To appear in Concurrent Engineering: Research and Applications (Special Issue on Conflict Management), 1994.

Co-ordinating Distributed ViewPoints: the anatomy of a consistency check

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

ANTHONY FINKELSTEIN, JEFF KRAMER & BASHAR NUSEIBEH
Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2BZ
{acwf, jk, ban}@doc.ic.ac.uk

Support for Concurrent Engineering must address the "multiple perspectives problem" - many actors, many representation schemes, diverse domain knowledge and differing development strategies, all in the context of distributed asynchronous development. Central to this problem is the issue of managing consistency between the various elements of an emerging design. In this paper, we argue that striving to maintain complete consistency at all points in the development process is unnecessary, and an approach based on tolerance and management of inconsistency can be adopted instead. We present a scenario which highlights a number of important issues raised by this approach, and we describe how these issues are addressed in our framework of distributed ViewPoints. The approach allows an engineering team to develop independent ViewPoints, and to establish relationships between them incrementally. The framework provides mechanisms for expressing consistency relationships, checking that individual relationships hold, and resolving inconsistencies if necessary.

1. Introduction

Concurrent engineering involves the collaboration and co-ordination of a physically distributed team with variable opportunities for communication with one another. Traditional approaches to the problems of distributed working use a central database, or repository, to which all team members have communication access. Consistency is managed in this database through strict access control and version management, along with a common data model or schema. Such centralised approaches do not adequately support the reality of distributed engineering, where communication with a central database cannot always be guaranteed, and access control rapidly becomes a bottleneck (Cutkosky, *et al.*, 1993).

The alternative, a fully decentralised environment, is seen to be problematic because of the difficulties of maintaining consistency between a large collection of agents. However, these problems can be overcome by recognising that maintaining global consistency at all times is an unnecessary burden. Indeed, it is often desirable to tolerate and even encourage inconsistency, to maximise design freedom, and to prevent premature commitment to design decisions. The focus therefore shifts from maintaining consistency to the management of inconsistencies.

Our interests centre on the problems of requirements definition for large and complex systems. Although we concentrate especially on specification of software, we are generally concerned with composite systems, where the software components interact with a variety of different technologies. The development of such systems necessarily involves many people - each with their own perspective on the system defined by their skills, responsibilities, knowledge and expertise. The intersections between these perspectives are far from obvious because the knowledge within each perspective is represented in different ways.

With concurrent development, different perspectives may be at different stages of elaboration and may each be subject to different development strategies. The problem of how to guide and organise development in this setting - many actors, sundry representation schemes, diverse domain knowledge, differing development strategies - we term "the multiple perspective problem".

In this paper, we describe our approach to the multiple perspectives problem. We use a distributed environment to support the development and comparison of different perspectives. Our philosophy is to avoid the need for any central database, and to tolerate inconsistency. In particular we focus on the problems of how co-ordination can be achieved without restricting the freedom to develop

multiple perspectives. The paper presents a scenario to illustrate some of the issues raised by this approach. We then consider each issue in turn and describe how our approach addresses it.

2. ViewPoints

The framework upon which we base this work supports distributed software engineering in which multiple perspectives are maintained separately as distributable objects, called ViewPoints (Finkelstein, *et al.*, 1992). A ViewPoint can be thought of as a combination of the idea of an 'actor', 'knowledge source', 'role' or 'agent' in the development process, and the idea of a 'view' or 'perspective' which an actor maintains. In software terms, ViewPoints are loosely coupled, locally managed, coarse-grained objects which encapsulate partial knowledge about the system and 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.

Most importantly, the framework tolerates inconsistency, with no requirement for changes to one ViewPoint to be consistent with other ViewPoints (Finkelstein, *et al.*, 1994). Consistency checking is performed through a set of inter-ViewPoint rules, defined by the method, which express the relationships that should hold between particular ViewPoints. These rules define partial consistency relations 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 protocol is provided for applying consistency checks between ViewPoints, with the checking process being initiated by either ViewPoints' owner. A fine-grained process model in each ViewPoint provides guidance for the resolution of inconsistencies (Nuseibeh, Finkelstein, & Kramer, 1993).

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.

The framework we have described offers a coherent approach to the management of multiple perspectives. The approach supports multi-language specification, without the requirement for a common data model or language. It therefore facilitates method integration, as well as the obvious benefits of distributed working. We have demonstrated elsewhere the use of the framework to

implement software engineering methods such as CORE (Nuseibeh, Kramer, & Finkelstein, 1993) and the CDA (Kramer & Finkelstein, 1991), and have demonstrated how relationships between different representation schemes may be expressed.

In this paper, we concentrate on how co-ordination between ViewPoints can be supported, without requiring consistency to be maintained. There are a number of technical problems introduced by this approach. For example, how can we test an individual consistency relationship between two ViewPoints which have an arbitrary number of inconsistencies; and how can we support the resolution of inconsistencies in an incremental fashion, so that resolutions are not lost when the ViewPoints continue to evolve? We present a scenario to illustrate the problems, and then consider the issues raised in more detail.

3. Scenario

To illustrate the problems of co-ordinating ViewPoints, we will present a scenario in which hierarchically related ViewPoints are delegated to separate members of a team to develop. We use dataflow diagrams (DFDs) as an example notation, although the scenario could equally well apply to any of a large range of software engineering notations. In a dataflow diagram, a node in a graph, representing a process, may be decomposed in a separate diagram. When this happens, the set of inputs and outputs to that process should be shown on the decomposition. However, the owners of each diagram may wish to modify them independently.

Consider the following. Anne has created a ViewPoint to contain a top level dataflow diagram of the system. Each process in the diagram is, by default, labelled as primitive, in that it is not decomposed further. She then selects process Y and changes its state to non-primitive, to indicate that it is decomposed. She then delegates the job of decomposing it to Bob.

At this point, there is an inconsistency:

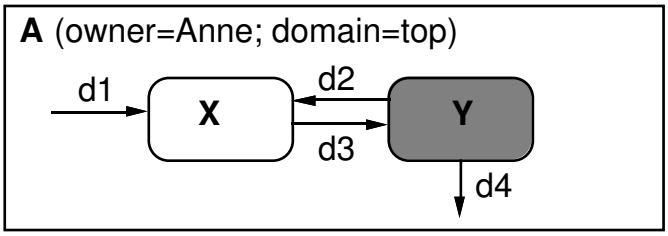

Figure 1. ViewPoint A contains a non-primitive process (process Y, shaded) for which no corresponding decomposition exists.

Bob creates a new ViewPoint to represent the decomposition. He gives the new ViewPoint a suitable label to indicate it is a decomposition of process Y.

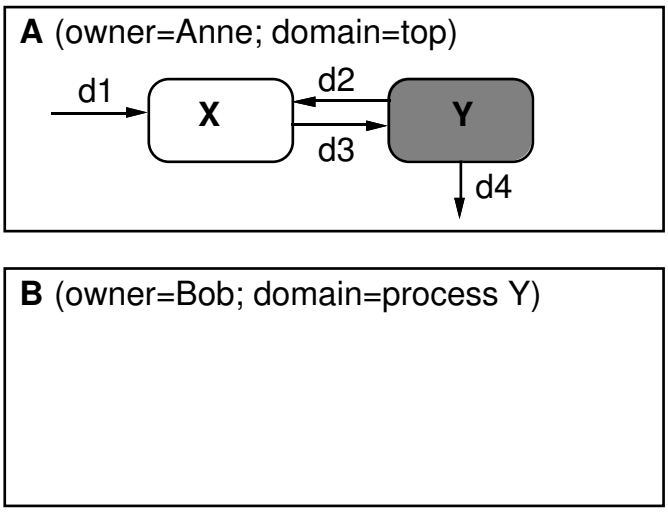

Figure 2. ViewPoint A contains a non-primitive process for which the decomposition does not have the same inputs and outputs (because the decomposition is empty).

Bob takes the input and output flows connected to process Y in the parent, and adds them to the

decomposition as its context. He then begins to define the processes that comprise the decomposition.

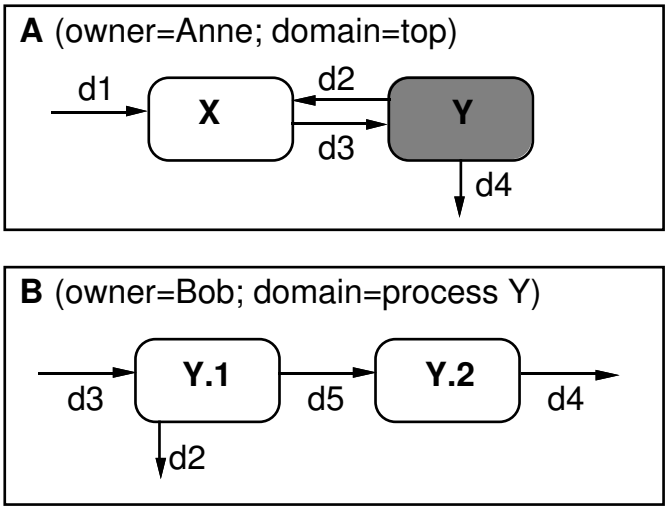

Figure 3. ViewPoint A is consistent with the decomposition.

Anne now does some more work on the parent. In the process of delegating decomposition of other processes, she realises one of the outputs of process Y is missing, and so she adds it (d7). Meanwhile, Bob has also noticed the omission, and adds it, using a different label (d6). He also adds another missing output (d9), and renames a third (d4 becomes d8).

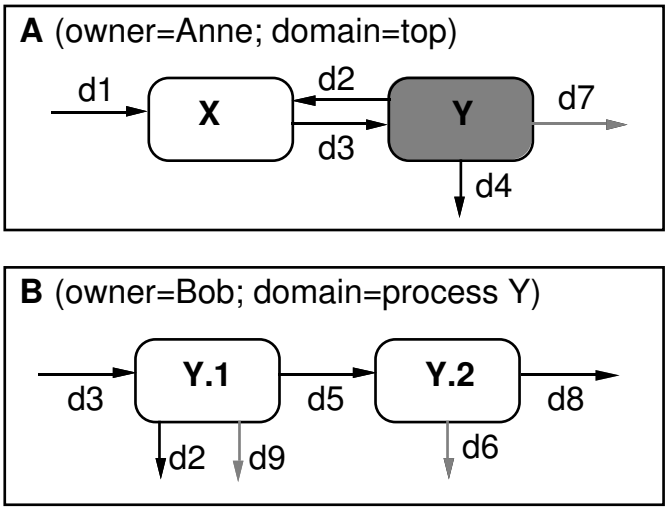

Figure 4. ViewPoint A is inconsistent with the decomposition, as the outputs to process Y in the parent do not match the contextual outputs in the decomposition ViewPoint.

Eventually, Bob discusses process Y with Chris, who is working on a decomposition for process X. They discover that the interaction between their processes is a lot more complex than Anne realised, and needs to be shown by merging their ViewPoints and then decomposing further levels. They discuss this with Anne, who agrees. They transfer elements of Bob's ViewPoint to Chris's and delete the ViewPoint for process Y.

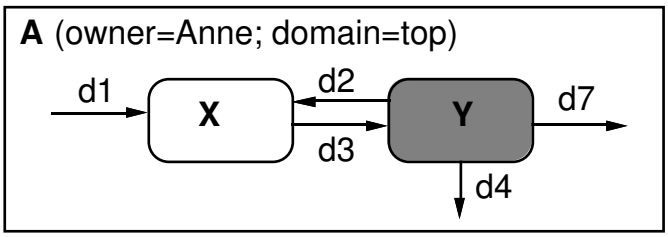

Figure 5. ViewPoint A contains a non-primitive process (process Y, shaded) for which no corresponding decomposition ViewPoint exists.

And so on. Note that the inconsistencies in figures 1 and 5 involve the same consistency relationship, as do those in 2 and 4. However, in each case the appropriate repair action is different. Note also that the inconsistency in figure 4 is vastly complicated by the fact that several changes have been saved up. In particular, it may not be possible to tell whether the last change made by Bob (i.e. renaming a flow) should merely be propagated to Anne's ViewPoint.

3.1. Issues

The scenario raises a number of questions about the provision of support for concurrent engineering activities:

- Where does the responsibility for creating a ViewPoint lie?
- How are relationships between ViewPoints expressed?
- When should relationships between ViewPoints be checked?
- How are relationships between ViewPoints checked?
- How are inconsistencies resolved?
- What happens if inconsistencies are not resolved?

We will now discuss each of these issues in turn.

4. Where does the responsibility for creating a ViewPoint lie?

The first step in the scenario was a decision to decompose a process in another ViewPoint. This decision implies that such a ViewPoint should be created, but it does not imply it should be created immediately. Nor is it clear where the responsibility for creating it lies. In the scenario, it is Anne's action which necessitates the creation of a new ViewPoint, but the development of the new ViewPoint is to be Bob's responsibility.

In an environment with strong consistency enforcement, the decomposition process described in the scenario might be a single action, which creates the new ViewPoint, flags the decomposition in the parent, and copies relevant information to the new ViewPoint. This would prevent the inconsistencies in figures 1 and 2 arising. However, it also confuses the division of responsibility for the two ViewPoints: the action of noting that a process should be decomposed only affects Anne's ViewPoint; the action of transferring material only affects Bob's. Changes that are local to a ViewPoint would normally be the responsibility of that ViewPoint's owner.

De-coupling these actions, as we have done in the scenario, permits more flexibility with development strategies. For example, Bob may have a pre-existing ViewPoint which he wishes to use as the decomposition. Alternatively, Bob may wish to begin development of the decomposition without copying Anne's contextual flows, perhaps because he has different ideas about what the contextual flows should be. In the scenario, Anne and Bob appear to be using a top down strategy, in which the system is described at the highest levels first. Later in the scenario they modify their descriptions in a bottom up manner. Clearly, there are a number of possible strategies they could adopt, and they may not wish to use the same strategy, nor stick to a particular strategy.

We have also de-coupled the actions of changing a label and propagating that change to other affected ViewPoints. There are a number of reasons for this: communication between ViewPoints may not always be possible; Anne may only be experimenting; Bob might not wish to accept the change yet (if at all); Bob may have simultaneously changed (or deleted) the same label.

De-coupling actions affecting different ViewPoints allows the actions to be combined in different ways depending on the development strategy (or strategies) chosen. Hence we do not attempt to define where the responsibilities for various actions lie, but instead we allow the maximum flexibility for allocating responsibilities. The method designer defines the set of possible development strategies for each type of ViewPoint, as part of the process model for each ViewPoint template. The process models will also define the mechanisms for dealing with potential inconsistencies that may arise as a result of following a particular strategy.

5. How are relationships between ViewPoints expressed?

In the scenario, the two ViewPoints have a relationship between them that needs to be clearly defined. This particular relationship arises from applying the software development method: the method provides dataflow diagrams as a notation, and decomposition of processes within a dataflow diagram as a development step. Similarly, a method which provides several notations will also specify how those notations should be used in combination, and how they inter-relate. Hence the possible relationships between ViewPoints are determined by the method.

The method designer defines the relationships that should hold between pairs of ViewPoints. Because inconsistency between ViewPoints is tolerated, the relationships are those that should hold, rather than those that actually do. Each relationship is expressed as a rule for determining whether that relationship holds. The rules can be applied as consistency checks when necessary.

Development of an individual ViewPoint may proceed unrestrained by relationships with other ViewPoints. When the relationships become important, the consistency rules provide the means for checking whether the relationships hold. The consistency checks are part of the ViewPoint, and hence are invoked by that ViewPoint. Just as there is no central database, there is no third party to check consistency between ViewPoints.

5.1. Types of Consistency Rule

Conceptually, there are three levels of consistency which might need to be checked: local to a ViewPoint, between two ViewPoints, and global. The ViewPoints framework supports the first two, as in-ViewPoint and inter-ViewPoint checks respectively. At the in-ViewPoint level, each rule defines a property that should hold of a specified ViewPoint. At the inter-ViewPoint level, each rule defines a relationship that should hold between two specified ViewPoints.

Handling global consistency is problematic in a fully distributed environment, in which there is no central database. In the ViewPoints framework, global consistency checking is eliminated by transforming global checks into in- and inter- ViewPoint checks. For example, if a particular method requires that some consistency condition holds for all ViewPoints, the method designer might define a ViewPoint to contain a representation of all the other ViewPoints. The global check then becomes an in-ViewPoint check for this new ViewPoint. However, such ViewPoints are not a privileged part of the framework, merely another type of ViewPoint that a method designer might choose to define.

It is also useful to distinguish between rules that check for existence (or absence) of information, and those that check for agreement of information. Existence and agreement rules are expressed slightly differently.

	Existence	**Agreement**
Level 1: In-ViewPoint Rules	*E.g. check for unconnected items, etc.*	*E.g. check for name clashes, etc.*
Level 2: Inter-ViewPoint Rules	*E.g. check for existence of a related ViewPoint.*	*E.g. check for consistency with information in a related ViewPoint.*

It is not necessary to assume at the inter-ViewPoint level that all the rules at the in-ViewPoint level hold. There may be circumstances under which a user may wish to perform an inter-ViewPoint check, without resolving local inconsistencies. An example is the consistency relationship between a parent and a child (decomposition) ViewPoint: we may wish to check and resolve the relationship with the parent as soon as the child is created, in order to transfer contextual information. The in-ViewPoint rules for the child (and possibly the parent) have not been applied, but the inter-ViewPoint check is still sensible.

5.2. Notation for Expressing Consistency Rules

Nuseibeh *et. al.* (1993; 1994) introduce a notation for inter-ViewPoint consistency rules based on

the expression of a relationship between a *source* ViewPoint (VP_S) and a *destination* ViewPoint (VP_D). The source ViewPoint is the one that invokes the rule. Relationships take the form:

$$\forall\ VP_S, \exists\ VP_D \text{ such that } \{ps_1\ \Re\ VP(t, d): ps_2\}$$

where VP(t, d) specifies the destination ViewPoint, with template t and domain d, and where ps_1 and ps_2 are partial specifications. The rule then states that ps_1 in the source ViewPoint is related to ps_2 in the destination ViewPoint by the relationship $\Re$. Example relationships are equality (=) and entailment ($\rightarrow$). The partial specifications will refer to relations, objects, typed attributes and values within the relevant notation. A 'dot' notation is used to refer to attributes of objects, so that for instance 'Arrow.Label.fred' refers to a value 'fred' of the attribute 'Label' of an object 'Arrow'. The partial specifications given in these rules may be arbitrarily complex, involving various logical connectives.

The quantifiers apply a relationship over the set of ViewPoints, to give a consistency rule. Given that the rules are defined as part of the ViewPoint template, they express relationships between ViewPoints that have not yet been created. Hence, the source ViewPoint is universally quantified, to indicate that the rule applies for every ViewPoint derived from that template. Once particular ViewPoints are instantiated from the template, this first quantifier can be dropped – the source ViewPoint is always the ViewPoint that contains the rule. For the remainder of this paper, we will present the rules as they would appear in an instantiated ViewPoint, i.e. without the first quantifier.

Here we introduce a few extensions to the notation. Firstly, the indication of the type of destination ViewPoint for which the relationship should hold is part of the quantification. Operationally, application of the rule involves determining to which destination ViewPoint the rule applies, determining whether it exists (invocation), and then checking the relationship holds (application). The quantifiers and the type definition for VP_D are needed for invocation, while the partial specifications and the relationship $\Re$ are needed for application. We will re-arrange the rules to reflect this:

$$\exists\ VP_D(t, d) \text{ such that } \{ps_S\ \Re\ VP_D: ps_D\}$$

where ps_S and ps_D are partial specifications from the source and destination ViewPoints respectively. Note that we use the prefix 'VP_D' on the right hand side of the relationship, to remind the reader that the partial specification on the right applies to the destination ViewPoint.

Secondly, note that this quantification insists that the destination ViewPoint must exist, implying that if it does not, it ought to be created. This form of rule therefore combines an *existence* relationship ("another ViewPoint exists...") and an *agreement* relationship ("...which is related in this way"). In some circumstances it may be necessary to express only one or other of these relationships, in which case we can decompose the rule:

$\{ps_S\} \rightarrow \exists\ VP_D(t, d)\ \{ps_D\}$ (an existence relationship)

$\forall\ VP_D(t, d)\ \{ps_S\ \Re\ VP_D: ps_D\}$ (an agreement relationship)

However, there may be some consistency rules for which it is not productive to separate the existence and the agreement relationships. In these cases the original, combined form will be used.

Finally, we shall label each rule so that it can be referred to by the ViewPoint's local process model. The process model provides guidance over when certain rules ought to be applied. It also specifies any relationships between rules, and it associates resolution actions with various rules. The labels are chosen arbitrarily and are local to the ViewPoint template in which the rule is defined[1].

5.3. Example Inter-ViewPoint Consistency Rules

5.3.1. Existence rules

There are two kinds of existence rules at the inter-ViewPoint level, depending on whether reference is made to the contents of the source ViewPoint. In some cases, the mere existence of the source

[1] Note that in this paper, we have labelled the example rules consecutively for convenience. In practice, the rules given would not appear in a single ViewPoint template, as they are drawn from several methods and refer to several different notations.

ViewPoint requires the existence of another related ViewPoint. An example is the rule *"Every Z schema must have a textual description"*. This type of rule can be expressed using null partial specifications, denoted by ∅. By convention, the partial specification for the destination ViewPoint can be omitted entirely. Every ViewPoint containing a Z schema would contain a rule of the form:

R_1: $\varnothing \rightarrow \exists\ VP_D\ (TD, D_s)$

Where TD is the template for 'textual description' and D_s means that the domain of the textual description ViewPoint should be the same as that of the source ViewPoint.

The second kind of existence relationship covers situations in which elements of the specification in one ViewPoint require other related ViewPoints to exist. An example is *"Every non-primitive process in a DFD must have a decomposition DFD associated with it"*. In this case only the partial specification of the destination ViewPoint will be null:

R_2: {Process.Status.Nonprimitive} $\rightarrow \exists\ VP_D$ (DFD, Process.Name)

Where DFD is the template for 'dataflow diagram' and Process.Name indicates that the domain of the decomposition ViewPoint should be the name of the process it represents.

Each of the types of rule described above can also be negated, for instance to specify that an element of a ViewPoint specification should not have another ViewPoint associated with it, or that a particular ViewPoint should be unique. Examples are *"A primitive process in a DFD should not be decomposed":*

R_3: {Process.Status.Primitive} $\rightarrow \neg \exists\ VP_D$ (DFD, Process.Name)

and, in an agent hierarchy ViewPoint, *"There should be only one agent hierarchy diagram":*

R_4: $\varnothing \rightarrow \neg \exists\ VP_D$:(AH, D_a)

where D_a indicates that the domain of the destination ViewPoint can be anything. It is important to note that consistency rules are always applied from a source ViewPoint, and the source ViewPoint will never be checked for consistency with itself (i.e. VP_D will never be instantiated as VP_S). Self-consistency is checked at the in-ViewPoint level, using a separate set of rules. Without this arrangement, rules like R_4 would always fail.

5.3.2. Agreement Relationships

In general, agreement rules express relationships between the contents of two ViewPoints. An obvious example is the relationship between the flows connected to a process in a DFD and the contextual flows in the decomposition of that process. An example consistency rule for the parent ViewPoint is *"Every output from a process in a DFD must appear as a contextual output in every decomposition of that process":*

R_5: $\forall\ VP_D$(DFD, From.Name) { link(From, _).Flow.Name =
VP_D: link(_, context).Flow.Name }

Where the underscore is used to denote 'any'; "link(A, B)" is an object in the DFD notation linking process A to process B; and the dot notation is used to extract attributes and values from the 'link' object.

Note that the destination ViewPoint is now universally quantified, in contrast to the existence rules defined above. Hence an agreement rule does not require the related ViewPoint to exist: a separate existence relationship expresses this. It also allows for the possibility that several alternative ViewPoints exist, for example where two conflicting ViewPoints have been proposed.

As well as expressing equality, the relationship might express exclusion, such as the rule *"Process names must be unique across all DFDs":*

R_6: $\forall\ VP_D$ (DFD, D_a) {Process.Name $\neq$ VP_D: Process.Name}

Note that this rule does not exclude duplicate process names within a single DFD, as the destination ViewPoint will never be instantiated to be the same as the source ViewPoint.

5.3.3. Handling 'Global' Consistency Rules

Consistency checks can not exist outside of some ViewPoint: if they could it would violate the requirement for distributability. However, conceptually there are some checks which we wish to

perform without knowing whether any of the ViewPoints being checked actually exist. An example of such a rule is *"there must be an agent hierarchy"*. To handle such rules, the method designer could create a template for a ViewPoint which has as its specification a graph representing other ViewPoints and the relationships between them. Such a ViewPoint is also useful as a browser for the current set of ViewPoints.

There might be any number of such ViewPoints to contain different management information. For instance, there may be one for each template, to keep track of relationships between all ViewPoints instantiated from that template. Alternatively, there may be just one for the entire collection. The choice is up to the method designer.

Because of the distributed nature of the ViewPoints framework, there is no guarantee that the specification in such a ViewPoint accurately represents the current set of ViewPoints: a graph representing other ViewPoints may get out of date. Inter-ViewPoint rules can be defined to check whether the graph is up-to-date. In-ViewPoint rules in this type of ViewPoint act as global checks over the set of ViewPoints represented.

For example, consider a ViewPoint that keeps track of all ViewPoints containing dataflow diagrams. A simple inter-ViewPoint rule in this ViewPoint might be *"Every node in the graph represents a dataflow ViewPoint":*

R_7: $\{Node\} \rightarrow \exists\ VP_D(DFD, Node.Name)$

This viewpoint might also need the rule *"Every dataflow ViewPoint is represented as a node in the graph":*

R_8: $\forall\ VP_D(DFD, D_a)\ \{Node.Name = VP_D: D_a\}$

Where D_a is the domain of the destination ViewPoint.

We can now express global rules using in-ViewPoint checks in this ViewPoint. For example, the rule *"there must be one top-level dataflow diagram"* is an in-ViewPoint check to test that there is only one node that has no parent.

6. When should relationships between ViewPoints be checked?

During the development of a ViewPoint, the ViewPoint owner will invoke a consistency rule whenever she needs to establish that the relationship expressed by the rule holds. This may be to move to a different stage in the development, or to test some property that depends on the relationship with another ViewPoint. The checking is always performed from the context of one of the ViewPoints: there is no central control.

Guidance about when particular rules should be applied comes from the local process model associated with the source ViewPoint. The process model defines the conditions under which the ViewPoint's consistency rules can be invoked, and the possible outcomes from invoking them. Note that applying a consistency rule only determines that a relationship currently holds: it does not guarantee that it will continue to hold. Hence, a particular rule may need to be applied a number of times during the development of a ViewPoint.

6.1. The process model

The process model is expressed in terms of preconditions and post-conditions for various actions. We adopt the following notation:

$$\text{preconditions} \Rightarrow [\text{agent, action}]\ \text{post-conditions}$$

For the examples here, the action is the application of a consistency rule, and it is always applied by the source ViewPoint. Pre- and post-conditions will be lists of predicates.

The preconditions to an entry in the process model come from two sources. The first source is information about the state of the ViewPoint. These are used to restrict the stages of development in which an action can be performed. For example, it might be necessary to prevent an inter-ViewPoint rule being applied unless certain in-ViewPoint rules have been applied. The second source of preconditions is information about relationships with other ViewPoints. These will normally be generated as post-conditions of other rules.

In general, the post-conditions of applying rule R_i will be a list of n instantiations of relationship $\Re_i$, contained in the rule, expressed as a set of predicates of the form $\Re_i(\sigma, \delta)$, where σ and δ are the specification items that matched ps_S and ps_D. If no partial specifications in the source and destination ViewPoints match the rule, then this set of relationships will be empty. We denote the set by:

$$\{\Re_i(\sigma_1, \delta_1),...,\Re_i(\sigma_n, \delta_n)\}.$$

For example, in the scenario, at diagram 3, the result of applying rule R_5 in Anne's ViewPoint will be a set of predicates describing the relationships between the output flows d2 and d4:

$$\Re_5(d2, VP_D(DFD, B):d2) \wedge \Re_5(d4, VP_D(DFD, B):d4)$$

If a rule fails, we need to record the nature of the failure. An agreement rule fails if the relationship does not hold for any items that match ps_S and ps_D. Note that an agreement rule can succeed for some items and fail for others. We represent failures using predicates of the form 'inconsistent(σ, δ, R_i).'

Hence the result of applying rule R_i to the partial specifications ps_S and ps_D is a set of items for which the relationship holds and a set of items for which the relationship should hold but does not. Hence the default entry in the process model for the agreement rule R_i is as follows:

$$\{\text{preconditions}\} \Rightarrow \quad [VP_S, R_i] \quad \{\Re_i(\sigma_1, \delta_1),..., \Re_i(\sigma_n, \delta_n)\} \cup \{\text{inconsistent}(\sigma_1, \delta_1, R_i),..., \text{inconsistent}(\sigma_m, \delta_m, R_i)\}$$

An existence rule fails when the destination ViewPoint is missing, or when no ViewPoint is found which contains a specification containing the pattern ps_D. Hence, existence rules have entries in the process model of the form:

$$\{\text{preconditions}\} \Rightarrow \quad [VP_S, R_i] \quad \Re_i(\sigma, \psi) \vee \text{missing}(\sigma, VP_D\langle t, d\rangle, R_i) \vee \text{missing}(\sigma, VP_D\langle t, d\rangle{:}ps_D, R_i)$$

Where σ is the actual item in the source ViewPoint that matched ps_S and ψ is the destination ViewPoint that satisfied the existence criteria. The 'missing' predicate can be read as "No ViewPoint of template t and domain d (and containing partial specification ps_D) was found to meet the existence criteria associated with partial specification σ as required in rule R_i".

6.2. Interactions between rules

One important aspect of the guidance provided by the process model is to do with interactions between consistency rules. In many cases, the applicability of some rule will be affected by some other rule. Linkages between rules allow the method designer to express complex interactions between ViewPoints such as *"rule B should be invoked only if relationship A holds"* and *"relationship A should hold unless rule B is applicable"*. Note here the distinction between a relationship holding and a rule being applicable. Each rule R_i expresses a relationship $\Re_i$. Hence to say that $\Re_i$ holds at a particular instance is to say that the rule has been applied and was satisfied. To say that a rule is applicable only states that any necessary preconditions are met.

We use the process model to identify such dependencies. Consider a rule for a dataflow diagram, which relates a decomposition ViewPoint to its parent, such as the inverse of R_5 above. Firstly, we cannot take the existence of the parent for granted, and we cannot assume that only one version of the parent exists. Hence, we express an existence rule to ensure that there is a parent[2], *"there must be a parent DFD containing the process represented by this DFD":*

R_9: $$\varnothing \rightarrow \exists\ VP_D(DFD, D_d)\ \{Process.Name.D_s\}$$

Where D_s is the domain of the source ViewPoint, which the rule says should appear as one of the processes in any parent ViewPoint.

We then define separately any rules which express relationships between the decomposition and its parent, such as *"contextual outputs in a DFD must have the same names as the outputs from the*

[2] Ignoring, for now, the possibility that the source ViewPoint might be the top level, and hence have no parent.

process in the parent DFD":

R_{10}: $\forall VP_D(DFD, D_d)$ { link(_, To.Name.'context').Flow.Name = VP_D: link(D_s, _).Flow.Name }

This second rule should only be applied where the destination ViewPoint is the parent, as established in R_9. Hence, the process model will specify that R_{10} should only be applied to destination ViewPoints for which R_9 has been successfully applied:

$\Re_9(\varnothing, \psi) \Rightarrow$ $[VP_S, R_{10}]$ $\{\Re_{10}(\sigma_1, \psi{:}\delta_1),..., \Re_{10}(\sigma_n, \psi{:}\delta_n)\} \cup$
$\{inconsistent(\sigma_1, \psi{:}\delta_1, R_{10}),..., inconsistent(\sigma_m, \psi{:}\delta_m, R_{10})\}$

Note that the precondition, $\Re_9$, defines the destination ViewPoint, ψ, to which rule R_{10} applies. The post-condition is a set of partial specifications for which $\Re_{10}$ holds and a set of partial specifications that are inconsistent. Both sets could be empty, if nothing in the ViewPoints' specifications matched the patterns in the rule.

A more complex example is provided by the rule *"Dataflow names must be unique across all DFDs unless related across a decomposition"*. As we have seen above, there are several ways that dataflow names can be related across a decomposition, including those specified in R_5, R_9 and R_{10}, and some others to deal with input flows, which we will ignore for now. Hence, we first express just the uniqueness rule:

R_{11}: $\forall VP_D(DFD, D_d)$ {link(_, _).Flow.Name ≠ VP_D: link(_, _).Flow.Name}

We then link it in the process model to the rules that specify the exceptions. When applying R_{11}, we are not interested in the set of partial specifications for which the rule holds, as this is just an exhaustive list of pairs of different dataflow names. However, we are interested in any partial specifications for which the relationship does not hold. Hence, the entry in the process model will be:

$[VP_S, R_{11}]$ $\{breaks(\sigma_1, \delta_1, R_{11}),..., breaks(\sigma_m, \delta_m, R_{11})\}$

Partial specifications for which the relationship does not hold are not necessarily inconsistent, as one of the exceptions may apply. Hence, we have introduced a predicate 'breaks', to indicate partial specifications for which the relationship should hold but does not. We then introduce another entry in the process model, that allows us to detect an inconsistency once we have also checked all the exceptions:

$breaks(\sigma, \delta, R_{11}) \Rightarrow$ $[VP_S, (R_5; R_{10})]$
$irrelevant(\sigma, \delta, R_5) \wedge irrelevant(\sigma, \delta, R_{10}) \rightarrow inconsistent(ps_S, ps_D, R_{11})$

Where $irrelevant(\sigma, \delta, R_i)$ is true if neither $\Re_i(\sigma, \delta)$ nor $inconsistent(\sigma, \delta, R_i)$ are true.

These entries in the process model state that we cannot conclude there is an inconsistency between ps_S and ps_D just from applying R_{11}. The inconsistency can only be demonstrated if we know that relationships $\Re_{11}$ does not hold, and we have tested to see that none of the exceptions apply.

7. How are relationships between ViewPoints checked?

When a consistency rule is applied, both the ViewPoints involved must co-operate to perform the check, and both ViewPoints need to know the result. Each ViewPoint will record, in its work record, the fact that the rule was invoked, and the outcome. This information may be used later to reason about actions that might resolve inconsistencies.

The ViewPoints might be evolving asynchronously, and hence the invocation and application of the rule need to be performed as a single action. We use a transaction management system to control the process. An inter-ViewPoint communication protocol specifies the checking process.

7.1. Transaction Management

By applying an inter-ViewPoint rule, we can determine whether a relationship holds, or whether there is an inconsistency between two ViewPoints. However, applying the rule does not ensure that any relationship or inconsistency will continue to hold as the two ViewPoints evolve. For instance, if ViewPoint A discovers there is an inconsistency with ViewPoint B, A has no way of knowing whether B did something that fixed the inconsistency after the rule was applied. In fact all

A knows is that there was an inconsistency *at the time* when the rule was applied.

Because of the asynchronous development of ViewPoints, we often need to ensure that several inter-ViewPoint actions are carried out as a single transaction. For example, in the previous section, rule R_{10} could only be invoked if it is known that relationship $\Re_9$ holds. If the source ViewPoint invokes R_9 at some instance, it establishes that $\Re_9$ held at that instance. Later, if R_{10} needs to be checked, it must invoke R_9 again to establish that $\Re_9$ still holds, and then invoke R_{10}. These two invocations must be carried out as a single transaction.

7.2. Communication Protocol

Application of the inter-ViewPoint consistency rules is achieved through a node-to-node interaction protocol. The protocol provides the set of rules by which all ViewPoint synchronisation and communication take place.

Application of a rule involves comparing partial specifications from each of the ViewPoints, possibly after some transformations have been applied. Identifying the relevant partial specification must be done locally by each ViewPoint, as the style and structure of a ViewPoint's specification may not be visible to other ViewPoints.

We will not describe the protocol in detail here. Essentially the sequence of actions is as follows. The source ViewPoint requests potential destination ViewPoints to identify themselves, and then transmits the rule, the partial specifications ps_S, and the pattern ps_D. The destination ViewPoint then applies the rule and transmits the results of applying the rule.

The protocol assumes the existence of a reliable communications network, and a distributed name service which helps locate and identify ViewPoints.

7.3. Example

To illustrate the application of consistency checking rules, we will demonstrate how inconsistency (4) from the scenario is handled. Firstly, note that the two ViewPoints of interest, A and B, each contain: a description, perhaps represented internally as shown below; a set of consistency checks; and a local process model to guide application of those checks.

ViewPoint A	**ViewPoint B**
process(x, primitive) process(y, non-primitive) link(y, x).Name.d2 link(x, y).Name.d3 link(y, external).Name.d4 link(y, external).Name.d5 (etc.)	process(y1, primitive) process(y2, primitive) link(context, y1).Name.d3 link(y1, context).Name.d2 link(y1, context).Name.d9 link(y2, context).Name.d6 link(y2, context).Name.d8 (etc.)
Relevant inter-ViewPoint rules: R_2 – each non-primitive process should be represented by another DFD containing the decomposition. R_5 – every output from a process must appear as a contextual output in the decomposition DFD (etc.)	**Relevant inter-ViewPoint rules:** R_9 – there must be a parent DFD containing the process represented by this DFD R_{10} – every contextual output must appear as an output from the process in the parent DFD (etc.)
Process Model: $[VP_S, R_2]$ {..., $\Re_2(\sigma_i, \psi_i)$,... } ∪ {..., missing(σ_i, VP_D<t, d>, R_2),... } $\Re_2(\sigma, \psi) \Rightarrow [VP_S, R_5]$ {..., $\Re_5(\sigma_i, \psi{:}\delta_i)$,... } ∪ {..., inconsistent(σ_i, $\psi{:}\delta_i$, R_5),...} (etc.)	**Process Model:** $[VP_S, R_9]$ $\Re_9(\varnothing, \psi) \vee$ missing($\varnothing$, VP_D<DFD, D_d>, R_9) $\Re_9(\varnothing, \psi) \Rightarrow [VP_S, R_{10}]$ {..., $\Re_{10}(\sigma_i, \psi{:}\delta_i)$,... } ∪ {..., inconsistent(σ_i, $\psi{:}\delta_i$, R_{10}),...} (etc.)

We have listed two of the consistency rules for each ViewPoint, and the corresponding entries in the process models. In each case, the process model requires the relationship specified by the first rule to hold as a precondition for the second rule. For ViewPoint A, this specifies that the decomposition ViewPoint for a process must exist before the correspondences between output flows can be checked. For ViewPoint B, the parent ViewPoint must exist before such correspondences can be checked. Note also that rules R_5 and R_{10} encode the same check, but from the perspective of each ViewPoint.

Consider first the application by ViewPoint A of rule R_5. The process model requires that this rule only be applied if the relationship specified by R_2 holds, i.e. if the decomposition ViewPoint exists. This precondition also identifies the decomposition ViewPoint, in preparation for application of R_5. Whenever R_5 is invoked, R_2 is automatically checked first, and both checks are performed as a single transaction. This ensures that the decomposition ViewPoint still exists, whether or not R_2 had been checked previously.

In this example, R_2 identifies the destination ViewPoint as ViewPoint B. R_5 then identifies the following relationships and inconsistencies:

$$\Re_5(d2, VP_D(DFD, B):d2)$$
$$\wedge\ inconsistent(d4, VP_D(DFD, B):link(\_, context), R_5)$$
$$\wedge\ inconsistent(d5, VP_D(DFD, B):link(\_, context), R_5)$$

These record that the relationship holds for the dataflow d2, but that for d4 and d5, the rule failed. Note that the first argument to the inconsistent predicate is the actual item that matched ps_S, whilst the second argument names the destination ViewPoint, and gives the partial specification (ps_D) for which no match was found.

8. How are inconsistencies resolved?

The resolution process is concerned with establishing a relationship between two ViewPoints. Resolution only becomes necessary if a consistency check failed, *and* the ViewPoint owner wishes to correct this. In many cases, resolution will not be necessary after the failure of a rule, because the inconsistency can be tolerated.

The goal of inconsistency resolution is to (re-)establish the relationships contained in the rule or rules which failed. If a relationship did previously hold, information about subsequent changes can be used to guide the resolution process. This information is available in the work record of each ViewPoint, along with a record of the results of previous consistency checks.

During the resolution of an inconsistency, the ViewPoint owners may wish to define new relationships between the ViewPoints, which are not encoded in any of the consistency rules. These are specific relationships which only apply to the two ViewPoints involved, or which are not expected to hold generally. Such relationships are also recorded in the work record, so that future changes which affect these relationships can be monitored.

Various actions may be taken by the ViewPoint owners during the resolution process. Some actions will alter one or other of the ViewPoints. Other actions might not alter the ViewPoints, but may analyse the nature of the inconsistency. The process may entail one ViewPoint owner requesting the other to take a particular action. When a sequence of actions resolves the inconsistency, both ViewPoints are notified, for the same reason that both are notified of the results of any consistency checks.

Our approach to supporting the resolution process is through the provision of a set of potential resolution actions, which the ViewPoint owners may wish to apply. The actions are defined by the method designer, as part of the process of defining the consistency relationships. Possible resolution actions are associated with each consistency rule in the process model. In this way, each rule will have a number of actions that may be performed in the event that the rule fails. Guidance for selecting among these actions is derived from information in the process model, along with information about the history of the ViewPoints in question.

8.1. Conflict and Inconsistency

To understand the resolution process, it is helpful to be clear about what is being resolved. We distinguish between inconsistency and conflict. An *inconsistency* occurs if a rule has been broken.

The rules are entered by the method designer, to specify the correct use of the method. Hence, what constitutes an inconsistency in any particular situation is entirely dependent on the rules entered during the method design. Rules will cover the correct use of a notation, and the relationships between different notations.

Conflict is 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. The definition says nothing about whether the conflict is intended by either party. Finally 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.

Inconsistency is a property of the state of a collection of ViewPoints. Conflicts and mistakes are properties of the actions that ViewPoint owners take on their ViewPoints. In other words, a given specification can be inconsistent, while actions on that specification may be mistaken or conflictual. Hence, we can test a specification for the existence of inconsistency, but we cannot test for conflicts or mistakes. Each inconsistency is considered to be either the result of a conflict between the ViewPoint owners[3], or the result of a mistake. Note that a mistaken or conflictual action might not necessarily result in any inconsistency in the set of ViewPoints.

8.2. Supporting Resolution

Consider the inconsistencies that arose in our scenario. The inconsistencies in figures 1 and 5 are identical, in that the same consistency rule is broken: no ViewPoint exists to represent the non-primitive process. In figure 1 the appropriate resolution is to create the missing ViewPoint. In figure 5 the appropriate resolution is either to delete the process or to mark it as non-primitive. Hence, there are at least three actions that might be offered to the ViewPoint owner when this particular check fails.

Furthermore, the choice between these three actions can be narrowed by reasoning about the history of the two ViewPoints. In particular, the key difference between the two cases is that for the latter one the relationship in question did hold at some point in the past. For this particular relationship, if it has never held in the past, it is likely that the decomposition ViewPoint has not yet been created. If it has held, this indicates that the decomposition ViewPoint has since been deleted. Note that in either case, the result is not conclusive. For example, if the decomposition ViewPoint was created and deleted without the check ever being applied, there will be no record that the relationship did hold at one point. Accordingly, this type of reasoning is used only to recommend a default action, and not to resolve the inconsistency automatically.

8.3. Expressing Resolution Actions

Each resolution action has the following components:

• A short label allows the action to be presented within a menu of possible actions.

• A piece of text explains the rationale for the action. This explanation should assist the method user in deciding whether the action is appropriate.

• A piece of code which performs the action. In some cases this will perform an edit on the ViewPoint's description. In other cases it will merely invoke one of the tools provided to support conflict resolution, or request the other ViewPoint owner to perform some action.

The process model associates preconditions and postconditions with each action. The preconditions determine the context under which the action is appropriate. For instance, some preconditions will associate the action with the failure of one or more consistency checks, whilst other preconditions may further restrict the applicability of the action. The post-conditions define the results of applying the action. These indicate whether the action fixes any inconsistencies, or whether it sets up conditions for other actions to be applied.

---

[3] Where two ViewPoints share the same owner, an inconsistency between them may indicate the owner is in conflict with herself.

The range of possible actions is large. Possibilities include: a transfer of information from one ViewPoint to another; a name change to prevent a clash or bring two ViewPoints into agreement; an analysis of the situation to determine whether conditions hold for more specific actions; invocation of tools to support negotiation. Ultimately, the user will select from among the actions suggested whenever an inconsistency is revealed by the checks, and in some cases a sequence of actions will be necessary. The user is also free to ignore the suggested actions.

In the scenario, if rule R_5 fails, there is a dataflow in the source ViewPoint for which no match can be found in the destination ViewPoint. In this case, some of the possible resolution actions are: copy the dataflow to the destination ViewPoint; delete the dataflow in the source ViewPoint; rename a flow in the destination ViewPoint; rename the flow in the source ViewPoint. Note that either of the first two of these alone remove the inconsistency, while the latter two might not. An actual resolution process might involve some combination of renaming, transfer, and deletion actions. Changes to the destination ViewPoint are sent as requests to the destination ViewPoint owner.

8.4. Designing Resolution Actions

The method designer may need assistance in identifying appropriate actions, and linking them to consistency rules. One could argue that support for the method designer is a low priority, as method design is a relatively rare activity, performed by experts. However, the task of defining actions is open-ended, and effort invested here will reduce the load on the method users. The more the set of actions are refined, the less effort the method user will have to expend in choosing between them.

Consistency rules and the resolution actions associated with them will be generated from four main sources:

Firstly, consideration of the rationale and operation of the method will provide a set of basic checks and actions. These ensure that the method is being used correctly.

Secondly, consideration of examples and case studies of the use of the method will provide further evidence of possible inconsistencies. In particular, reference to examples will help to refine the set of resolution actions and the conditions under which they apply.

Thirdly, further checks and actions can be derived from the experiences of method users. This exploits the fact that software designers already handle routine conflicts. Where a method designer is implementing an existing method, current users of the method can be used. If the method is new, then related methods can be used as a source of expertise. In either case, knowledge elicitation techniques can be applied to discover heuristics used by these experts. Note that quite apart from the difficulties of elicitation, the approaches used by these ‘experts’ need to be evaluated carefully, as there is no guarantee they will be appropriate or sensible. In particular, existing software practices tend to suppress conflict. Furthermore, this approach will not offer any insight into conflicts that only emerge as a result of using the ViewPoints framework (see the comments on negotiation below).

Finally, there are a set of inconsistency exploration tools which can be applied to any inconsistency. An example is the Computer Supported Negotiation tool described in Easterbrook (1991), which allows ViewPoint owners to identify correspondences between their disparate descriptions, and evaluate a number of potential resolutions.

Because the task is open-ended, further consistency checks and potential resolution actions might be identified as a result of method use. We recognise that method design and method use take place within a larger cycle of method refinement, in which lessons learned from one software engineering project are used to improve the method for subsequent projects. Our approach supports this form of method refinement in the sense that it makes many aspects of a method explicit, including the relationships between the notations used in the method, the development actions that apply to each notation, and the preconditions and postconditions of each action.

9. What happens if inconsistencies are not resolved?

As inconsistencies are tolerated, there is no obligation to repair them immediately. The resolution process for any particular inconsistency can be delayed indefinitely, especially if the effort of

resolving it may prove unnecessary. For example, the inconsistency might be in a part of the ViewPoint that is only tentative; it may be the result of a known conflict that the owners are not yet ready to resolve; or some anticipated future action will resolve it anyway. If the resolution process is entered, there is no obligation to continue applying resolution actions until the inconsistency is removed. It may be useful to take some steps towards a resolution and delay the remainder.

An important consideration is that resolving an inconsistency does not ensure it stays resolved. Successful application of a consistency check confirms a relationship holds between two ViewPoints. Resolving an inconsistency achieves the same result. It does not entail merging ViewPoints, nor does it lock the ViewPoints into the relationship. ViewPoint developers have the freedom to continue to evolve their ViewPoints independently. Hence, establishing a relationship through inconsistency resolution does not guarantee that future changes will not interfere with that relationship.

The problem of subsequent changes affecting a relationship is dealt with by recording the relationships as they are checked. Subsequent changes to the ViewPoint can be analysed for their effect on the relationship. If a change to the ViewPoint upsets an established relationship with another ViewPoint, this fact is recorded in the work record. The information is then available for the owners to implement a stronger locking strategy, if they wish, or as a record to be saved for some future explicit resolution process.

Tolerance of inconsistencies offers flexibility both in terms of development strategy applied, and of division of responsibility. However, it also introduces problems in that inconsistencies may accumulate. Resolution may be fairly straightforward if only one or two rules have been broken. However, if large numbers of rules have been broken, it may be hard to find appropriate resolution actions. Our approach to this problem is incremental resolution. ViewPoint owners may choose to resolve only some of the inconsistencies between their ViewPoints and ignore others. In this way the relationships gradually become disentangled over a period of time.

9.1. Evolution of relationships

Resolution of inconsistency synchronises divergent ViewPoints, but does not prevent them from subsequently diverging again. The inter-ViewPoint communication protocols allow ViewPoint owners to initiate consistency checking and resolution whenever they wish. When a relationship is established between two ViewPoints, details of this relationship are recorded so that if subsequent actions affect the relationship, the support tools can detect this. The ViewPoint owners may change their ViewPoints without worrying about the effects on other ViewPoints. However, the recorded information ensures that effort spent resolving an inconsistency is not lost as the ViewPoints continue to evolve.

For example, in the scenario, when rule R_{10} is applied by ViewPoint B, at the point when the ViewPoints are consistent (figure 3), the rule detects, among other things, that the dataflows labelled d4 in each ViewPoint are equivalent. The following relationship is recorded:

$$\Re_5(\text{d4}, \text{VP}_\text{D}\ (\text{DFD}, \text{A}):\text{d4})$$

Subsequently, Bob renamed the dataflow d4 to d8. This re-labelling affects not just the dataflow in the description, but also any inter-ViewPoint relationships which depend upon it. Hence, any occurrence of the label d4 in the list of relationships is replaced with the new label:

$$\Re_5(\text{d8}, \text{VP}_\text{D}\ (\text{DFD}, \text{A}):\text{d4})$$

In this way, when the rule R_{10} is next checked, there is already a record that the relationship has evolved. This in turn indicates that the most likely resolution action should be to propagate the change to the other ViewPoint. However, this action may not always be the one chosen. The ViewPoint owners may negotiate a new label to use, or, if the method permits it, define the two labels as synonyms.

Finally, the propagation of changes to other ViewPoints may sometimes be incorrect. For instance, the fact that two labels were once the same does not always imply that they should remain the same. Consider the case where a ViewPoint owner changes the name of a label, then later discovers she has accidentally chosen a label used in another ViewPoint, and so changes it back again. A consistency check applied between the changes will discover that there is an equivalence, but it is a relationship that should not be maintained.

9.2. Dealing with accumulated inconsistencies

Allowing inconsistencies to accumulate can cause problems when it comes to resolution. The difficulty is that ViewPoints might be arbitrarily different, with no common basis whatsoever. In the worst case, every single inter-ViewPoint consistency rule between two ViewPoints may fail. We offer two approaches to this problem. The first is a technique described in Easterbrook (1991) which allows two ViewPoint owners collaboratively to compare arbitrarily different ViewPoints, and analyse the differences between them.

An alternative approach is through incremental resolution. If two ViewPoints have a large number of inconsistencies, the most practical way forward might be to resolve one or two and ignore the rest. In this way two disparate ViewPoints may gradually converge over a period of time. This approach is especially useful where actions by several parties might be necessary for a complete resolution.

As an example of incremental resolution to deal with accumulated inconsistencies, consider again the inconsistency in figure 4, and imagine Anne is trying to resolve it. Several changes have been made to both ViewPoints. Merely comparing the ViewPoints does not provide enough information to derive a resolution: Anne's ViewPoint has flows labelled d4 and d7 which should match Bob's flows labelled d6, d8 and d9.

From the record of past relationships, described in the previous section, it can be determined that d4 and d8 were once equivalent. Anne can resolve this inconsistency by re-labelling her d4 to match Bob's change. She may then decide that the remaining inconsistencies need to be resolved in a face-to-face meeting. However, in preparation for this she sends a message to Bob's ViewPoint suggesting he re-label his d6 as d7, and that they get together to discuss d9. Anne's actions bring the ViewPoints closer together, but do not entirely resolve them. They may, however, facilitate subsequent actions by Bob to bring them closer still.

10. Related Work

System specification from multiple perspectives using many different specification languages has become an area of considerable interest. The integration of methods, notations and tools has generally been addressed by the use of a common data model, usually supported by a common, centralised database (Alderson, 1991; Wasserman & Pircher, 1987). Recent work by Jackson & Zave (1993) proposes the composition of partial specifications as a conjunction of their assertions in a form of classical logic. A set of partial specifications is then consistent if and only if the conjunction of their assertions is satisfiable.

Other authors have also considered multi-perspective or multi-language specifications. In Wileden et al. (1991) specification level interoperability between specifications or programs written in different languages or running on different kinds of processors is described. The interoperability described relies on remote procedure calls and ways that interoperating programs manipulate shared typed data. Wile (1991) on the other hand uses a common syntactic framework defined in terms of grammars and transformations between these grammars. He highlights the difficulties of consistency checking in a multi-language framework.

Traditionally, multiparadigm languages, which deploy a common multiparadigm base language, have been used to combine many partial program fragments (Hailpern, 1986), while more recently the use of a single, common canonical representation for integrating so-called 'multi-view' systems has been proposed (Meyers & Reiss, 1991).

11. Conclusions

We have presented a framework for concurrent engineering in which there is no requirement for consistency maintenance, and no central database or common data schema. The framework is fully distributable, in that local objects ('ViewPoints') encapsulate sufficient development knowledge to act as independent specification development tools. The descriptions contained in different ViewPoints may be developed concurrently. Multiple notations and a diversity of development strategies are encouraged.

The current status of the work is that we have implemented a prototype environment to support the ViewPoints framework, which we are now using as a testbed in which to explore the issues

described in this paper. Several software engineering methods have been implemented, and experience with the process of method design has been valuable in refining our approach (Nuseibeh, Finkelstein, & Kramer, 1994). For each of the issues raised by the scenario in this paper, we have sketched out the approach and tested it on small examples. We have devised further experiments for each of the issues described, and are currently investigating the applicability of the approach using larger examples.

11.1. Advantages

This approach to computer support for concurrent software engineering provides the flexibility to support distributed activities without assuming perfect communication links. Inconsistencies are tolerated, allowing separate designers to pursue their ideas without being constrained because of conflicts with other members of the team. Inconsistencies are explicitly resolved at appropriate stages, and guidance is provided for resolution through local process models. Resolution of inconsistency does not prevent the ViewPoints becoming inconsistent again, but information about the relationship is retained to assist repairing subsequent inconsistencies.

An important advantage of this approach is that it more accurately reflects actual working practices. Tolerance of inconsistency allows actions affecting more than one ViewPoint to be de-coupled. This facilitates distributed working by allowing responsibility to be devolved to individual ViewPoints. All decisions regarding development of a ViewPoint, including decisions about resolving inconsistencies with other ViewPoints, are taken locally. The principle of local action and local responsibility is further reinforced by the provision of a local process model in each ViewPoint, which guides the development of that ViewPoint.

The approach also permits reasoning in the presence of inconsistency. Local reasoning within a ViewPoint is always possible, whether or not the ViewPoint is consistent with other ViewPoints. Furthermore, by explicitly expressing the relationships that should hold between ViewPoints, it is possible to reason about individual relationships, while ignoring others. For example, possibilities for resolving a particular inconsistency can be explored, without having to devise a complete resolution, and without having to worry about whether other consistency relations hold.

11.2. Future Work

There are a number of remaining problems with the framework which we are investigating. The first of these is support for discussion about ViewPoints by the ViewPoint owners. This is especially needed in the early stages of inter-ViewPoint inconsistency resolution: ViewPoint owners may need to make suggestions, requests and comparisons between ViewPoints.

An interesting possibility is to view the consistency relations as constraints in the conventional AI sense and to use constraint satisfaction techniques to maintain and to reason with such relations. For instance, Galileo3 (Browen & Bahler, 1992) has demonstrated how to use constraint networks to enforce relationships between design artefacts and to support system-mediated negotiation between users of different perspectives.

The ViewPoint framework offers a relatively unconstrained model of collaboration. The model is essentially of a group of peers developing a set of loosely coupled ViewPoint descriptions. This model is a sufficient basis for some types of conflict resolution approaches. In particular, it is sufficient for approaches which ignore the nature of the working relationship between ViewPoint owners, and the physical and temporal properties of their interaction. Stronger models of collaboration may be needed if more prescriptive support is to be offered. One possible approach is to model explicitly the relationships between ViewPoint owners.

So far we have treated ViewPoint owners as if they are always peers. This is not always the case, and it is possible that power relationships between ViewPoint owners will undermine or subvert the resolution processes supported by the tools. It may be possible, by explicitly modelling the relationships between ViewPoint owners, to make allowances for these (Gotel & Finkelstein, 1994a; Gotel & Finkelstein, 1994b).

Acknowledgements

The work described in this paper was partly funded by the UK Department of Trade and Industry (DTI), as part of the Advanced Technology Programme (ATP) of the Eureka Software Factory

(ESF). The authors would like to acknowledge the constructive comments of Martin Feather, Fox Poon and Amer Al-Rawas on an earlier version of this paper.

References

Alderson, A. (1991). Meta-CASE technology. In Endres & Weber (Ed.), *European Symposium on Software Development Environments and CASE Technology*, Königswinter, June 1991, (vol. LNCS 509, pp. 81-91). Springer-Verlag.

Browen, J., & Bahler, D. (1992). Frames, Quantification, Perspectives, and Negotiation in Constraint Networks for Life-Cycle Engineering. *International Journal of Artificial Intelligence in Engineering*, 7, 199-226.

Cutkosky, M. R., Engelmore, R. S., Fikes, R. E., Genesereth, M. R., Gruber, T. R., Mark, W. S., Tenenbaum, J. M., & Weber, J. C. (1993). PACT: An Experiment in Integrating Concurrent Engineering Systems. *IEEE Computer*, 26(1), 28-37.

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

Easterbrook, S. M., 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.

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), 31-57.

Finkelstein, A. C. W. F., Gabbay, D., Hunter, A., Kramer, J., & Nuseibeh, B. (1994). Inconsistency Handling in Multi-Perspective Specifications. *IEEE Transactions on Software Engineering (to appear)*.

Gotel, O. C. Z., & Finkelstein, A. C. W. (1994a). An Analysis of the Requirements Traceability Problem. In *Proceedings of the IEEE International Conference on Requirements Engineering (ICRE-94)*, Colorado Springs, April 1994.

Gotel, O. C. Z., & Finkelstein, A. C. W. (1994b). Modelling the Contribution Structure Underlying Requirements. In *Proceedings of the First International Workshop on Requirements Engineering: Foundations of Software Quality (REFSQ-94)*, June 1994.

Hailpern, B. (1986). Special issue on multiparadigm languages and environments. *IEEE Software*, 3(1), 10-77.

Jackson, M., & Zave, P. (1993). Domain Descriptions. In *IEEE International Symposium on Requirements Engineering*, San Diego, 4-6 January 1993, pp. 56-64. IEEE Computer Society Press.

Kramer, J., & Finkelstein, A. C. W. (1991). A Configurable Framework for Method and Tool Integration. In *Proceedings of European Symposium on Software development Environments and CASE Technology*, Königswinter, Germany, June 1991, (vol. LNCS 509, pp. 233-257). Springer-Verlag.

Meyers, S., & Reiss, S. P. (1991). A System for Multiparadigm Development of Software Systems. In *Proceedings of the Sixth International Workshop on Software Specification and Design*, Como, Italy, 25-26th October 1991, pp. 202-209.

Nuseibeh, B., & Finkelstein, A. C. W. (1992). ViewPoints: A vehicle for Method and Tool Integration. In *Proceedings of the IEEE International Workshop on Computer-Aided Software Engineering (CASE-92)*, Montreal, Canada, 6-10th July 1992.

Nuseibeh, B., Finkelstein, A. C. W., & Kramer, J. (1993). Fine-Grain Process Modelling. In *Proceedings of the Seventh International Workshop on Software Specification and Design (IWSSD-7)*, Redondo Beach, CA, 6-7 December 1993, pp. 42-46. IEEE Computer Society Press.

Nuseibeh, B., Finkelstein, A. C. W., & Kramer, J. (1994). Method Engineering for Multi-Perspective Software Development. *Information and Software Technology Journal*, (to appear).

Nuseibeh, B., Kramer, J., & Finkelstein, A. C. W. (1993). Expressing the Relationships Between Multiple Views in Requirements Specification. In *Proceedings of the 15th International Conference on Software Engineering (ICSE-93)*, Baltimore, 17-21 May 1993, pp. 187-200. IEEE Computer Society Press.

Nuseibeh, B., Kramer, J., & Finkelstein, A. C. W. (1994). A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering (to appear)*.

Wasserman, A. I., & Pircher, P. A. (1987). A Graphical, Extensible Integrated Environment for Software Development (Proceedings of 2nd Symposium on Practical Software Development Environments). *SIGPlan Notices*, 22(1), 131-142.

Wile, D. S. (1991). *Integrating Syntaxes and their Associated Semantics* (Technical Report No. RR-92-297). USC/Information Sciences Institute.

Wileden, J. C., Wolf, A. L., Rosenblatt, W. R., & Tarr, P. L. (1991). Specification-level interoperability. *Communications of the ACM*, 34(5), 72-87.