Co-ordinating Conflicting ViewPoints by Managing Inconsistency

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

1. Introduction

Our research centres on the problems of requirements modelling 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. Different teams of designers work in different locations, and the sources of information used in modelling requirements may be dispersed at different sites.

The task 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 often regarded as impractical because of the difficulties of maintaining consistency between a large collection of agents. However, our concern with distributed systems and distributed working leads us to regard such decentralised environments as both necessary and advantageous. Many of the problems can be overcome by recognising that maintaining global consistency at all times is an unnecessary burden. In a design process, 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.

Rather than trying to represent the people involved in the development process, we focus on the perspectives that they hold. With concurrent development, different perspectives may be at different stages of elaboration and may each be subject to different development strategies. The intersections between these perspectives are far from obvious because the knowledge within each perspective is represented in different ways. 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 perspectives problem".

Our recent work has focused on the problems of consistency checking in a distributed environment, and in particular, how conflicts are managed and resolved. The work is based on ViewPoints (Finkelstein, *et al*., 1992), which we use as a framework for supporting distributed software engineering. Using this approach, multiple perspectives are maintained separately as distributed objects, with each object also encapsulating representation knowledge and software process knowledge. Each ViewPoint has an *owner* and an identified *domain*. Each ViewPoint has a *style*, which is the notation used to develop a *specification;* a *work plan*, which describes the actions that can be performed on a description in that notation; and a *work record* which contains an annotated history of actions performed on the ViewPoint.

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.

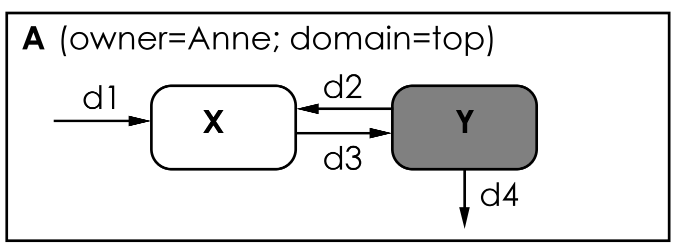
Most importantly, the framework tolerates inconsistency, with no requirement for changes to one ViewPoint to be consistent with other ViewPoints. Consistency checking is performed through a set of inter-ViewPoint rules, defined by the method, which express the relationships that should hold between particular 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).

2. Scenario

To illustrate the problems of co-ordinating ViewPoints, we will present a simple scenario using dataflow diagrams as an example notation. 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 the shown on the decomposition. However, the owners of each diagram may wish to modify them independently.

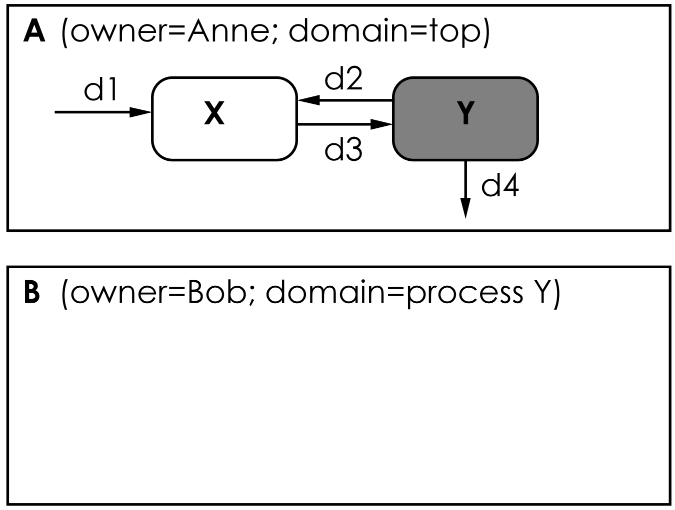
Consider the following. Anne has created a ViewPoint to contain a top level dataflow diagram of the system. Each process in the diagram is, by default, labelled as primitive, in that it is not decomposed further. She then selects process Y and changes its state to non-primitive, to indicate that it is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

At this point, there is an inconsistency:



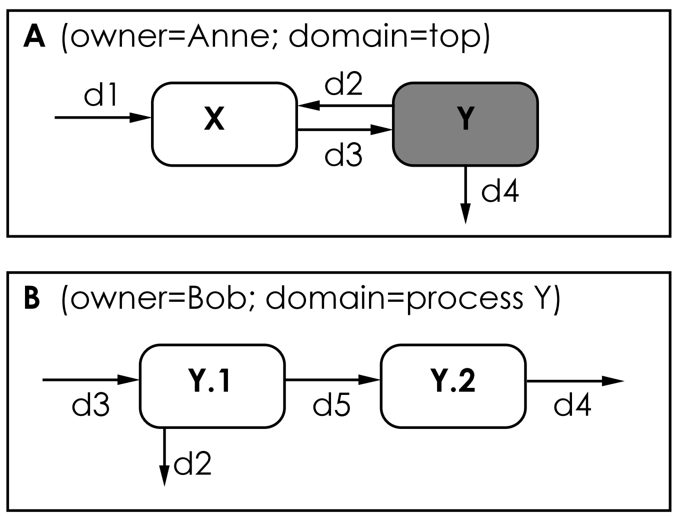
1. ViewPoint A contains a non-primitive process (shaded) for which no corresponding decomposition exists.

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.



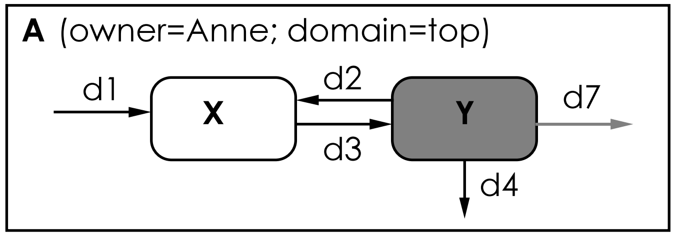
2. ViewPoint A contains a non-primitive process for which the decomposition does not have the same inputs and outputs (because the decomposition is empty).

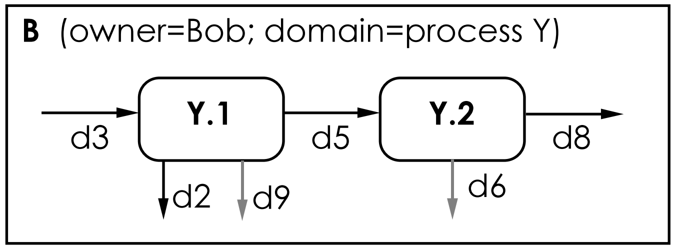
Bob copies the input and output flows connected to process Y from Anne's ViewPoint to his own. He then begins to define the processes that comprise the decomposition.



3. ViewPoint A is consistent with the decomposition.

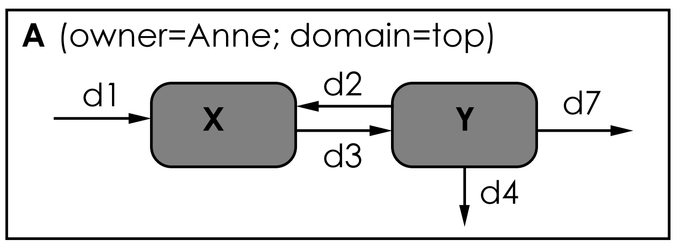
Anne now does some more work on the parent. In the process of delegating decomposition of other processes, she realises one of the outputs of process Y is missing, and so she adds it (d7). Meanwhile, Bob has also noticed the omission, and adds it, using his own label (d6). He also adds another missing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





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



5. ViewPoint A contains a non-primitive process for which no corresponding decomposition ViewPoint exists.

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

2.1. 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.

3. 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 inconsistencies (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.

We have also de-coupled the actions of changing a label in one ViewPoint, and propagating that change to other 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 that affect different ViewPoints allows those 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.

4. 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.

Nuseibeh *et. al.* (1993) introduces our notation for expressing these relationships between ViewPoints. The notation allows the method designer to express relationships that should hold between elements of any two ViewPoints, no matter what representation scheme each ViewPoint uses. Global consistency is expressed in terms of pairwise consistency relationships between ViewPoints.

5. When should relationships between ViewPoints be checked?

Development of an individual ViewPoint can proceed unrestrained by relationships with other ViewPoints. The ViewPoint owner will invoke a consistency rule if she needs to check whether 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. The checking is always performed from the context of one of the ViewPoints: there is no central control. We call the ViewPoint that invokes the rule the *source* ViewPoint, and the other the *destination* ViewPoint.

Guidance about when particular rules should be applied comes from the local process model in the source ViewPoint. This defines conditions under which the rule can be invoked, and the possible outcomes from invoking it. Possible outcomes for a consistency rule include that the relationship it expresses holds, or that the rule failed for one of a number of reasons.

A ViewPoint's process model is expressed in terms of preconditions and post-conditions for various actions. All development actions which may be performed on the ViewPoint are included in the process model, but here we are most concerned with the actions that apply consistency rules and which repair inconsistencies. The preconditions refer to the current state of the ViewPoint, and current relationships with other ViewPoints. These are used to restrict the stages of development in which an action can be performed.

6. How are relationships between ViewPoints checked?

When a consistency rule is applied, both ViewPoints must co-operate to perform the check, and both ViewPoints need to know the result. 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.

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 does not know whether B did something that fixed the inconsistency just after the rule was applied. In fact all A knows is that there was an inconsistency *at the time* when the rule was applied. Hence, we may need to ensure that several inter-ViewPoint actions are carried out as a single transaction, to guarantee that the relevant preconditions hold.

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.

7. 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 owner of the source ViewPoint wishes to correct this. In many cases, resolution will not be necessary after the failure of a rule, because the inconsistency can be tolerated.

7.1. Conflict and Inconsistency

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, nor does it imply that the conflict was 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.

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[1], or the result of a mistake.

7.2. Inconsistency Resolution

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.

For example, consider the inconsistencies in our scenario. Inconsistencies (1) and (5) both involve the same rule: no ViewPoint exists to represent a non-primitive process. In (1) the appropriate resolution is to create the ViewPoint. In (5) the appropriate resolution is either to delete the process or to mark it as primitive. 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 reasoning with information in the work record cannot always correctly diagnose an appropriate resolution action. 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. As another example, consider the propagation of changes from one ViewPoint to another: if labels in two different ViewPoints were once the same, it does not always imply that they should remain the same. A ViewPoint owner might change the name of a label, then later discover she has accidentally chosen a label used in another ViewPoint, and so change 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.

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.

Our approach to supporting the resolution process is through the provision of a set of pre-defined set of 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 consistency relationships. Possible resolution actions are associated with each consistency rule in the process model: 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

---

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

information about the history of the ViewPoints in question. The ViewPoint owner may, of course, choose not to apply any of the suggested actions.

The method designer may need assistance in identifying appropriate actions, and linking them to consistency rules. The task of defining actions is open-ended, and effort invested here will reduce the load on the method users. Consistency rules and the resolution actions associated with them will be generated from four main sources:

- the rationale and operation of the method (i.e. rules and actions that ensure that the method is being used correctly);
- examples and case studies of the use of the method will help to refine the actions and the conditions under which they apply;
- experiences of method users – Software designers already handle routine conflicts, hence their expertise may be gradually encoded into the guidance offered by the tool;
- general purpose conflict exploration tools – e.g. Computer Supported Negotiation (Easterbrook, 1991) allows ViewPoint owners to identify correspondences between their disparate descriptions, and evaluate a number of potential resolutions.

8. 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 concern a part of the ViewPoint that is only tentative; it may be the result of a known conflict which the owners are not yet ready to resolve; or some anticipated future action will resolve it anyway. Also, 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 is 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, so that 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.

9. 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 tool 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 using the tool, 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.

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).

References

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.

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, 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, 187-200). IEEE Computer Society Press.