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

Using ViewPoints for Inconsistency Management

STEVE EASTERBROOK†

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

BASHAR NUSEIBEH

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

0. Abstract

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

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

The paper demonstrates how inconsistency management is used as a tool for requirements elicitation, and how ViewPoints provide a vehicle for achieving this. Inconsistency is used as a stimulus for eliciting missing information and capturing user-defined relationships that arise between elements of an evolving specification.

1. Introduction

This paper is concerned with inconsistency management. Inconsistency is important in the software specification process, because on the one hand, inconsistent specifications cannot be satisfied, while on the other, it is hard to achieve consistency in a large specification written by a team. At first sight, frameworks which make use of viewpoints to represent alternative views of a specification seem to exacerbate the problem, in that they encourage the proliferation of inconsistent specification fragments. In this paper we will show that viewpoints can be used as the cornerstone of an effective inconsistency management strategy. We will argue that proper management of inconsistencies can lead to a more robust specification, because inconsistencies provide important clues about missing information.

---

† Currently a visiting research associate at the NASA/WVU Software IV&V Facility, in Fairmont, West Virginia, operated by West Virginia University, and supported under NASA cooperative agreement NCCW-0040 under the supervision of the NASA headquarters Office of Safety and Mission Assurance (Code Q).

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

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

2. Inconsistencies in an Evolving Specification

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

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

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

2.1. Tolerating inconsistency

A specification that contains an inconsistency is dangerous because it can be interpreted in more than one way. This leads to a serious problem in version control. In effect, an inconsistent specification can be regarded as two (or more) versions rolled into one. Anyone reading parts of

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

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

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

Inconsistencies can be tolerated during the development of a specification if we can overcome the versioning problem. For example, Balzer (1991) makes use of *pollution markers* to address this problem in a programming support environment. Pollution markers are used (a) to identify inconsistent data to code segments and to human agents, and (b) to screen inconsistent data from other segments that are sensitive to inconsistency.

Once inconsistencies are marked, their effects can be traced. Schwanke and Kaiser (1988) describe a configuration management tool, CONMAN, which performs this task. They identified six kinds of inconsistency pertaining to the relationships between code modules, versions, and data types. CONMAN assists in identifying and tracking these inconsistencies without requiring them to be removed. It also protects programmers from the problems of inconsistent code, and facilitates compilation in the presence of inconsistency.

An alternative approach is 'lazy consistency', proposed by Narayanaswamy and Goldman (1992) This approach favours software development architectures where impending or proposed changes - as well as changes that have already occurred - are "announced". This allows the consistency requirements of a system to be "lazily" maintained as it evolves. The approach is particularly effective in a distributed environment where conflicts or "collisions" of changes made by different developers may occur.

Finally, for formal specifications, Besnard and Hunter (1994) have explored the use of a paraconsistent logic to facilitate reasoning in the presence of inconsistency. In general, it is possible to partition an inconsistent specification so that each partition is internally consistent,

---

[1] Formally: $\{a, \neg a\}$ J b *because of the axioms:* $a \rightarrow (a \vee b)$ *and* $(\neg a \wedge (a \vee b)) \rightarrow b$

but that when the partitions are merged the inconsistency reappears. In this paper, we use viewpoints as the building blocks by which to achieve the partitioning.

The ViewPoints framework divides a specification into many overlapping partitions, and provides a distributed approach similar to 'lazy consistency', in which separate ViewPoints are responsible for checking their own consistency rules. In (Easterbrook, et al., 1994), we discuss how coordination between viewpoints can be supported without requiring consistency to be maintained. In this paper we describe how the framework provides support for the whole range of inconsistency management activities.

2.2. Inconsistency management

Having argued that tolerance of inconsistency is useful, we shift the emphasis from consistency maintenance to management of inconsistency. Management of inconsistency consists of a number of activities (Nuseibeh, 1994a):

Inconsistency *detection* focuses on identifying specification knowledge that breaks a consistency rule. For example, discovering (or deriving) a contradiction from a specification is a common way of detecting inconsistencies in logic-based specifications. A contradiction denotes any situation in which a fact, X, and its negation, ¬X, are found to hold simultaneously. More generally, an inconsistency is detected if any relationship, $\Re$, which should hold between elements of a specification, is found not to hold.

Inconsistency *classification* focuses on identifying the kind of inconsistency that has been detected in a specification. There are two facets of inconsistency that may be subject to classification. On the one hand, inconsistencies may be classified according to their cause. For example, some inconsistencies may be the result of inadvertent mistakes such as typographical errors, while others may be the result of deeper conflicts between development participants such as conceptual disagreements. On the other hand, inconsistencies may be classified into different pre-defined kinds prescribed by a method engineer or tool designer. For example, the CONMAN system (Schwanke & Kaiser, 1988) identifies six different kinds of inconsistency that may arise in programming, and uses this classification to react accordingly.

Inconsistency *handling* focuses on acting in the presence of inconsistencies (Finkelstein, et al., 1994a). For example, when an inconsistency is detected, the appropriate action may be one of:

- *Ignore* - this is appropriate if the inconsistency is relatively isolated, and its presence does not prevent further development, or where the level of risk does not justify the cost of fixing it. It will still be necessary to keep track of the inconsistency, in case its impact increases.
- *Delay* - this may be used when further information is required, but is not immediately available. Again the inconsistency needs to be monitored in case it interferes with other aspects of development, or in case it disappears as a result of other actions.

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

2.3. Inconsistency implies missing information

We believe that inconsistency management is an invaluable tool for knowledge elicitation. In software development, one often talks about under- and over-specification (and, in fact, inconsistencies are often the result of either of these two cases). A requirements specification that is under-specified usually means that too much design freedom has been left to the designer. Therefore, there is a need to elicit further requirements.

Over-specification is less obvious. Over-specification usually means that a specification is overly constrained, and that typically it cannot be implemented. The traditional approach to over-specification is to weaken one of the constraints, but this is not always the best course of action. Consider the following scenario.

> A requirement for an interactive operating system stated that all user commands must be completed in less than two seconds. One of the operations concerned involved extensive disk access, and the designers spent many days trying to optimise the operation so that they could meet this requirement, without any success. Eventually, they returned to the analyst who pointed out that all they needed to do was to display a message indicating how long the operation would actually take!

The problem above arose because the original requirements were not fully understood. The speed requirement was not that machine operations should be quick, but that the user should get rapid feedback that the machine was operating as requested. In other words, the problem was not that the specification was overly constraining, but that not enough was known about the original requirements.

To recap, inconsistencies in a specification usually indicate that further knowledge elicitation is needed. Managing inconsistency as outlined in section 2.2, and illustrated in this paper, provides the opportunity for more focused elicitation.

3. Inconsistency and ViewPoints

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

3.1. Key problems of inconsistency management

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

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

3.2. ViewPoints

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

A ViewPoint combines the notions of 'actor' or 'knowledge source' in the development process, with 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 (Finkelstein, et al., 1994b) 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 (Ross & Schoman, 1977) and CORE (Mullery, 1979), and more recently deployed for representing partial specifications (Ainsworth, et al., 1994; Niskier, 1989), validating requirements (Leite & Freeman, 1991), domain modelling (Easterbrook, 1993) and service-oriented specification (Greenspan & Feblowitz, 1993; Kotonya & Sommerville, 1992). We use ViewPoints to organise multi-perspective software development (knowledge) and to manage inconsistency.

3.3. Inconsistency management using ViewPoints

In our framework, there is no requirement for changes to one ViewPoint to be consistent with other ViewPoints (Finkelstein, et al., 1994a). 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 focus on the management of inconsistencies, so that the specification process remains a co-ordinated 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 that provides several notations will also specify how those notations should be used in combination,

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

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

4. Scenario

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

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

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 (Ghezzi, et al., 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).

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 (Zave, 1993).

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

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

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

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

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

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

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 (that is, a telephone handset) [as permitted by step (b) of the method], a number of consistency relationships must hold between their ViewPoints [defined in step (c) of the method].

2 We use the extensions to state transition diagrams proposed by Harel (Harel,). 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.

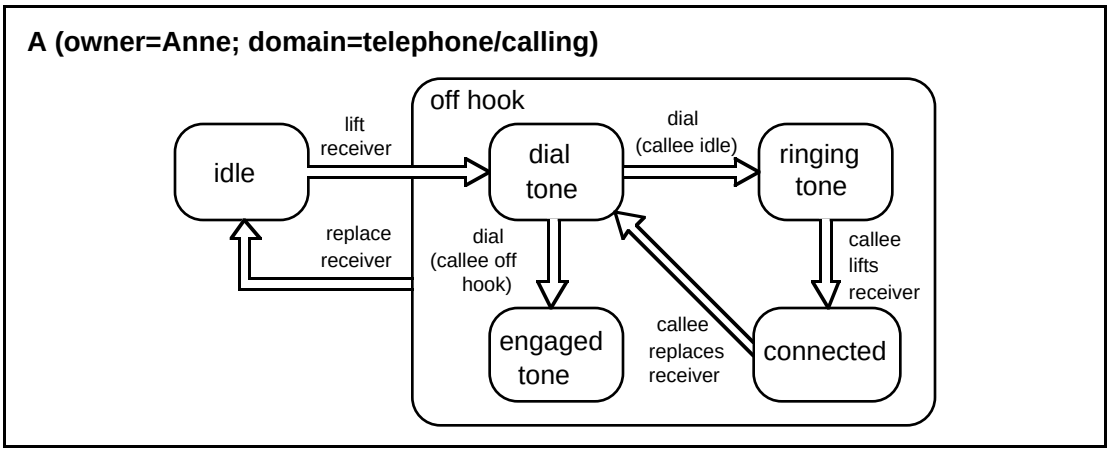

Figure 1: *Anne's initial ViewPoint specification for making a call.*

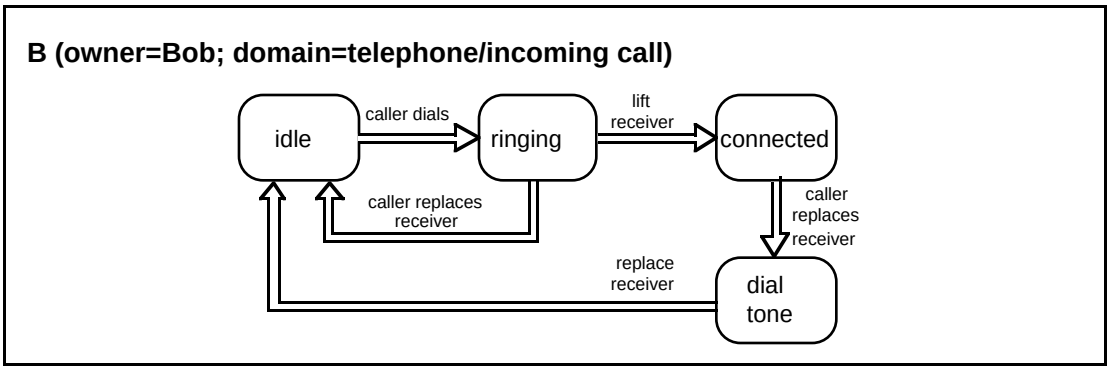

Figure 2: *Bob's initial ViewPoint specification for receiving a call.*

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 of the consistency rules in more detail. Both rules arise directly from the method, and act as constraints on the way in which viewpoints can be partitioned, in order to remove any ambiguity:

(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 whether 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 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, VP_S, (from which the rule is invoked) and a destination ViewPoint, VP_D (Easterbrook, et al., 1994; Nuseibeh, et al., 1994). 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:

$\{preconditions\} \Rightarrow [agent, action]\ \{postconditions\}$

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

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

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), R1)

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

4.4. Resolution of inconsistencies

Anne and Bob now 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. 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 3.

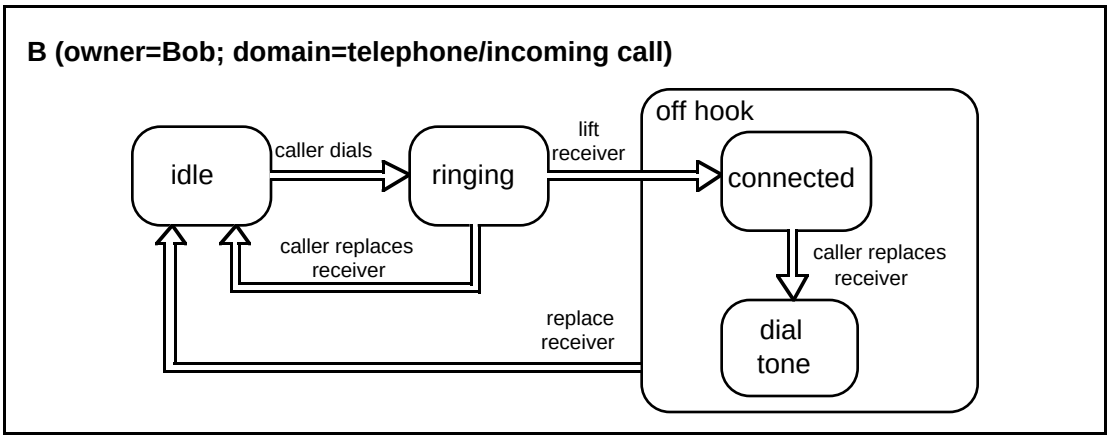

Figure 3: *Bob's ViewPoint specification after resolution.*

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.

[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.

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

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 a resolution process, and is available whenever ViewPoint owners wish to handle 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. They also have the list of suggested actions for tackling the resolution. The available actions are given in table 1.

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. For example, action (13) is suggested because under-specification may be the cause of the problem, and could 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 the application of a rule. They may also have resulted from the experience of method users in the past: we assume that methods evolve as lessons are learnt about their use.

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

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

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

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.

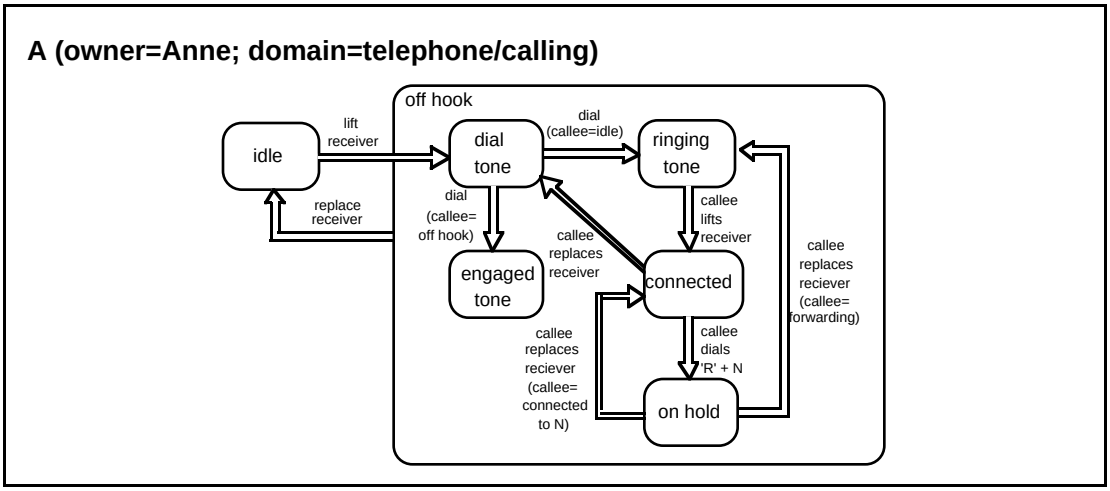

Figure 4: *Adding an 'on hold' state to Anne's ViewPoint specification.*

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.

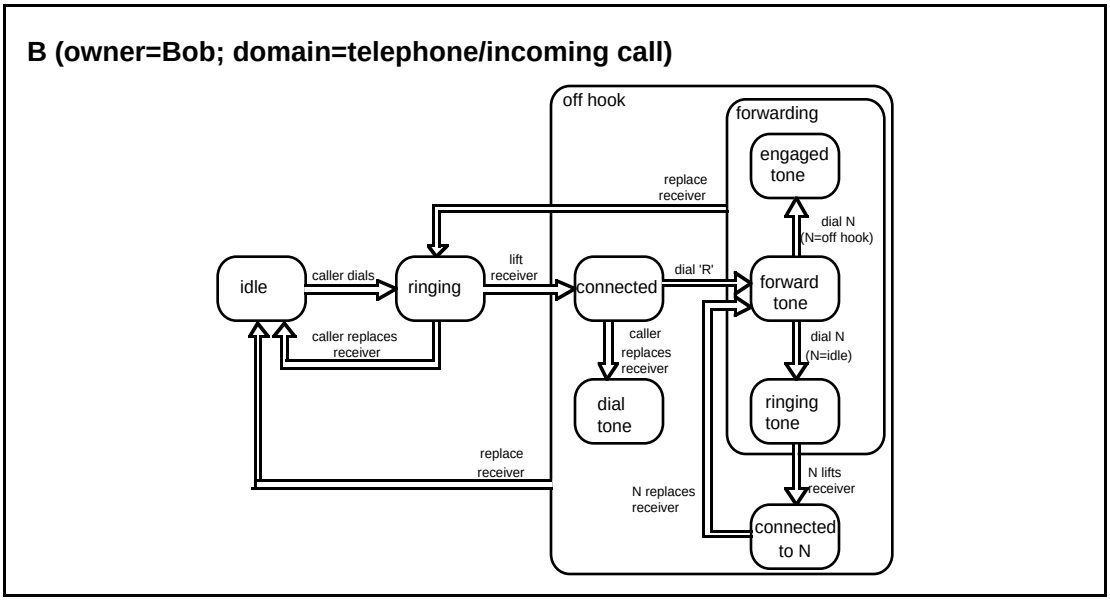

Figure 5: *Extending Bob's ViewPoint specification to handle call forwarding.*

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, 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 as shown in figure 6.

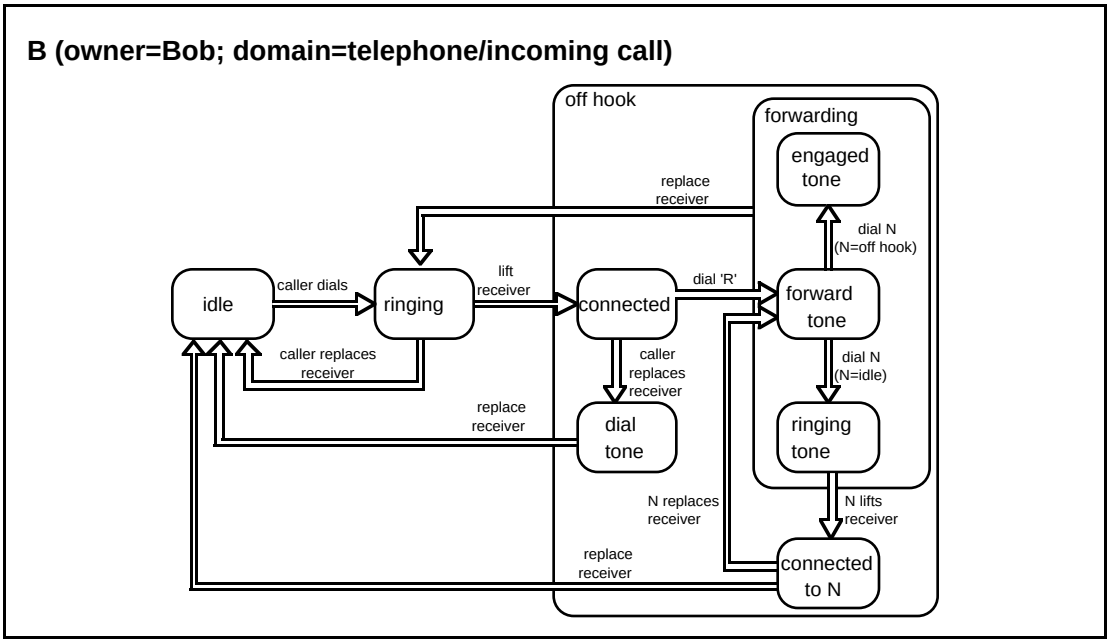

Figure 6: *Replacing the receiver only returns the phone to an 'idle' state if there is a 'dial tone' or 'connected to N'.*

This now re-introduces 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 of 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: 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 effect. The inconsistency is annotated to indicated that it was resolved by some 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. User-defined relationships

There are still a number of inconsistencies between the two ViewPoints which have not yet 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 [step (d)] 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 different instances of the device, 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 [step (e)]. 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’ (See figure 4), 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 as co-existent states. The most sensible resolution of this inconsistency is to delete the transition in Anne’s ViewPoint, rather than 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.

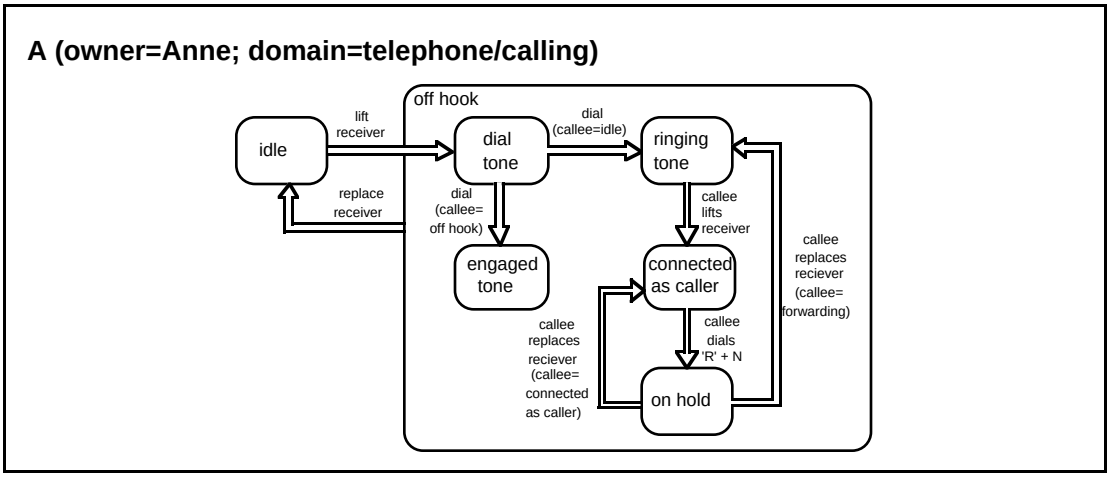

Figure 7: *Anne’s ViewPoint specification after conflict resolution.*

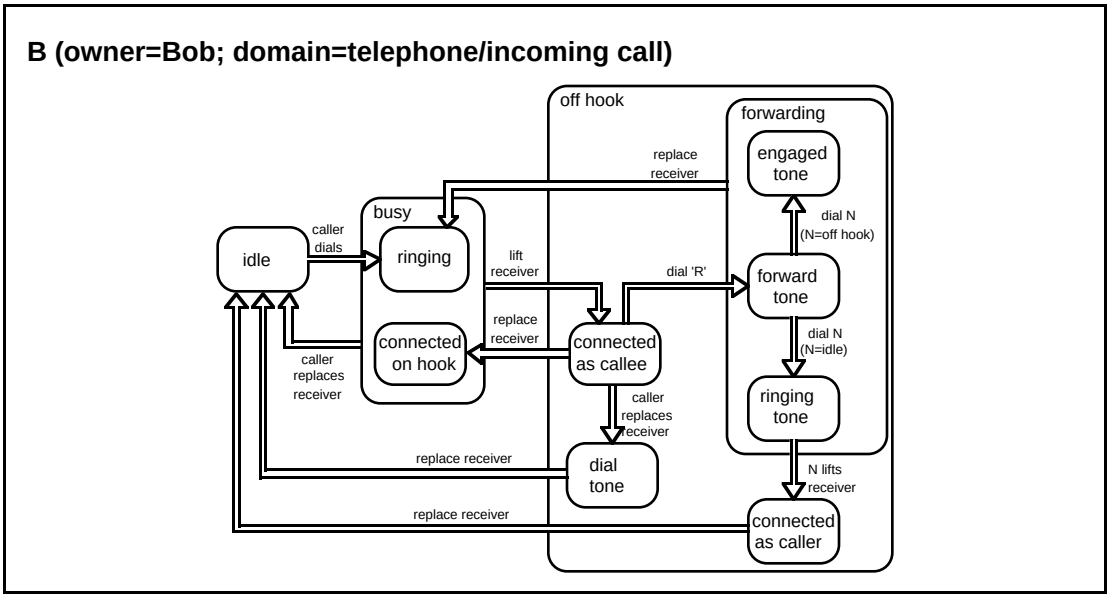

Figure 8: *Bob's ViewPoint specification after conflict resolution.*

4.9. Support for user-defined relationships

ViewPoint owners can define relationships between ViewPoints. These supplement the relationships defined by the 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 as representing the same action.

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 similar relationships, 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:

corresponds(X, Y) ≡ (X ~ Y) ∨ (X.name = Y.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(STD, D_a)

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

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

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

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

transition(connected, idle).name.'replace receiver'

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

5. Discussion

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

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

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

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

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

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

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

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

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. During 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 record the results of all such consistency checks in the appropriate ViewPoint's work record. A prototype Inconsistency Handler has also been implemented, to illustrate the kind and scope of inconsistency management we expect tool support to provide (figure 9).

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

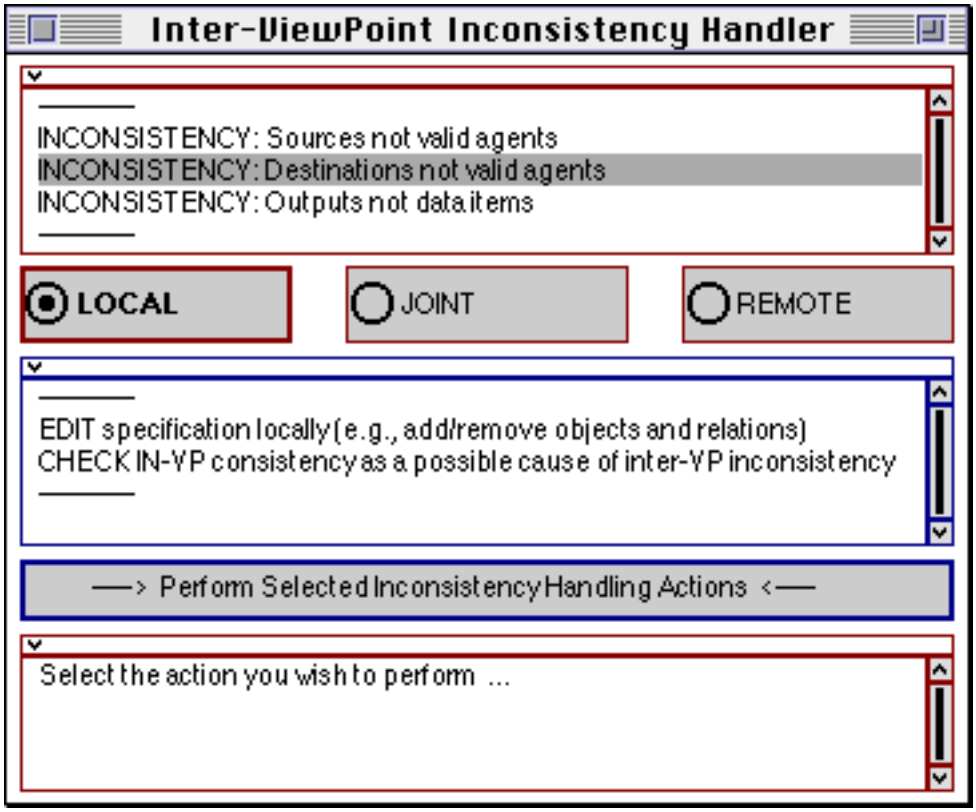

Figure 9: *The user-interface of an inter-ViewPoint Inconsistency Handler provided by The Viewer. A list of broken consistency rules is shown in the top pane, and a list of inconsistency handling actions for any selected rule is shown in the middle pane. These actions may be "local" to the source ViewPoint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

7. Conclusions and Future Work

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

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

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

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

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

8. Acknowledgements

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

9. References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Narayanaswamy, K., & Goldman, N. (1992). "Lazy" Consistency: A Basis for Cooperative Software Development. In *Proceedings of International Conference on Computer-Supported Cooperative Work (CSCW '92)*, (pp. 257-264). Toronto, Ontario, Canada: ACM SIGCHI & SIGOIS.

Niskier, C., Maibaum, T. S. E., and Schwabe, D. (1989). A look Through Prisma: Towards Pluralistic Knowledge-Based Environments for Software Specification Acquisition. In *Proceedings, Fifth IEEE International Workshop on Software Specification and Design, Pittsburg, Penn*

Nuseibeh, B. (1994a). *Computer-Aided Inconsistency Management in Software Development* (Technical Report No. Department of Computing, Imperial College, London, UK.

Nuseibeh, B. (1994b) *A Multiple-Perspective Framework for Method Integration*. PhD Thesis, Department of Computing, Imperial College, London, UK, October 1994.

Nuseibeh, B., & Finkelstein, A. (1992). ViewPoints: A Vehicle for Method and Tool Integration. In G. Forte, N. H. Madhavji, & H. A. Muller (Ed.), *Proceedings of 5th International Workshop on Computer-Aided Software Engineering (CASE '92)*, (pp. 50-60). Montreal, Canada: IEEE Computer Society Press.

Nuseibeh, B., Finkelstein, A., & Kramer, J. (1993). Fine-Grain Process Modelling. In *Proceedings of 7th International Workshop on Software Specification and Design (IWSSD-7)*, (pp. 42-46). Redondo Beach, California, USA: IEEE Computer Society Press.

Nuseibeh, B., Kramer, J., & Finkelstein, A. (1994). A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *Transactions of Software Engineering*, *20*(10), 760-773.

Pohl, K. (1994) *A Process Centred Requirements Engineering Environment*. PhD Thesis, University of Aachen, Aachen, Germany, October 1994.

Ross, D. T., & Schoman, K. E. (1977). Structured Analysis for Requirements Definition. *Transactions on Software Engineering*, *3*(1), 6-15.

Schwanke, R. W., & Kaiser, G. E. (1988). Living With Inconsistency in Large Systems. In J. F. H. Winkler (Ed.), *Proceedings of the International Workshop on Software Version and Configuration Control*, (pp. 98-118). Grassau, Germany: B. G. Teubner, Stuttgart.

Zave, P. (1993). Feature Interaction and Formal Specifications in Telecommunications. *IEEE Computer*, *26*(8), 20-30.

Zave, P., & Jackson, M. (1993). Conjunction as Composition. *Transactions on Software Engineering and Methodology*, *2*(4), 379-411.