Co-ordinating Distributed ViewPoints the anatomy of a consistency check

STEVE EASTERBROOK
oo of Co . Co p tin . n . s r . rsit of ss. F . r Br ton B
. st. r roo o s s s

ANTHONY FINKELSTEIN, JEFF KRAMER & BASHAR NUSEIBEH
D. p rt . nt of Co p tin p. Co . . . . n s G t. on on BZ
f n o

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, *. t* ., 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.

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;
-

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

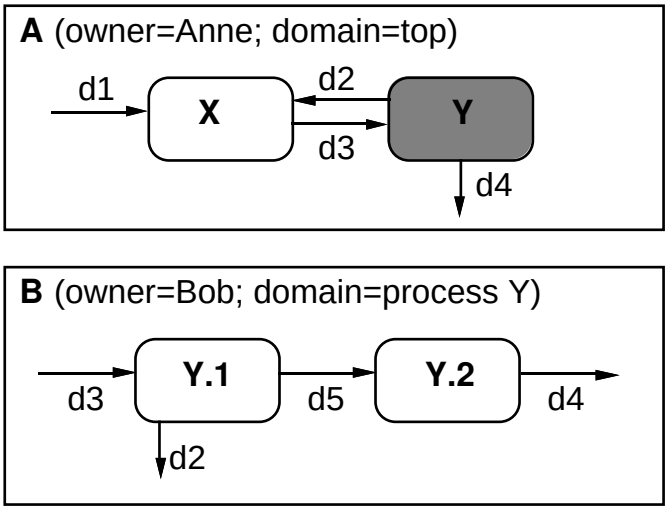
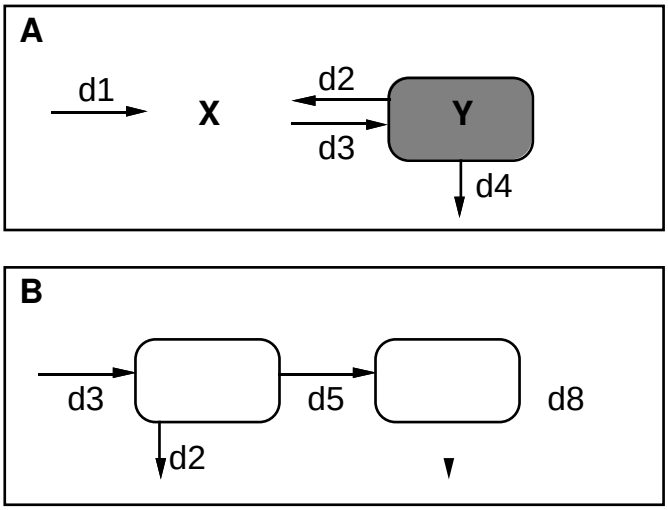


Figure 2 Point A is consistent with decomposition

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).



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 , for n onn, t, st s , t*	*E , for n , s , s , t*
Level 2 Inter-ViewPoint Rules	*E , for , st, n , of r, t, , oint*	*E , for ons, st, n , t , nfor , tion n r, t, , oint*

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 , t (1993; 1994) introduce a notation for inter-ViewPoint consistency rules based on

the expression of a relationship between a *so r* ViewPoint (VP_S) and a *sin tion* 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 'ArrowArro2d.j 02

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: $\emptyset \rightarrow \exists\ VP_D\ (TD, D_S)$

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 have a decomposition*.

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

and, in an agent hierarchy ViewPoint: *There should not be more than one agent with the same name*.

R_4: $\emptyset \rightarrow \neg \exists\ VP_D$:(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.

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 data flow from a process in a DFD must appear as a context flow in the 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 DFDs*

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

Consistency checks

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 *t r st n nt r r* . 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 *E r no n t r p r pr s nts t f o nt*

R_7: {Node} → ∃ VP_D(DFD, Node.Name)

This viewpoint might also need the rule *E r t f o nt s r pr s nt s no n t r p*

R_8: ∀ VP_D(DFD, D_a) {Node.Name = VP_D: D_a}

Where D_a

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 souu2na

pro ss nt p r nt 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, *D tf o n s st n q ross DFDs n ss r t ross o position*. 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(σ_1, δ_1, R_{11}),..., breaks(σ_m

A knows is that there was an inconsistency *t t , ƒ , ,* 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**

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:

$$\Re_5(\text{d2}, \text{VP}_D(\text{DFD, B}):\text{d2})$$
$$\wedge\ \text{inconsistent}(\text{d4}, \text{VP}_D(\text{DFD, B}):\text{link(\_, context)}, R_5)$$
$$\wedge\ \text{inconsistent}(\text{d5}, \text{VP}_D(\text{DFD, B}):\text{link(\_, context)}, R_5)$$

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.

. 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, *n* 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.

.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.

4.2. 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.

4.3. 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

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 ViewPointse it stsTng an inconsistency does nachievesemainsamlt of a ay beot ensure ita inmerg removeteps actionexion itution process
r.279emoveaccumushietsTng aproces usefulfairtatrerue in ura re ifeentatoneme a0.1 rulenohavrebo removeHowhver,ywf larg nnumbe noonstulenohavrebo

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

(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.), *E rop n pos on of t r D op nt En ron nts n CA E no o* , 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. *nt rn tion orn of Artf nt n n En n rn* , 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. *EEE Co p t r*, 26(1), 28-37.

Easterbrook, S. M. (1991). Resolving Conflicts Between Domain Descriptions with Computer-Supported Negotiation. *no A q s on An nt rn tion orn* , 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.), *C C Co op r tion or Conf t* (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. *nt rn tion orn of of t r En n rn n no En n rn* , 2(1), 31-57.

Finkelstein, A. C. W. F., Gabbay, D., Hunter, A., Kramer, J., & Nuseibeh, B. (1994). Inconsistency Handling in Multi-Perspective Specifications. *EEE r ns tions on of t r En n rn to pp r* .

Gotel, O. C. Z., & Finkelstein, A. C. W. (1994a). An Analysis of the Requirements Traceability Problem. In *ro n s of t EEE nt rn tion Conf r n on q r nts En n rn C E* , Colorado Springs, April 1994.

Gotel, O. C. Z., & Finkelstein, A. C. W. (1994b). Modelling the Contribution Structure Underlying Requirements. In *ro n s of t First nt rn tion or s op on q r nts En n rn Fo n tions of of t r t EF* , June 1994.

Hailpern, B. (1986). Special issue on multiparadigm languages and environments. *EEE of t r* , 3(1), 10-77.

Jackson, M., & Zave, P. (1993). Domain Descriptions. In *EEE nt rn tion pos on q r nts En n rn* , 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 *ro n s of E rop n pos on of t r op nt En ron nts n CA E no o* , 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 *ro n s of t t nt rn tion or s op on of t r p f tion n D s n*, 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 *ro n s of t EEE nt rn tion or s op on Co p t r A of t r En n rn CA E* , Montreal, Canada, 6-10th July 1992.

Nuseibeh, B., Finkelstein, A. C. W., & Kramer, J. (1993). Fine-Grain Process Modelling. In *ro n s of t nt nt rn tion or s op on of t r p f tion n D s n D* , 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. *nfor tion n oft r , no o o rn* , (to appear).

Nuseibeh, B., Kramer, J., & Finkelstein, A. C. W. (1993). Expressing the Relationships Between Multiple Views in Requirements Specification. In *ro , n s of t , t nt rn tion Conf r n on oft r En n rin C E* , 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. *EEE r ns tions on oft r En n rin to pp r* .

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

Wile, D. S. (1991). *nt r tin nt , s n t , r Asso t , nt s* (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. *Co ni tions of t , AC*