4.9. Support for user-defined relationships

ViewPoint owners can define relationships between ViewPoints. These supplement the relationships defined by the set of rules defined by the method designer. Relationships can be defined by direct manipulation (i.e., the partial specifications are displayed graphically, and the user selects items within them to link together), or by entering the relationships in the rule notation outlined in section 4.3.

For example, Anne defines all transitions labelled "callee replaces receiver" in her ViewPoint as equivalent to all transitions in Bob's ViewPoint labelled "replace receiver". This is recorded as:

VP_S.transition(_, _).name."callee replaces receiver" ~ B.transition(_, _).name."replace receiver"

where the underscore is used to denote 'any' element of a specification. The relationship denoted by '~' is a method-specific relationship, used to link together transitions in two interacting devices, in order to analyse their interaction.

Having defined this relationship, Anne may then choose to export it to Bob's ViewPoint, in which case Bob will have to decide whether or not to accept the suggestion. Once they have defined a number of relationships like those above, Anne or Bob may choose to apply some of the consistency rules relating to device interaction. For example, if two states are linked together, then for any transition from one of them, there must be a corresponding transition from the other. If we define a corresponds predicate:

$$\text{corresponds}(X, Y) \equiv (X \sim Y) \vee (X.\text{name} = Y.\text{name})$$

so that transitions in two ViewPoints correspond if they have been linked together, or have the same name. The consistency rule can then be expressed as:

R_3: $\forall\ VP_D(\text{STD}, D_a)$

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

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.

4.10. Implications on requirements specification

Incremental exploration and resolution of the inconsistencies reveals an important mismatch between the conceptual models held by the two participants described in our scenario; namely about who a connection can be terminated by, 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.

5. Undetected conflicts

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, as well as mistakes. There remains the possibility that some conflicts and mistakes will not manifest themselves as inconsistencies.

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 defined by the method, rather than a problem with the approach. 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 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.

6. Implementation

A prototype computer-based environment and associated tools (*The Viewer*) have been constructed to support the framework [Nuseibeh & Finkelstein 1992]. *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 selected in- and inter-ViewPoint consistency rules, and records the results of all such consistency checks in the appropriate ViewPoint's work record for later analysis. A preliminary prototype of an 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.

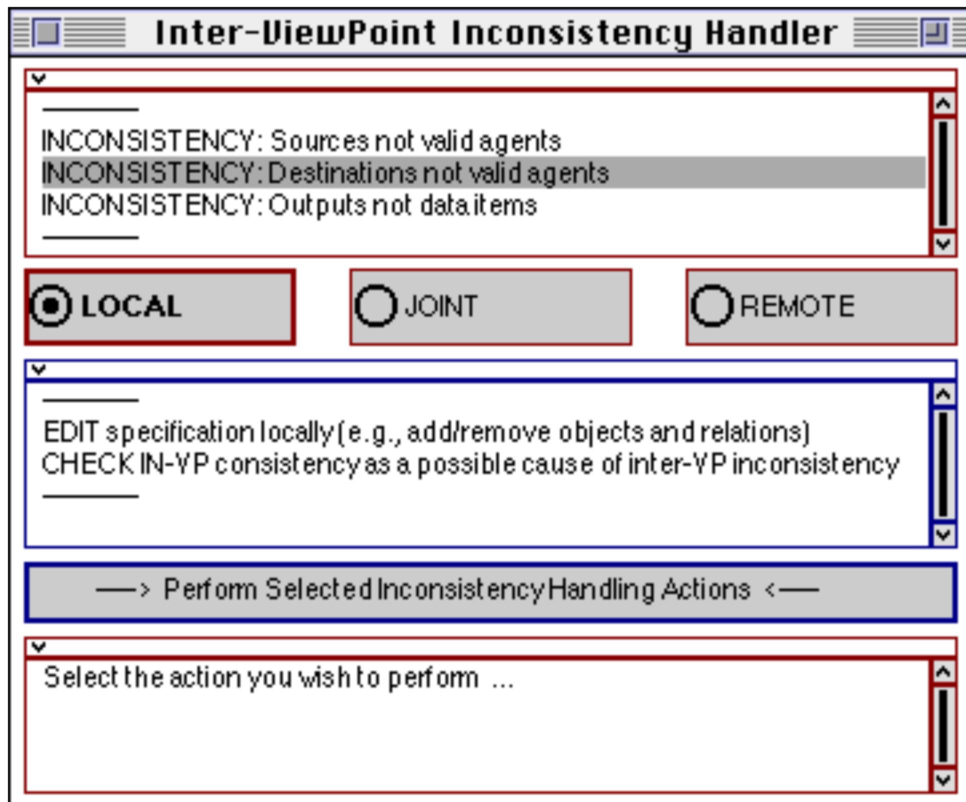

Figure 9: *The user-interface of an inter-ViewPoint Inconsistency Handler. A list of broken consistency rules is shown in the top pane, and a list of inconsistency handling actions for any selected rule is shown in the middle pane. These actions may be "local" to the source ViewPoint initiating the checks (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.*

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 in two ways: as part of a method, and as additional links identified by development participants during development. We have demonstrated how inconsistencies identified by checking these relationships may be resolved, and illustrated how subsequent evolution affects a resolution. Undone 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 have demonstrated that 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, development 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. We believe 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

The authors would like to acknowledge the contributions and feedback of Anthony Finkelstein and Jeff Kramer. This work was partly funded by the UK Department of Trade and Industry as part of the Eureka Software Factory (ESF) project, and the EPSRC as part of the VOILA project.

References

Ainsworth, M., A. H. Cruickshank, L. G. Groves and P. J. L. Wallis (1994); "Viewpoint Specification and Z"; *Information and Software Technology*, 36(1): February 1994; Butterworth-Heinemann.

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

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.

Easterbrook, S. (1993); "Domain Modelling with Hierarchies of Alternative Viewpoints"; *Proceedings of International Symposium on Requirements Engineering (RE '93)*, San Diego, CA, USA, 4-6th January 1993, 65-72; IEEE Computer Society Press.

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

Easterbrook, S., E. E. Beck, J. S. Goodlet, L. Plowman, M. Sharples and 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.

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

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

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.

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

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

Greenspan, S. and M. Feblowitz (1993); "Requirements Engineering Using the SOS Paradigm"; *Proceedings of International Symposium on Requirements Engineering (RE '93)*, San Diego, CA, USA, 4-6th January 1993, 260-263; IEEE Computer Society Press.

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

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

Kramer, J. (1991); "CASE Support for the Software Process: A Research Viewpoint"; *Proceedings of Third European Software Engineering Conference*, Milan, Italy, October 1991, 499-503; LNCS 550, Springer-Verlag.

Leite, J. C. S. P. and P. A. Freeman (1991); “Requirements Validation Through Viewpoint Resolution”; *Transactions on Software Engineering*, 12(12): 1253-1269, December 1991; IEEE Computer Society Press.

Mullery, G. (1979); “CORE - a method for controlled requirements expression”; *Proceedings of 4th International Conference on Software Engineering (ICSE-4)*, 126-135; IEEE Computer Society Press.

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.

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.

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.

Nuseibeh, B., J. Kramer and A. Finkelstein (1994); “A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification”; *(to appear in) Transactions of Software Engineering*, 20(10): October 1994; IEEE Computer Society Press (earlier version appeared in Proceedings of ICSE-15, May 1993, pp.187-200).

Ross, D. T. and K. E. Schoman (1977); “Structured Analysis for Requirements Definition”; *Transactions on Software Engineering*, 3(1): 6-15, January 1977; IEEE Computer Society Press.

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.

Zave, P. (1993); “Feature Interaction and Formal Specifications in Telecommunications”; *IEEE Computer*, 26(8): 20-30, August 1993; IEEE Computer Society Press.

Zave, P. and M. Jackson (1993); “Conjunction as Composition”; *Transactions on Software Engineering and Methodology*, 2(4): 379-411, October 1993; ACM Press.