

# Structured graphs

a visual formalism for scalable graph based tools  
& its application to software structured analysis

Submitted by Mark J. Sifer to meet the requirements of the Doctor of Philosophy in the  
School of Computing Sciences at the University of Technology, Sydney in 1996.

# CERTIFICATE

I certify that this thesis has not already been submitted for any degree and is not being submitted as part candidature for any other degree.

I also certify that the thesis has been written by me and that any help that I have received in preparing this thesis, and all sources used, have been acknowledged in this thesis.

**Signature of Candidate**

Production Note:

Signature removed prior to publication.

## Acknowledgement

I am indebted to my principal supervisor John Potter for both his guidance and collaboration in this work. I also wish to thank my co-supervisor John Leaney for his support. Some of the concepts in the second half of this thesis had their genesis while I was a research student at Hiroshima University, under the supervision of Tadao Ichikawa.

This work has also benefited from discussions and collaborations with Peter Eades, Masahito Hirakawa, David Lowe, and former colleagues in the Computer Systems Engineering Division of Computer Sciences Corporation (Australia). I also wish to thank the thesis examiners for their detailed comments. Addressing these, has significantly improved the clarity and presentation of the thesis. The work was funded by an Australian government postgraduate award with a top-up from the School of Computing Sciences.

## Abstract

Very large graphs are difficult for a person to browse and edit on a computer screen. This thesis introduces a visual formalism, *structured graphs*, which supports the scalable browsing and editing of very large graphs. This approach is relevant to a given application when it incorporates a large graph which is composed of named nodes and links, and abstraction hierarchies which can be defined on these nodes and links.

A typical browsing operation is the selection of an arbitrary group of nodes and the display of the network of nodes and links for these nodes. Typical editing operations is: adding a new link between two nodes, adding a new node into the node hierarchy, and moving sub-graphs to a new position in the node hierarchy. These operations are scalable when the number of user steps involved remains constant regardless of how large the graph is. This thesis shows that with structured graphs, these operations typically take one user step.

We demonstrate the utility of structured graph formalism in an application setting. Computer aided software engineering tools, and in particular, structured analysis tools, are the chosen application area for this thesis, as they are graph based, and existing tools, though adequate for medium size systems, lack scalability.

In this thesis examples of an improved design for a structured analysis tool, based on structured graphs, is given. These improvements include scalable browsing and editing operations to support an individual software analyst, and component composition operations to support the construction of large models by a group of software analysts.

Finally, we include proofs of key properties and descriptions of two text based implementations.

# Contents

Certificate	ii
Acknowledgement	iii
Abstract	iv
<b>1 Introduction and Review</b>	<b>1</b>
1.1 Limitations of current Graph Models and Visual Formalisms	1
1.1.1 Interacting with large graphs on a computer	2
1.1.2 Graph models	2
1.1.3 Visual formalisms	4
1.1.4 The need for a new visual formalism	5
1.2 Limitations of current graph based tools	5
1.2.1 Hypertext tools	6
1.2.2 CASE tools	6
1.2.3 Structured analysis tools	12
1.3 Levelled Data Flow Diagrams as the basis of a new formalism	14
1.4 Structured graphs	15
1.5 Discussion	16
1.6 Thesis overview	16

## Part I Structured Graphs

<b>2 Viewing a Structured Analysis Model</b>	<b>18</b>
2.1 The model	18
2.2 Browsing and editing the model	20
2.3 Discussion	27
<b>3 Browsing and Editing Structured Graphs</b>	<b>28</b>
3.1 Network Abstraction	29
3.2 Browsing Models	35
3.3 Incomplete Models	38
3.4 Editing Models	40
3.5 Model Limitations	41
3.6 Summary	45
<b>4 Ordered Sets : Background</b>	<b>46</b>
4.1 Standard Terminology	46
4.2 View Orders	49
<b>5 Structured Graph Formalism</b>	<b>53</b>
5.1 Structured Graphs	54
5.2 Compact and Abstract Models	54

5.3 Properties	57
5.4 Model Editing	58
5.5 Model Viewing	58
5.6 Limitations	59
5.7 Discussion	61

## Part II Structured Graph Components

<b>6 Building with Structured Analysis Components</b>	<b>62</b>
6.1 Example Components	62
6.2 The composed model	67
6.3 Data flow meshing	69
6.4 Viewing a component	71
6.5 Discussion	72
<b>7 Typed Link Orders</b>	<b>73</b>
7.1 Tree typed link orders	74
7.2 General typed link orders	75
7.3 Link schema constraints	77
7.4 Summary	81
<b>8 Component Composition</b>	<b>82</b>
8.1 Limitations of model composition	83
8.2 Components	84
8.3 A component composition example	85
8.4 Component merge	86
8.5 Component composition	87
8.6 A component limitation	91
<b>9 Typed Order Formalism</b>	<b>93</b>
9.1 Labelled orders	93
9.2 Typed orders	96
9.3 Typed tree schema	97
9.4 Typed order schema	97
<b>10 Structured Graph Component Formalism</b>	<b>101</b>
10.1 Pre-components	101
10.2 The merge operator and components	102
10.3 The component mesh operator	105
10.4 The component composition operator and proper components	109

<b>11 Conclusion</b>	110
11.1 The contribution of part one: structured graphs	110
11.2 Limitations of structured graphs	111
11.3 Future work for part one	112
11.4 The contribution of part two: structured graph components	112
11.5 Limitations of structured graph components	113
11.6 Future work for part two	113
11.7 A component schema example	113
11.8 Final Discussion	115
<b>Bibliography</b>	116
<b>Appendices</b>	123
A Glossary	123
B Structured graph proofs	125
C Gofer implementation of structured graphs	131
D A console based structured graph tool	160

---

---

## Introduction and Review

---

The aim of this thesis is to develop a foundation for the design of software tools which can support scalable browsing and editing of graph representations. This foundation will be demonstrated through its application to structured analysis. The contributions of the thesis are summarised in chapter 11, the conclusion.

Many *software tools use graphs* to provide visual representations of relationships, taking advantage of the display and processing capabilities of today's personal computers and workstations. A small sample of relevant applications includes: project planning, network management, traffic control, software analysis and design, hypermedia, and more recently world wide web site design.

Software tools for these applications should facilitate user interaction with graph based representations. This interaction typically includes browsing and editing of the graphs. Ideally the user process for browsing and editing would involve the direct manipulation of the displayed graph.

A lack of *scalability* is a problem common to many of these software tools. As the graph representation becomes very large, browsing and editing becomes increasingly difficult. Browsing and editing suffer from loss of perspective and an increasing number of required user steps: this is what we refer to as a lack of scalability. Scalability is, for us, a user perspective concern, and has nothing to do with underlying computational complexity. A tool is scalable when it can be used to browse and edit both small and large structures equally well.

Section 1.1 discusses a range of graph models and visual formalisms, focusing on their limitations. The major limitation is the difficulty of supporting the flexible browsing and editing of large graphs. A new visual formalism is proposed to address this need. Section 1.2 presents the lack of scalability limitations discussed earlier in a variety of application contexts, reinforcing the need for a new formalism. Levelled data flow diagrams (DFDs), which are used in structured analysis, are then proposed as the starting point for the new formalism in section 1.3. Section 1.4 introduces this formalism, structured graphs, with some discussion following in section 1.5. The final section gives an overview of the rest of the thesis.

### 1.1 Limitations of current Graph Models and Visual Formalisms

This section identifies a general limitation of current graph based tools (section 1.1.1),



then establishes a foundation for improving the design of these tools. Graph models (section 1.1.2) then visual formalisms (section 1.1.3) are considered for this foundation.

### **1.1.1 Interacting with large graphs on a computer**

Designing software tools which allow a person to interact with a large relationship (between two types of entities), which is presented visually on a computer screen as a network of labelled nodes and links, is difficult. The interaction typically takes two forms: reading relationships between nodes and sets of nodes, and modifying the network. Clearly, modifications would be preceded by reading. For large and perhaps complex relationships a single network is often incomprehensible even when presented visually. At worst, browsing would require looking at the whole network, possibly with a zoom and pan facility, and editing would require direct manipulation of individual nodes and links.

This problem can be attacked in many ways. Two categories are: manipulation of the presentation of the network, and the addition of new information to somehow structure the network. An example of the first is geometric manipulation of the network to highlight a portion of the network, as in fish eye views (Sarkar and Brown, 1994). In the second, the network is structured in a way so that a person need only focus on a meaningful portion of the network; this implies an ability to select this portion. This can be done by adding aggregation hierarchies, usually just on the nodes, but sometimes on both the nodes and links.

When a network is structured, as in the second category, the effort to read the network may have decreased, as a person can see relationships between summary (aggregate) nodes. However the effort to modify the model in some situations may have increased, due to the overhead of maintaining the additional structuring information in a consistent state, and it is not obvious what the additional structuring information should be.

The ideas of problem analysis can help identify what the additional structuring information might be. Yeh and Zave (1980) isolated: partitioning, abstraction and projection as the structuring principles used in problem analysis. Partitioning refers to aggregation relationships, abstraction to generalisation, and projection to the use of multiple viewpoints/perspectives.

### **1.1.2 Graph models**

The simplest graph model is the visual depiction of a binary relation. Nodes represent the domain (and range) elements, and arcs between two nodes represent the node pairs that are members of the relation. A comprehensive treatment of graphs is given by Berge (1962) and Carre (1979). When we restrict the relation to being symmetric, we have an undirected graph with edges between nodes.

There are many extensions of graphs, some are: p-graphs, multigraphs, bipartite graphs, hypergraphs, nested graphs, hypernodes, compound graphs and higraphs. P-graphs allow multiple arcs between nodes, multigraphs extend simple graphs by allowing multiple edges between nodes, and in a bipartite graph the nodes are partitioned into two sets, where nodes from the same partition are never directly connected (Carre 1979).

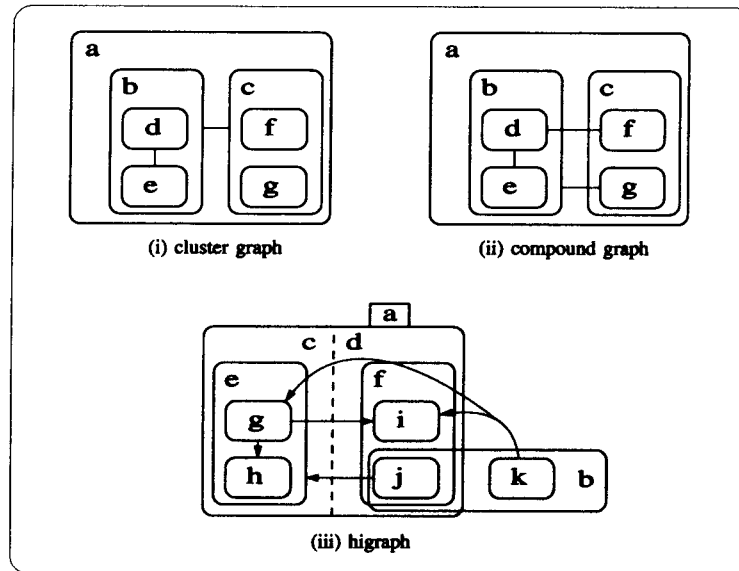


Figure 1.1 Example graphs

In hypergraphs an edge between a pair of nodes is replaced by a hyperedge between sets of nodes (Berge 1973). In compound graphs a node may represent a sub-graph, and edges are allowed to cross abstract node boundaries to connect with a contained node (Sugiyama 1991). Clustered graphs are restricted compound graphs where edges may not cross abstract node boundaries (Feng 1991). Clustered graphs have been called nested graphs in other work.

Hypernodes are nested graphs where nodes are augmented with type information so that the nesting may be recursive (Poulovassilis and Levene, 1994), (Levene and Poulovassilis, 1990). The hypernode model has also been proposed as an underlying formalism for hypertext (Levene and Loizou, 1995). Higraphs are the result of extending Euler/Venn diagrams to represent Cartesian product integrated with hypergraphs (Harel 1988). Examples of a cluster graph, compound graph and higraph are shown in figure 1.1.

Though not quite a visual formalism, GraphLog is a query language which merges logic based queries with graph based visualisation (Consens, Mendelzon, Ryman, 1991). Queries are posed as graphs. They regard a query “as defining a template for matching subgraphs of a graph”. They also use a variation of Higraphs for display. Hypernodes also have an associated logic based query language.

### 1.1.3 Visual formalisms

A visual formalism is a diagrammatic notation with a formal syntax and formally defined browsing and editing operations. Generally a visual formalism is concerned with diagram topology rather than its geometric layout. David Harel introduced the term visual formalism when presenting Higraphs: the formalism behind statecharts. The article, “On Visual Formalisms” (Harel 1988) provided motivation for visual formalisms:

The intricate nature of a variety of . . . systems and situations can, and in our opinion should, be represented by *visual formalisms*: visual because they are to be generated, comprehended, and communicated by humans; and formal, because they are to be manipulated, maintained, and analyzed by computers . . . We believe that in the next few years many more of our daily technical and scientific chores will be carried out visually . . . The languages and approaches we shall be using in doing so will not be merely iconic in nature (e.g. using the picture of a trash can to denote garbage collection), but inherently diagrammatic in a conceptual way . . . [emphasis in original].<sup>1</sup>

Harel also made the distinction between diagram topology and geometry, with visual formalisms being concerned only with the former. That is, they are concerned with which diagram elements are connected or contained rather than how they are laid out. In the description of the Higraph visual formalism his mathematical model describes only what the allowable structures are. No browsing or editing operations are specified.

Johnson et al. (1993) developed the idea of a visual formalism further. They were interested in using visual formalisms for frameworks which facilitate the construction of applications. They provide the following definition:

Visual Formalisms are diagrammatic displays with well-defined semantics for expressing relations.

They give tables, graphs, plots, maps and others as examples. However, when it comes to an implementation of a visual formalism they include operations:

The purpose of implementing Visual Formalisms as application frameworks is . . . to provide strong representational, editing, and browsing capabilities . . . An implementation of a Visual Formalism should provide editing and browsing capabilities that make sense for the formalism.

Such operations should be intrinsic to the formalism itself, rather than something to be added at the implementation stage, because editing operations should be structure preserving. Formal definition of the operations ensure structure preservation. A visual formalism could then be considered a visual abstract data type: a mathematical definition of some data structure, associated structure preserving operations (including browsing

---

<sup>1</sup>This quote is repeated from Johnson et al. (1993)

and editing), and with a corresponding visual representation.

This section has motivated the use of a visual formalism for the problem this thesis is addressing, in terms of it being better to solve a more generic problem. There are other reasons for creating a visual formalism when one's goal is an improved tool design. A formalism provides a clear specification from which tool designers can work. The proofs associated with the formalism give the designers greater confidence in the specification. Formal properties and their associated proofs identify what some limitations of the tool will be. By abstracting away from specific applications, the mathematical modelling is easier, as there are fewer entities to deal with; for example, we deal with nodes rather than processes, stores and process activation tables.

#### **1.1.4 The need for a new visual formalism**

This thesis aims to provide a foundation for the design of improved graph based tools. Section 1.1.1 has asserted that an area in which improvement is required is scalability. Section 1.2 will establish this assertion by identifying the lack of scalability in a number of application settings.

Existing graph models and visual formalisms have been examined. The graph models describe a representation but not editing operations so they can not be a complete solution. Cluster graphs, compound graphs or hi-graphs which incorporate summary graphs could however be starting points for a representation.

The foundation needs to be a visual formalism, to include browsing and editing operations with the diagrammatic representation. However existing visual formalisms do not meet the scalability requirements. This thesis presents a new visual formalism, as the foundation for meeting these requirements.

### **1.2 Limitations of current graph based tools**

To get a clearer idea of the need for improved scalability and how it should be addressed, a range of application areas needs to be examined. In this section hypertext, a range of CASE tools, and structured analysis tools will be examined.

In the following sub-sections, a common need for scalable editing and browsing will be shown. We will also see there have been many attempts to formalise the data models of these applications, where a part of this model is describing a graph representation. So providing a foundation for designing new graph based tools may also assist with creating formal descriptions of, reasoning with and classifying existing tools and the graph representations they manipulate.

### 1.2.1 Hypertext tools

Tools for hypertext (Conklin 1987) and more recently hypermedia are graph based tools, with similar problems to graph based CASE tools. Such tools are concerned with allowing the online presentation of large amounts of loosely structured information (Nielsen 1990), whilst CASE tools are concerned with allowing both the online presentation and editing of large amounts of information. In the latter, the information is relatively cohesive. However, both are graph based, concerned with online visual presentation and with scale.

Feiner (1988) described a hypertext tool. To overcome the tendency for users to become disoriented as they browse, some hypertext tools allow the display of the whole hypertext graph. He introduces the general difficulties of the online presentation of large graphs: when arcs cross, as occurs in most large graphs, visual comprehension becomes difficult. Selective display and distortion of a graph using techniques such as fish eye views (Sarkar and Brown, 1994) allow a viewer to focus on a portion of the graph, but at the cost of making it difficult for a viewer to relate multiple view instances together. He proposes combining a strict tree hierarchy with an arbitrary directed graph structure. The final model resembles a compound graph, in which portions are visually displayed using a nested topology similar to that used in higraphs. The rule used for showing links between summary nodes is:

When preparing to display a link between two nodes, IGD [the tool] traces the chapter ancestries [the non-leaf nodes] of both nodes to find their first common ancestor, which in the limit is the document itself. An arc is drawn between the two children of the common ancestor that are on the path to the two original nodes.

This rule is shown in full here because of its similarity to a simple structured graph as presented in this thesis. The resultant tool IGD (Interactive Graphical Documents) uses the hierarchy to allow a variety of information hiding and decluttering techniques to reduce the amount of detail displayed.

### 1.2.2 CASE tools

This subsection is concerned with work on graph based CASE tools which focuses on the issue of scale: how to cope with very large models. In particular, we are interested in tools which facilitate the online interaction with models using graph based visualisations. First we consider the range of CASE tools.

Software engineering encompasses a wide range of activities which include: planning, analysis/design, construction, testing, maintenance, project management, process/quality assurance, and configuration management. CASE clearly has a role to play in all these activities and Forte and McCulley (1991) have surveyed them. Each of

these areas alone may use a range of tools, for example construction could include (Fuggetta, 1993): assemblers, compilers, cross-assemblers, cross-compilers, debuggers, interpreters, linkage editors, precompilers/ preprocessors, code generators and code restructurers. Some but clearly not all of these tools could be graph based, in the sense that they allow a user to interact with information via graphs. Areas which make significant use of graph based tools include: program visualisation/reverse engineering, project planning and analysis/design.

A substantial portion of software engineering is system maintenance. To assist the maintenance of large systems where the documentation is either missing or out of date reverse engineering tools have emerged. An important subclass of these, are tools for design recovery, which transform code into analysis/design notations. RE-Analyser (O'Hare and Troan, 1994) is a system which transforms code into structured modelling abstractions: DFDs, ERDs and finite state machines. RECAST (Edwards and Munro, 1993) takes COBOL code and derives a no-loss representation of the system in a structured systems analysis and design method format. The IAASys (Canfora et al., 1992) tool converts ADA into dynamic DFDs and also supports animation.

Khan and Miyamoto (1993) and Khan (1994) have also produced HVIEW (Hierarchical View) a program visualisation system. Unlike the previous two systems they do not generate a traditional analysis or design model, for example DFDs and structure charts. Rather, they produce a model with which a person can interact. They note:

Various graphical techniques have been used informally to facilitate program representation and analysis both in forward and reverse engineering, such as *flow graphs* ... A common problem with most of these ... is that their apparent size and complexity grow intractably from the human perception point of view with the size and complexity of the software system.

They then give as their requirement:

In reverse engineering, where the purpose is to produce effective process perception in a human expert ... The size and complexity of the visual information can seriously effect the quality and response efficiency of human understanding... Program visualization therefore, in addition to multi perspective support, requires decomposability of the representation formalism and the reducibility of the information volume in the visualization schema.

Their system represents code as a network. They also introduce two abstraction hierarchies: for data and for functions. Browsing of the model is done by selecting the desired data and function abstraction levels; by taking cross-sections of the hierarchies. The results of the browsing is a network at the selected abstraction levels. Their rule for deriving the relationship between abstract data items and abstract function is domain specific. For example when an abstract function contain a part which "reads" a data

item, and another which “writes” the data item, the derived relationship between the abstract function and the data item could be “reads then writes”. The significance of this work for us is the use of two orthogonal hierarchies to facilitate flexible abstraction on an underlying network.

MultiView (Read and Marlin, 1996) is another project concerned with software visualisation and maintenance. MultiView tools provide multiple views of software which can be independently edited while maintaining overall consistency. These views include: a text view of code, a tree view and a flow graph view. An interesting feature of this work is their approach to the data model behind the various views. They state:

One approach to maintaining multiple viewpoints in an environment (such as a programming environment) is to store the object of interest in terms of some suitable canonical representation and then derive various viewpoints from this canonical representation. This is the approach we have adopted in our MultiView programming environment.

This approach makes maintaining consistency easier, and simplifies the specification of editing operations.

Other work on software visualisation which is concerned with scale includes, Kimelman (1995) and Storey and Muller (1995). The former paper is concerned with the visualisation of a dynamic system, that is an executing program.

The paper proposes a combination of “abstraction” or “reduction” techniques for reducing the visual complexity of a graph, while preserving or even enhancing the significant information that it was meant to convey. A number of means are provided for automatically selecting nodes and edges ... Operations are then provided for “disposing” of these selected nodes: “ghosting” (relegating nodes to the background visually), “hiding” (removing nodes from the display entirely), and “grouping” (grouping nodes under a single meta-node).

Storey and Muller take a different approach to the visualisation of software structures. They combine the geometric distortion of a fish eye lense with nested graphs. Their motivation was:

A basic incentive for writing this tool (their Simple Hierarchical Multiple Perspective views, SHriMP tool) is to provide a mechanism for visualizing detail of a large information space and at the same time provide contextual cues concerning its context.

Better contextual cues are provided for the visualizer as they navigate the hierarchy implicit in the nested graph because:

All steps in the path travelled are visible, in the form of nested nodes. A user can elect to return to any subsystem in the branch travelled, and elide the information contained in that system.

An emerging area which utilises graph based models is software architecture (Dean and

Cordy, 1995). The example architectural language they use is based on typed directed multigraphs; which provides typed nodes and edges and permits more than one edge of a given type between nodes, extended by allowing edges with arbitrary arity. If these diagrams become large, then the issues of scalable browsing and editing may emerge.

A current concern though of software architecture is building systems through the composition of components. Shaw et al. (1995) describe a system as:

Systems are composed from identifiable components and connectors of various distinct types. The components interact in identifiable, distinct ways. ... Components may be either primitive or composite. ... Similarly, connectors may be primitive or composite.

A motivating factor in the design of an architecture is given as:

To build truly composable systems we must allow flexible, high-level connections between existing systems in ways not foreseen by their original developers. ... Components and connectors must be reusable in different settings, ...

The concern with flexible composition of components appears to be a key concern here. Shaw et al. (1996) contrast current approaches with the inflexibility of early module interconnection languages.

[ they ] require considerable prior agreement between the developers of different modules. For example, they assume that simple name matching can be used to infer inter-module interaction, ...

Flexible composition could also be desirable when constructing other graph based models. Very large graph based models require a team effort. This effort would proceed more easily, if each team member could build their own components, which can be flexibly composed to form the final model.

Project planning is another area of software engineering that makes significant use of graph based tools. Traditional project planning use a precedence network to show the dependencies between tasks. The alternative representation is PERT charts which show dependencies between events. For any significant project these networks are very large. Interacting with this network was traditionally done by looking at a large wall chart produced by a plotter. A work breakdown structure (WBS) was used to describe the hierarchy of tasks.

Structured planning (Sifer, 1988), (Wilson and Sifer, 1988), (Wilson and Sifer, 1990), (Potter and Sifer, 1988) is the application of functional decomposition to project planning. A project task with its inputs and output deliverables can be successively decomposed. This results in a set of levelled work flow diagrams, which include the WBS and a deliverable dictionary. A tool was created to allow the flexible browsing and editing of a structured planning model. Like HVIEW this flexibility was achieved



by allowing a person to select the desired task level and deliverable level of abstraction and then show the precedence network at this level. However, the structured planning tool also allowed the direct editing of any abstract network. The limitations of this work were: the editing operations were not formally defined, the implicit formalism was expressed in terms of tasks and deliverables only, the hierarchies were limited to trees and only top-down editing was supported. Further work on structured planning was done by Cimitile and Visaggio (1994). They have extended structured planning by providing a dynamic interpretation of structured planning networks using Petri nets and evaluated its use.

There has been an evolution in analysis and design tools. A change from supporting structured methods to object oriented tools mirrors the change in popular methodologies. A list of earlier CASE tools is given in (Davis, 1990). For real time analysis, two major tools in commercial use are: Software Through Pictures and Teamwork<sup>®</sup>. These supported the construction of both DFD and ERD models. These two tools have since been evolved into object oriented analysis tools, which still support a DFD model (or functional model) as a part of the OOA model. However, the browsing and editing capabilities of these models have not changed significantly. The next work discussed does address this.

Teorey et al. (1989) introduced Entity Relationship clustering to aid user browsing of large ERDs. Their motivation was:

When the scale of a database or information structure is large and includes a large number of interconnections among its different components, it may be very difficult to understand the semantics of such structure and to manage it, especially for the end users or managers.

They then justify their introduction of clustering with:

The clustering concept is ... important because it provides a method to organise a conceptual database schema into layers of abstraction, and it supports the different views of a variety of users.

Clustering is on the entities. This clustering may be recursive, effectively inducing an abstract entity hierarchy on the underlying ERD. They define a number of heuristics for clustering, including an abstraction grouping, which uses generalisation, aggregation, classification and membership to form an entity cluster. Their clustering process has been defined in a bottom-up manner.

Chen and Chung (1991) argue that a major limitation of existing DFD tools is the difficulty of restructuring such models. "Using basic editing operations to restructure large systems with voluminous data-flow diagrams is tedious, laborious and error prone". They define a set of restructuring operations and prove they are consistent. Consistent means the operations do not change the underlying network of leaf

processes and data flows. The major limitation of this work is a lack of support for composite data flows. Their formal treatment has processes, stores, terminators and flows. This results in a more complex model than if they had abstracted to just nodes and links for their core model. Also, like other formal treatments described in this section the process hierarchy is limited to a tree.

Guindon (1992) discusses at length the requirements for the graphical interface of a software design assistant. This is based on her previous empirical studies on the early stages of software design (Guindon et al., 1987), (Guindon, 1990a), (Guindon, 1990b). She says:

The most influential finding is that the early stages of design are opportunistic and do not follow a top-down dynamic, and moreover, this is *good* design practice.

She states the behavioral characteristics of opportunistic design include:

- (1) interleaving the development of partial solutions at various levels of abstraction and in different subsystems,
- (2) inference of new requirements and design constraints throughout the solution development, often leading to *drastic restructuring of the design solution*, [My italics]
- (3) extensive mental simulations of scenarios in the task domain triggering the discovery of new requirements in widely different levels of abstraction and subsystems.

Having established the requirements, the key features of her visualization tool are given:

- (1) The display of any software modules at arbitrary levels of abstraction ...
- (2) The simultaneous display of software modules from different subsystems,
- (3) The unrestricted, smooth navigation between these displayed software modules ...

Her displays show nested code stubs, which may be a module down to a code block, with their inputs and outputs, shown as horizontal line entering or leaving the blocks. Blocks may also be nested in other blocks. Feature (1) is achieved by replacing a block with its child blocks perhaps recursively; the layout is done automatically. Feature (2) is simply achieved by displaying modules in separate windows and allowing multiple windows to be displayed. Feature (3) is achieved by allowing all displayed objects to be mousable; an object can be selected with the mouse, a user can expand it, go to an inputting or outputting object, or edit it.

### 1.2.3 Structured analysis tools

In this subsection the scalability problem this thesis is addressing is introduced by considering the use of a structured analysis tool on a large project. A short history of structured analysis is then given, followed by a review of formal treatments of

structured analysis.

A major problem with structured analysis tools (such as Cadre, 1990) is that large models containing a few hundred processes develop editing inertia, that is, editing becomes increasingly difficult as the model grows in size. For sub-models containing up to a hundred processes many structural variations can be tried. However, few variations would be tried for the overall model structure because the manual effort to revise all the affected data flow diagrams (DFDs) is too great. For example, in my experience it can take three weeks of manual work to completely restructure a model containing five hundred processes with thirteen levels, to one with seven levels, whilst preserving all data flow dependencies.

In summary, as a model under consideration grows larger, restructuring operations on the whole model become significantly more time consuming. This is because each affected diagram has to be manually edited to maintain a balanced model; a model is *balanced* when each DFD's external inputs and outputs matches the DFD's parent process inputs and outputs. For example, adding a data flow could require a change to the DFD in which the producer process appears, a change to the DFD in which the consumer appears, and changes to all higher level DFDs in which the data flow is present, to maintain a balanced model. But from the tool user's perspective the change is the addition of a single data flow, which can be specified by nominating the producer and consumer processes. The user is required to make a series of DFD changes to maintain model balance where the number of such changes increases with the size of the model. If the intention is to move a sub-model to a new position in the overall model, rather than just adding a data flow, the situation is far worse. This need to manually adjust all affected DFDs, thus results in a lack of scalable editing.

Structured analysis was introduced by the works of Ross and Brackett (1976), Ross and Schoman (1977), Ross (1977). These early papers authored by Ross introduced what would become the structured analysis and design technique, commonly known as SADT. Though SADT does not use DFDs, its diagrams have the same levelled structure, generated by performing a structured decomposition. In Ross's 1977 paper titled *Structured Analysis: A Language for Communicating Ideas*, his aim seems more general than just providing a technique and notation for software requirements definition. He states:

... the language of structured analysis (SA), a new way of putting together old ideas, provides the evolutionary natural language appropriate to the needs of the computer field.

He states further:

The only function of SA is to bind up, structure, and communicate units of thought expressed in any other chosen language. Synthesis is composition, analysis is decomposition. SA is structured

decomposition, to enable structured synthesis to achieve a given end.

This implies that the lowest level boxes and edges on his diagrams are place markers for other content. The approach of this thesis to treat the SA notation of DFDs in a generic fashion is consistent with Ross's original ideas.

DeMarco (1979), Gane and Sarson (1979) and Yourdon and Constantine (1979), described the methodology of structured analysis which was to be captured on paper by the notation of data flow diagrams, a data flow dictionary and process specifications.

Structured analysis and its notation has been extended to support real time systems by Ward (1986), Goma (1986), Hatley and Pirbai (1987), Hashimoto (1987), Peters (1988) and Shoval (1988). Formal treatments of DFDs extended to support real time include: (Richter and Maffeo, 1993) using Petri nets, and (Beek 1993) using timed statecharts.

The next evolution of structured analysis was as part of object oriented analysis (OOA). The object modelling (OMT) technique of Rumbaugh et al. (1991) had three components: a object model, a dynamic model and a functional model. The function model used a variation of data flow diagram notation. So structured analysis which describes the functional decomposition of a system had been incorporated in OMT as the functional model. A discussion of structured analysis and object oriented analysis was given in the conference panel lead by Champeaux (1990). There is also work which builds on both OMT and the logical relational design methodology (LRDM) (Teorey 1986) to create a methodology which provides a closer integration of a entity-relationship models with a DFD model (Kuo 1994). Earlier work with a similar aim is (Ward 1989), (Lee 1990), (Fuggetta et al., 1993).

Since the late eighties several formal treatments of structured analysis models have been published. This work was partly motivated by a desire to improve the structured analysis based CASE tools of the time. Adler (1988) provided an algebra which describes process decomposition. It describes not only what the allowable data flow diagram models are, but also the decomposition transformations, which are top-down. The data flow hierarchy is not included. He also describes a set of quality measures for good decomposition. Arndt and Guercio (1992) asserted that Adler's algebra does "not correspond to the intuitive notion of good decomposition" and "leads to an inefficient decomposition process". They provide an alternative which addresses these issues.

Tao and Kung (1991) in their paper "provide a formal basis for the DFD on which consistency in process decomposition and completeness of a DFD specification can be formally checked". Though they do not include the data flow hierarchy in their formal treatment, they state "it is a straightforward extension to allow data flow decomposition" and indicate how this could be done. Olive (1983) used an "is-used-to-

produce” relation in his discussion of information derivability analysis for logical information systems. Tao and Kung (1991) is an extension of both Adler’s and Olive’s work.

Tse and Pong (1989) have produced a formal DFD model. Tse in his (1991) book, has provided a formal models for Yourdon structure charts, DeMarco data flow diagrams and Jackson structure texts. He then identifies the categorical mappings between these three systems. The aim of this work was to demonstrate the commonality of the three methods, to specify refinement operations and to establish measures for the structuredness of a model. Tse et al. (1994) have implemented a tool to transform DFDs into structure charts using Prolog. Boloix et al. (1992) provided a formal treatment of DFDs and structure charts, and established mappings between their formal representations. Butler et al. (1995) has provided a formal model of DFDs which includes a semantics based on a formal process algebra.

### **1.3 Levelled Data Flow Diagrams as the basis of a new formalism**

In this subsection we try to draw out the generic aspects of levelled DFDs. This is done by considering a levelled DFD as an abstraction of a single large DFD. We then highlight the other application areas where a levelled DFD approach has been used. This establishes the fact that a levelled DFD structure, when cast in terms of nodes and links, can abstract large graphs. However, this is only a start. In addition to a visual representation, a visual formalism must also have browsing and editing operations defined for it.

A DFD model comprises a collection of DFD diagrams and a data dictionary. Each DFD diagram also has a nominated parent process. So a DFD diagram includes two relationships: a parent to child process relationship, and data flow connectivity between sibling processes. In totality, a DFD model then consists of data flow to process/store relationships, a process hierarchy and a data flow hierarchy, together with some flow balancing constraints between levels.

The DFD modelling approach can be used to improve the scalability of browsing large graphs. We have already discussed the difficulty of interacting with large networks. A structured analysis model does not directly fit into this mould of a single network. However, if all leaf processes (processes which are not further refined) are joined by matching leaf data flows, a single network is formed. Each DFD is then a summary of a portion of this underlying network; this improves browsing but can make modification of the model less scalable. Again, this will be expanded upon in chapters two and three.

The levelled DFD model has a generic nature. In structured analysis a DFD model is used to represent the function decomposition of a system using processes and data

flows. However, the interesting (for this thesis) aspect of a DFD model, is not the interpretation of an individual DFD but the way a collection of DFDs form a model, and the balancing rules this implies. This levelled collection of diagrams has similarities with a compound graph. However, unlike nested graphs, a DFD model also has a link hierarchy.

A sense of the generality of the topological aspects of a DFD model can be seen from the range of other (not structured analysis) areas to which it has been applied. Other areas include: visual programming (Kodosky 1991), formal specification (Randell 1990), (Fraser et al., 1991), (France 1993), (Liu 1993), and project planning (Wilson and Sifer, 1990), (Cimitile and Visaggio, 1994).

#### **1.4 Structured graphs**

A major contribution of this thesis is the concept of a structured graph. A structured graph is a generalisation of the DFD model, supplemented with scalable browsing and editing operations. Nodes and links replace the processes and data flows of DFDs. The key requirement for structured graphs to be applicable to a given graph based application is that node and link abstraction hierarchies can be defined on an underlying graph of nodes and links.

A structured graph comprises two partially ordered sets of nodes and links, and accompanying producer/consumer relationships. Canonical representations for structured graphs are identified in this thesis, to support network abstraction and design refinement. This provides the basis for formally defining viewing and editing operations on such graphs.

A structured graph can be considered from two perspectives: firstly as the closure (with respect to an operator which adds implied content) of a structured graph, and secondly as the minimal information required to generate the former. An analogy with directed acyclic graphs (DAG) representing partial orders would be: firstly the reflexive and transitive closure of the graph, and secondly the graphs unique basis graph which is the covering relation for a partial order.

In simple cases, from the first perspective a structured graph is a labelled bipartite graph whose vertices are labelled leaf nodes and leaf links, supplemented with node and link ordered sets whose leaf elements include those participating in the graph. Such a bipartite graph is a richer structure than a hypergraph for apart from the labelling, the connections between nodes are directed. In a hypergraph the representation of a hyperedge is as a subset of the node set, which does not contain any arc direction information.

Unlike bipartite graphs and hypergraphs, structured graphs are not just a visual data

representation structure, but are a full visual formalism as they include browsing and editing operations.

## 1.5 Discussion

A common aim of related work has been to facilitate the modelling of large systems with graphs or extensions of graphs, such as hypergraphs and bipartite graphs. However, as several authors have pointed out, these are not adequate by themselves, as a graph becomes very large its previous advantages in assisting human comprehension are lost. A common approach to address this problem has been: allow a user to interact with the graph based model at varying levels of abstraction without getting lost.

The proposed solutions to this problem in various contexts have been to decompose the large graph in some way. One way has been by recursive nesting of graphs as in: nested graphs, hypernodes, Higraphs, structured analysis, ERD clustering and Guindon's design assistant. Another approach has been to add one or more hierarchies to facilitate network abstraction as in: HVIEW, structured planning and IGD. This work has focused on scalable browsing while Chen and Chung's (1991) work, described in section 1.2.2, focused on scalable editing.

The structured graph formalism follows both approaches. In our treatment we generalise the hierarchies supported, from the tree structures used in most of the above, to partially ordered sets. We also provide well defined scalable editing operations both locally within a structured graph, and as the flexible composition of structured graph components, which can be applied to incomplete models in a top-down, bottom-up or arbitrary fashion.

## 1.6 Thesis overview

This thesis is presented in two parts, and each part has three layers. The first part covers basic structured graphs. The second part covers the extension of structured graphs to support the flexible composition of structured graph components. Across both parts, the top layer is the application of the formalism (to structured analysis), the second layer is a generic presentation of the graphical notation and capabilities of the formalism, and the bottom layer presents the actual mathematical definitions and properties of the formalism. Proofs of the properties presented in part one, are given in appendix B, while part two presents properties as unproved conjectures. A summary of introduced terminology is given in appendix A. A text based implementation of part one structured graphs, implemented in Gofer, a functional language, is given in appendix C, and in C++ in appendix D.

In summary, the two major contributions for this thesis are: (i) creating the design for a scalable structured analysis tool and (ii) creating *structured graphs*, a visual formalism

to assist the design of scalable graph based tools in general. The thesis provides a rigorous treatment of structured graphs, of structured graph components and their composition including definitions and properties. Proofs of properties and text based implementations are in appendices.

There are two recommended ways to reading this thesis: firstly just in chapter order, and secondly in layer order. This is shown in figure 1.2. Reading in layer order (reading the part one before part two chapters though) would give a complete overview of the capabilities of structured graphs first. This includes: the scalable browsing and editing in a structured analysis setting, the extended structured analysis notation to support data flow types, and a demonstration of further scalable editing by constructing a model with components.

	Part I	Part II
<b>Layer 1</b> Structured Analysis Example	Chapter 2	Chapter 6
<b>Layer 2</b> Generic Formalism	Chapter 3	Chapters 7,8
<b>Layer 3</b> Mathematical Definitions	Chapters 4,5	Chapters 9,10

Figure 1.2 Thesis organisation

Figure 1.3 is provided as an additional reading aid. It shows the dependencies between chapters.

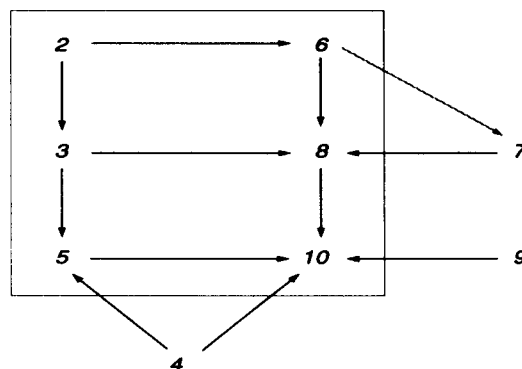


Figure 1.3 Chapter dependencies



---

## Viewing a Structured Analysis Model

---

In this chapter we will browse and edit a structured analysis model, thus demonstrating aspects of a design for an improved structured analysis tool. Recall, in the introduction two major contributions for this thesis were identified: (i) creating the design for a scalable structured analysis tool and (ii) creating *structured graphs*, a visual formalism to assist the design of scalable graph based tools in general. This chapter demonstrates how a hypothetical tool for the former would be used.

To best demonstrate scalable browsing and editing of a structured analysis model we would like to use a large example. Space does not allow this, so a small example is used here. However, another example is presented in Appendix D. First the example model is presented, then we will walk through a couple of updates to the model.

### 2.1 The model

Figure 2.1 shows a partial model which describes an aircraft landing simulator. Four DFDs are given including a context diagram. The data dictionary is also included. The process hierarchy is separated from the collection of levelled DFDs and shown explicitly in figures 2.2 and 2.3 in two different ways. Figure 2.4 shows the data flow hierarchy.

Note in figure 2.1 the `jumbo_display` flow appears in the context diagram while its parent `game_display` appears in the `trainer` DFD. At first glance this seems unbalanced, but `jumbo_display` is part of the `session_display` flow which is also in the context diagram, ensuring the model is balanced.

Neither the process nor data flow hierarchies are trees. The process `timer` shown in figure 2.2 has two parents: `game` and `debrief`. The data flow `time_command` shown in figure 2.4 also has two parents. One of these parents `instructor_debrief_command` does not appear as a flow in the DFDs.

When the process or data flow hierarchy is browsed by the user, it could be presented using the layout of figure 2.3. Each process appears as a horizontal bar, with children under their parents. The total space between children bars is a fixed proportion of the parent's bar length, ensuring the proportions of this diagram are invariant under zooming. As the process `timer` has two parents, it appears twice. It is also shaded to indicate this repeated appearance in figure 2.3.

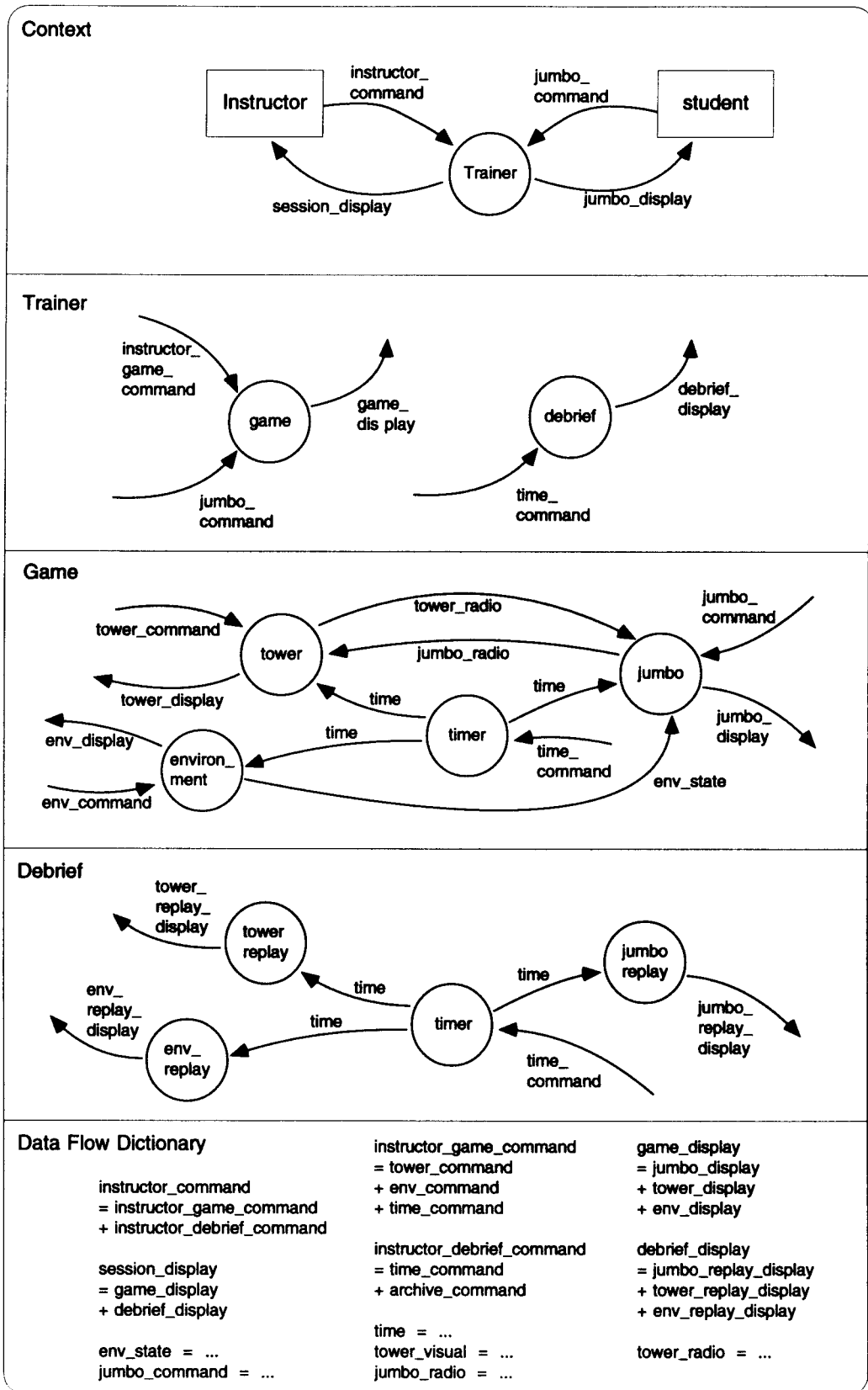


Figure 2.1 Aircraft landing simulator model

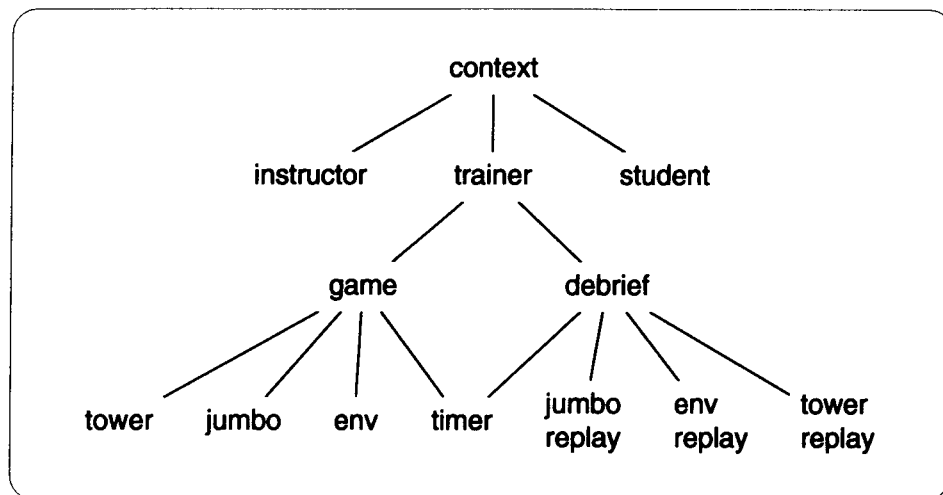


Figure 2.2 Aircraft landing simulator process hierarchy

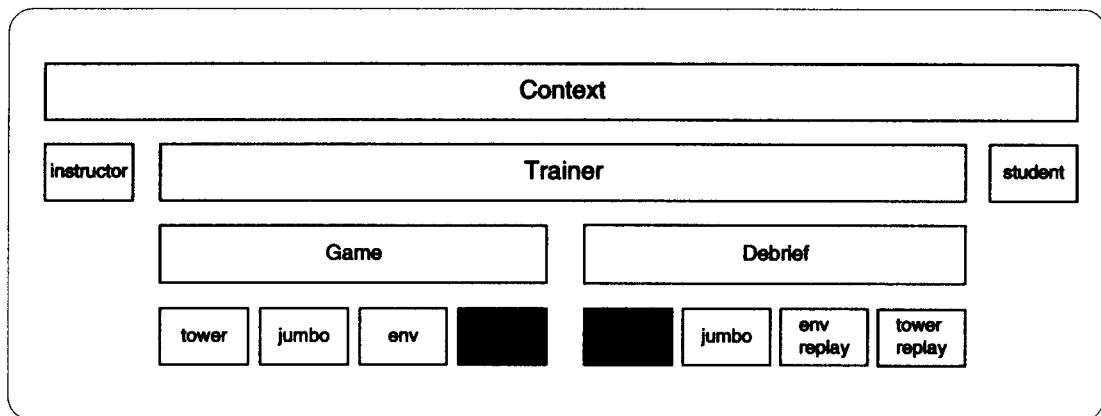


Figure 2.3 An alternative process hierarchy visual interface

## 2.2 Browsing and editing the model

Suppose the user wishes to connect the flow `tower_visual` between `tower` and `student`. Traditionally this is done by selecting each DFD between `tower` and `student`, and adding the `tower_visual` flow to the diagram. With our approach the user brings up the process view hierarchy and selects the processes `tower` and `student`. The selected processes in the hierarchy view are highlighted with a thick box border. The associated *model view*, the graph showing the selected processes and their flows, is automatically generated. This is shown in figure 2.5 where we can see there are no flows between the two processes.

Now the user adds the flow `tower_visual` between the `tower` process and `student` terminator. The current model view is updated and is shown in figure 2.6.

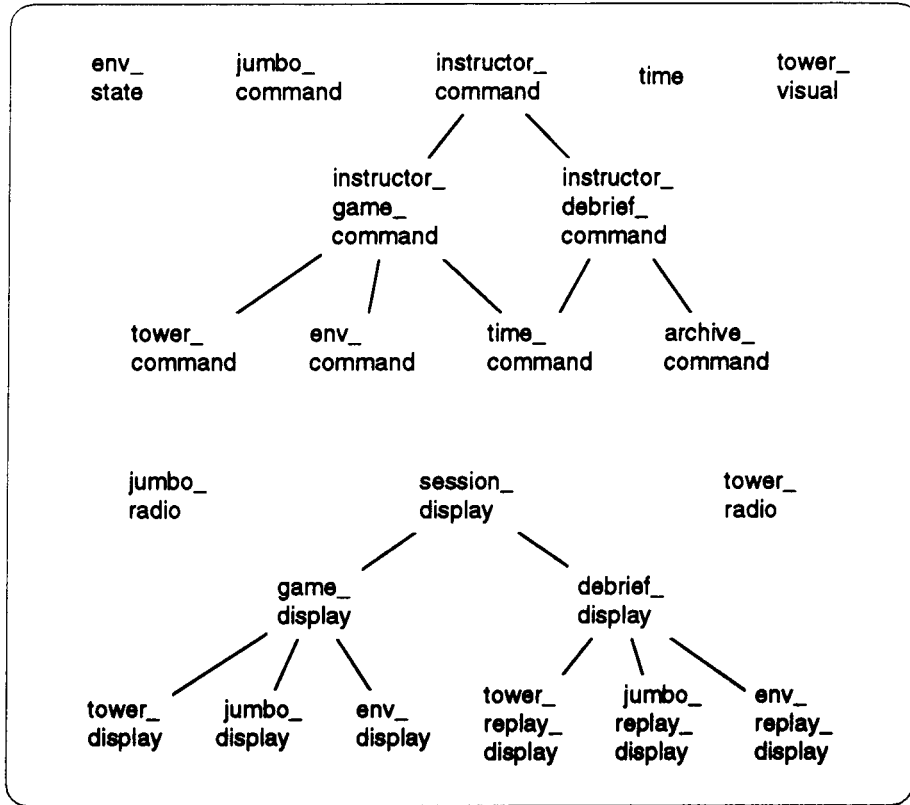


Figure 2.4 Aircraft landing simulator data flow hierarchy

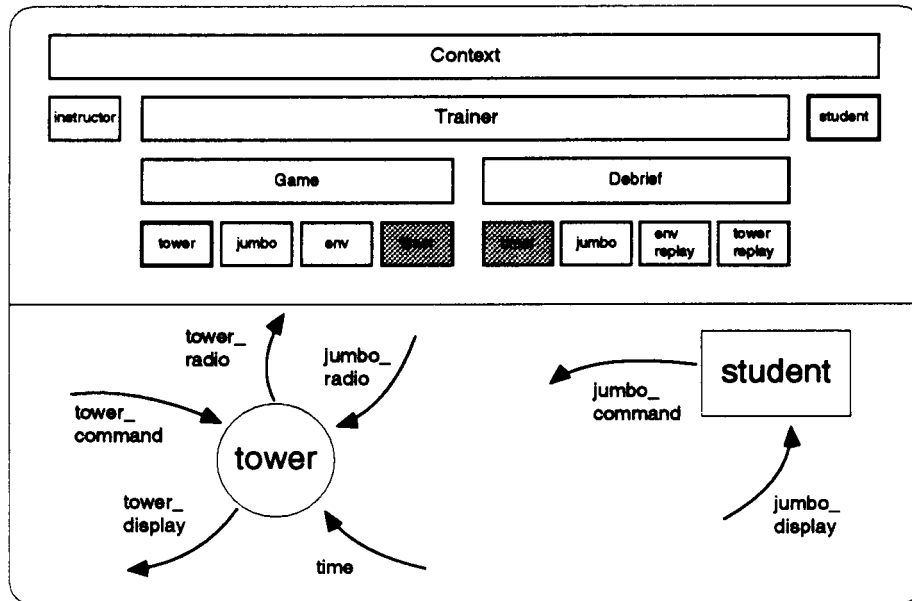


Figure 2.5 User interface before adding a flow

The system has automatically updated all model views affected by this change. The resulting context, trainer and game model views appear in figure 2.7. Using conventional tools this would have taken three steps to add the flow and re-establish model balance, taking one step to change each DFD affected. With our approach it takes one step to add the flow.

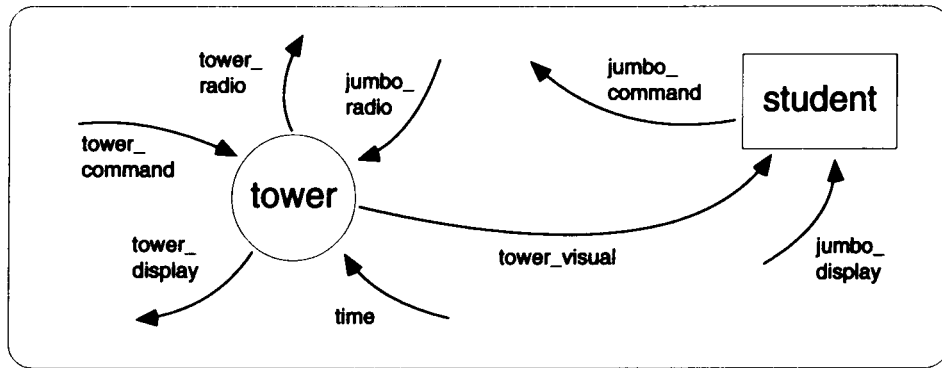


Figure 2.6 The new model view of tower and student

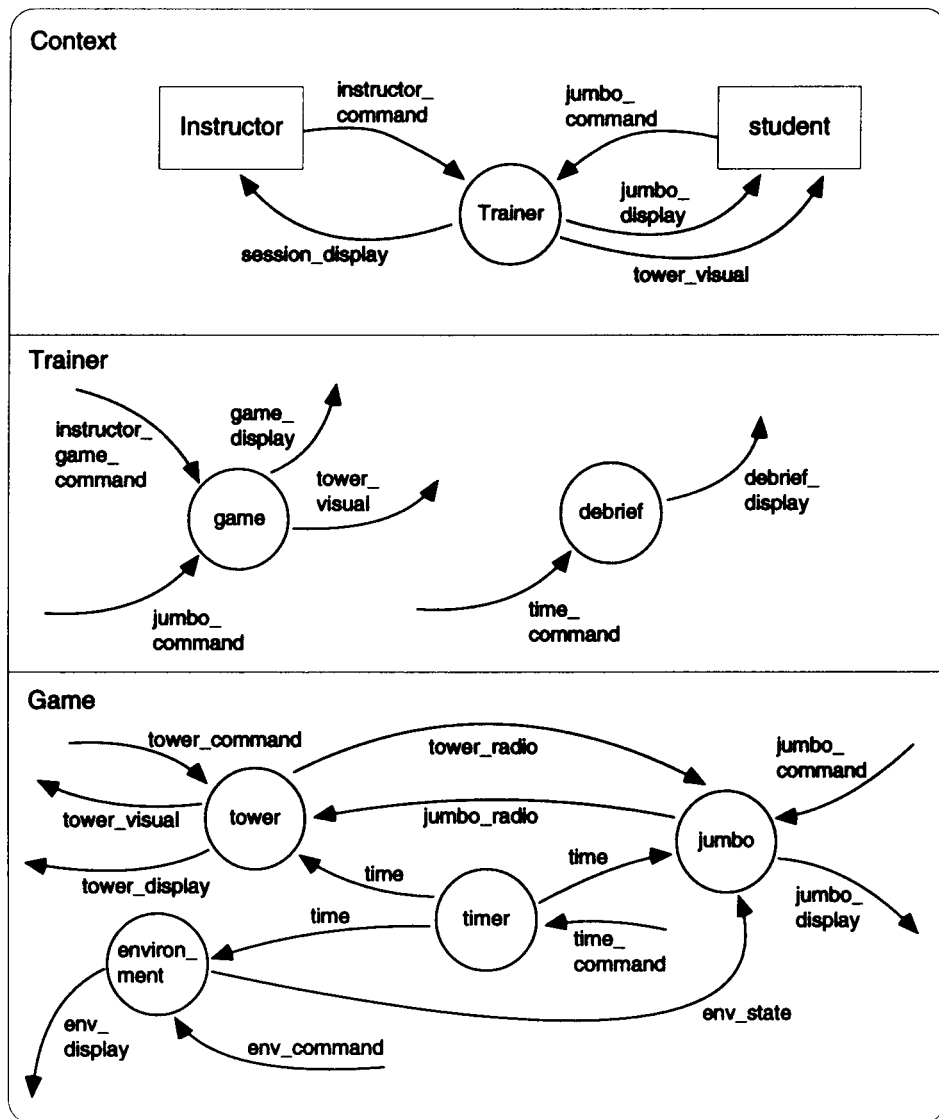


Figure 2.7 The updated aircraft landing simulator model

Consider the resulting model, if the user now added the `tower_visual` flow as an input to the `instructor` terminator, and made `tower_visual` a child of `game_display`. There would be no change to the context, trainer and game

model views, as `tower_visual` is a component of `game_display` and `session_display` flows, which already appear in the process parent path between `tower` and `instructor`. This illustrates that flow interfaces between processes are shown at their most summarised level possible, making full use of the data flow dictionary.

Now suppose the user wishes to remove the `tower_display` flow. The user selects the data flow hierarchy which is displayed as shown in the top of figure 2.8, then selects the `tower_display` data flow. The model view of `tower_display`, which includes its producer and consumer processes (the `instructor terminator` and `tower` process) is displayed as shown in the bottom of figure 2.8. The user then selects the `tower_display` flow in the model view and removes it. The updated model view is shown in figure 2.9.

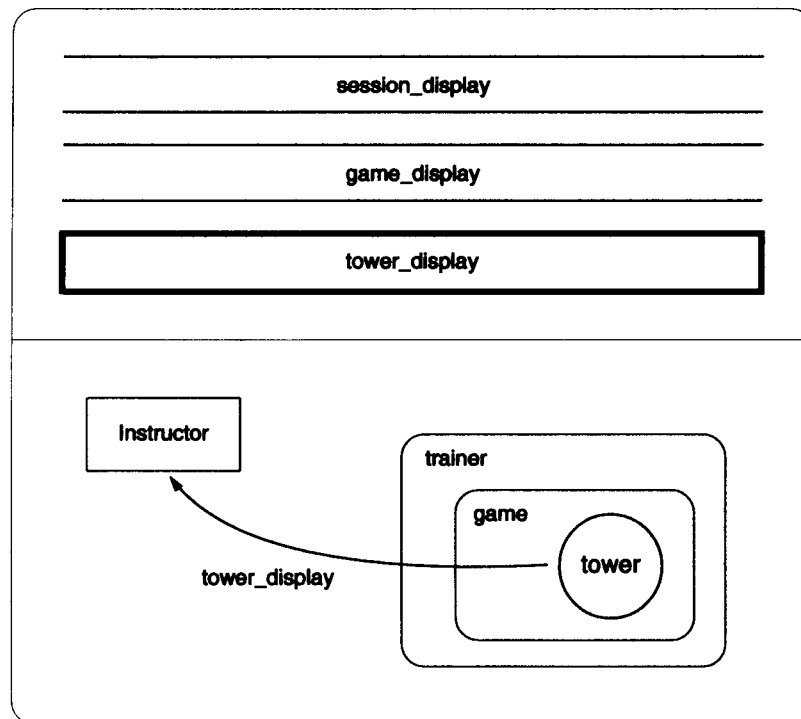


Figure 2.8 The model view before removing the `tower_display` flow

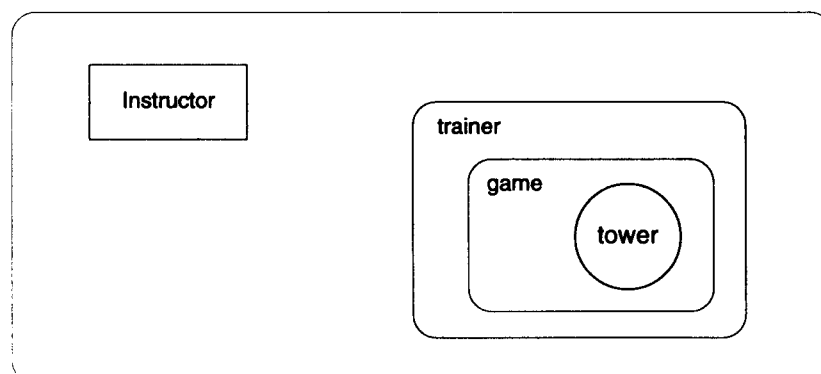


Figure 2.9 The model view after removing the `tower_display` flow

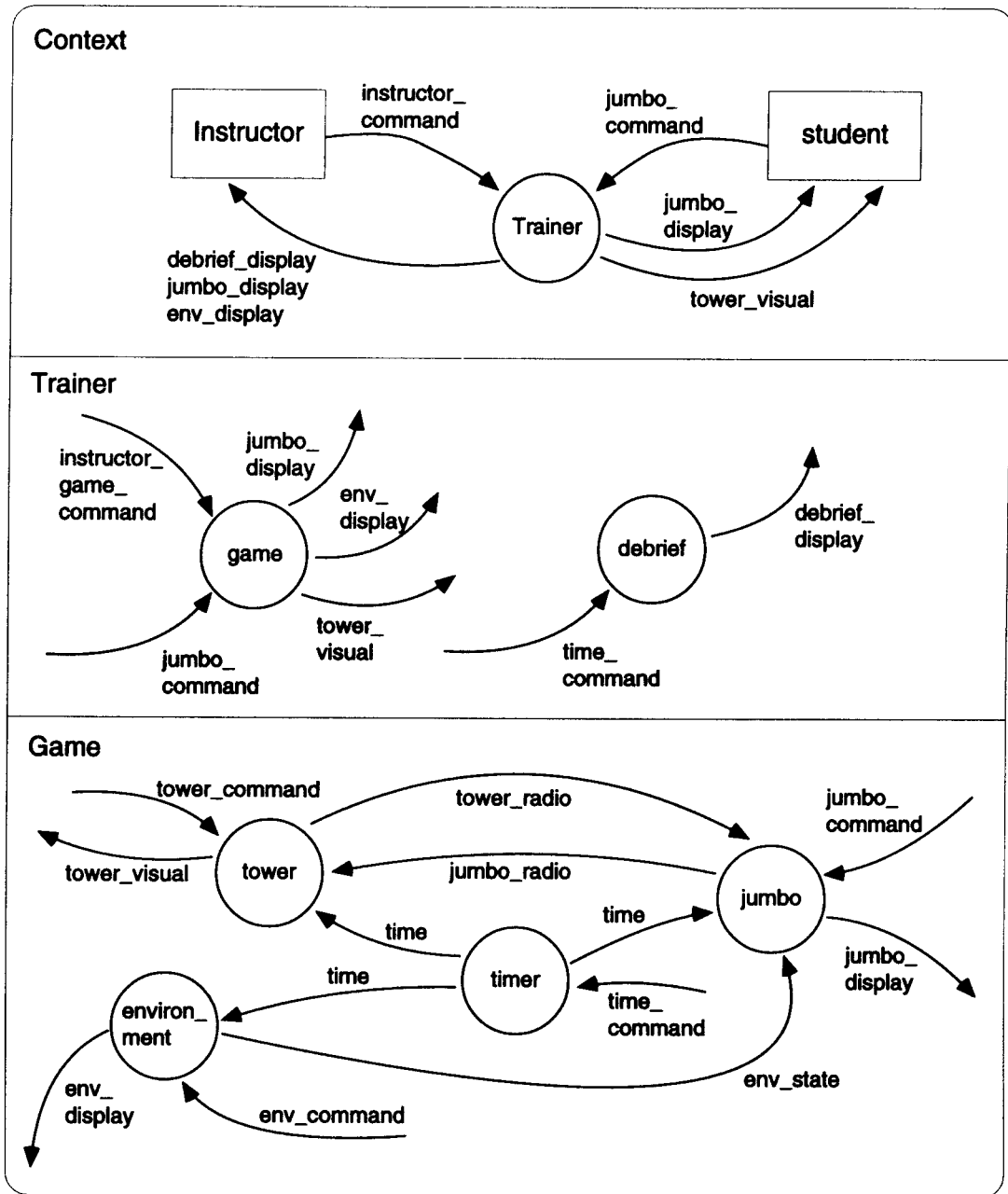


Figure 2.10 Updated model

The user has removed the `tower_visual` flow with essentially one step, while the system ensured all other model views are kept consistent. If the user were to view the context, trainer, and game model views they would appear as shown in figure 2.10. In the game model view `tower_display` itself has been removed. In the trainer model view `game_display` has been replaced by `jumbo_display` and `env_display`. In the context model view the `session_display` flow has been replaced by: `debrief_display`, `jumbo_display` and `env_display`, as this is the most summarised representation of `session_display` minus

tower\_display. Using conventional tools this change would have taken at least three steps to remove the flow and re-establish model balance, taking one step to change each DFD affected. With our approach it takes one step to remove the flow.

Next let us look at a larger change to the model. We decide that the timer process should not appear twice and should be moved under the trainer. Traditionally this would mean updating three diagrams. All flows to timer in the game and debrief DFDs must be changed to off-page flows; also, the timer process itself must be removed from these diagrams. Then the timer process must be added to the trainer DFD and time flows into the game and debrief processes must be added. This would be about eight user steps.

With our approach the user selects a process view of timer and its connected processes: those processes which have flows between them and timer. The user selects to show only timer's flows. Again, the seven processes in the selected view are shown with a thick border box in figure 2.11. The associated model view is automatically generated and shown in figure 2.11.

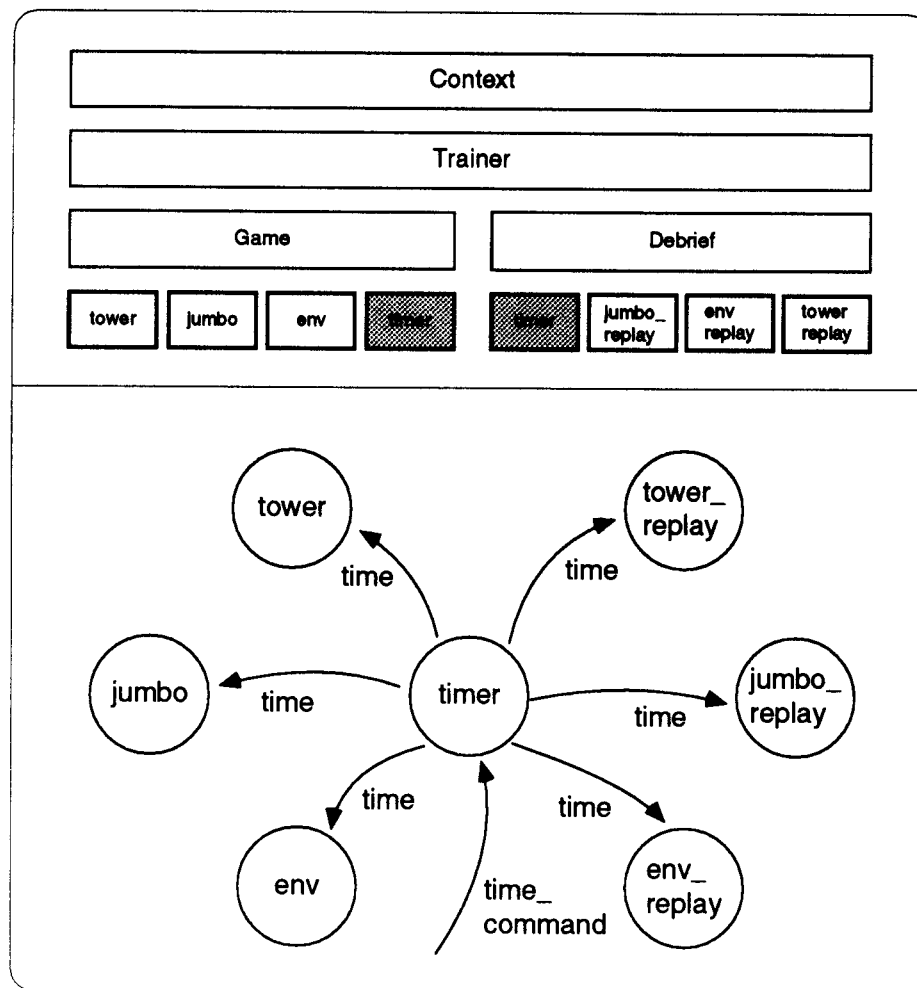


Figure 2.11 The user interface before moving timer



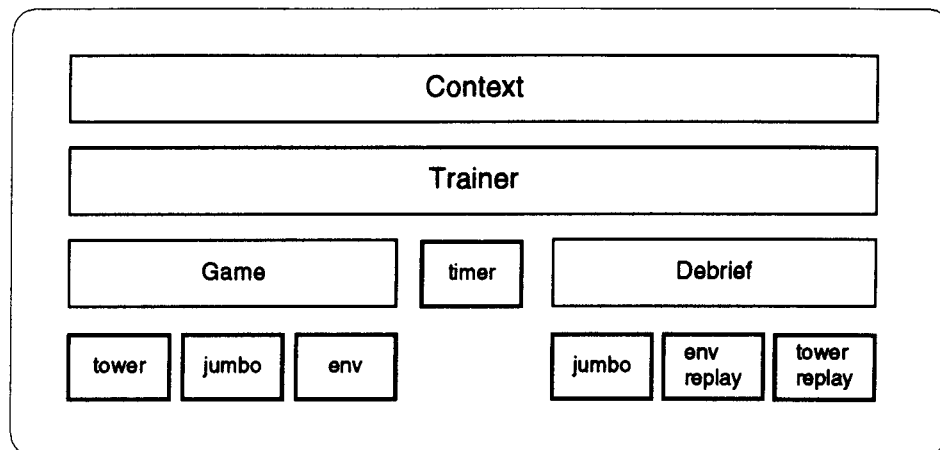


Figure 2.12 The process hierarchy after moving `timer`

To achieve the move the user clicks the `timer` process and selects the move menu, then chooses `trainer` as the destination parent. Again, the system automatically updates all model views affected by this change, keeping the model balanced. Also the process hierarchy view is updated. The move has not affected the current model view though; all connections between `timer` and environment, `jumbo`, `tower` and the respective replay processes are unchanged. The process hierarchy has changed and is shown in figure 2.12. The resulting `trainer`, `game` and `debrief` children model views are as shown in figure 2.13.

This move has taken the user essentially one step. Even in this small model, this move would have taken around eight steps with conventional tools to move the process and re-establish model balance. If the model was more realistic and larger, many more steps could be required for a similar move process operation. But with this system it would still only be one user step.

Another property to note distinct from editing is the flexible browsing capability. Any set of non-comparable processes can be selected as the current process view, and the associated graph of flow between these processes can then be shown in one user step.

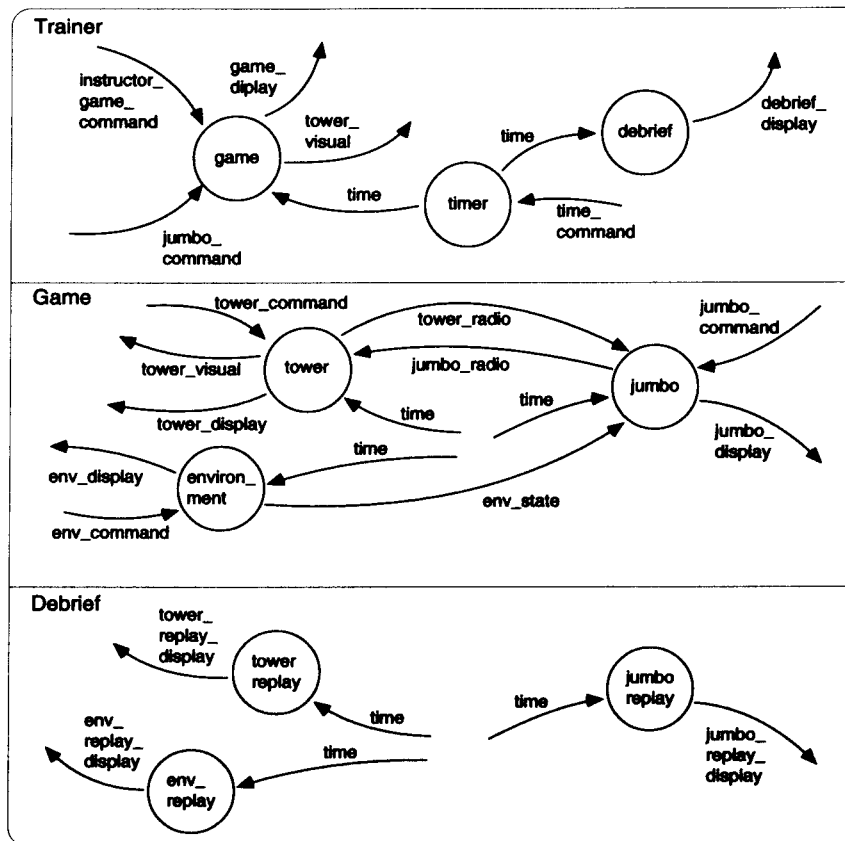


Figure 2.13 The model after moving timer

### 2.3 Discussion

In this chapter browsing and editing of a structured analysis model using a hypothetical structured graph based tool has been demonstrated. The browsing was scalable as a user is able to directly view the data flow diagram formed by a selection of arbitrary processes. The user's effort is required, only to choose the processes of interest, using the process hierarchy display. The editing was scalable as any DFD view generated when browsing can then be directly edited, so the data flows can be directly added between arbitrary processes. Further more, the entire model can be restructured by editing the process hierarchy, to move a sub-model to a new location whilst keeping the model balanced.

The major limitation is: a user only determines the process hierarchy, and the producer and consumer processes of each data flow are; semi-automatic layout of model views is required to position processes and data flows. In practice a user would indicate the positions of most processes and some data flows in the traditional DFD views leaving the system to generate layouts for other model views. There are other limitations, but these will be given in the more general context of structured graphs which are presented in the next chapter.

---

## Browsing and Editing Structured Graphs

---

Structured graphs were designed to generalise the scalable viewing approach described in Chapter two. There, a structured analysis model was browsed and edited in a scalable fashion with a hypothetical tool. The constraints which facilitated the scalable operations, that the model is always balanced and interfaces are shown summarised, were not dependant on the interpretation of the diagrams just on their structure. Model structure was determined by the parent to child relationships between processes and data flows, and the input and output relationships between processes and data flows (stores were not considered). So it is the relationships between processes and other processes, data flows and other data flows, and between processes and data flows that is relevant here. A structured graph captures just these relationships.

Structured graphs are used to provide a visual representation of models within a problem domain, be it software analysis, project planning or network management for example. The use of structured graphs could extend these tools, so that a tool user is given more representation choices in their modelling. Their modelling vocabulary and their ability to flexibly compose this vocabulary could increase. For example, traditional structured analysis models use process and data flow hierarchies which are trees, structured graphs when applied back to structured analysis, would allow arbitrary process and data flow hierarchies to be supported. Also, browsing and editing is with respect to an arbitrary view rather than just sibling views (DFDs).

Recall the second contribution of this thesis, stated in section 1.6, was to present a general formalism, structured graphs. Such a formalism should use terminology which is not specific to a particular application. For example, the graph elements in structured analysis are processes and data flows, whilst in project planning they are tasks and deliverables. The more generic terms, *nodes* and *links*, are used for structured graph elements. Further, in the following *structured graph* will sometimes be abbreviated to *model*. A listing of definitions for these and other terms introduced in this thesis, is in Appendix A, Glossary. An example, which presents a model as structured nodes and links, and then as DFDs of processes and data flows, is in Appendix D.

Models can be built in a bottom-up or top-down manner. Sections 3.1 and 3.2 present models from a bottom-up perspective. They present a model as an underlying network of nodes and links, upon which node and link hierarchies have been built. A collection of summary networks for browsing can then be automatically derived. Section 3.3 extends models to include incomplete models built top-down. In our formalism

complete models are a special case of these. Section 3.4 briefly presents editing. Section 3.5 describes limitations of models, the need for context dependent links in special cases and the inability of some links to ever appear in a model view. In summary, we show structured graphs support both bottom up abstraction and top-down refinement.

### 3.1 Network Abstraction

The starting point for a model is a network of nodes and links. For an application, nodes and links are usually distinguished by unique labels. Each node may have several input links and several output links, while each link may have several producer nodes and several consumer nodes. This is more general than say Data Flow Diagram (DFD) graphs, where a data flow is restricted to a single producer, but having this symmetry between nodes and links allows a more general model to be described, while allowing a simpler formal description. An example network is shown in figure 3.1.

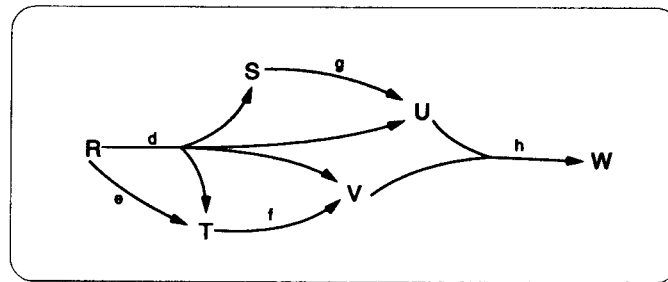


Figure 3.1 A network of nodes and links

When a network of nodes and links becomes large it is difficult to browse and edit. Clearly some kind of abstraction is needed. One technique is to partition the network into blocks, where each block is a small network, then to treat each block as an abstract node. Note as the network is symmetrical, the partitioning into blocks could have been done on nodes or links, but partitioning on nodes matches the conventions of typical applications. Usually this partitioning into blocks will need to be applied recursively to the abstract nodes (blocks).

When the abstract nodes are ordered by set inclusion (of nodes), they form a tree. An example of this is shown in figure 3.2. The whole network is represented by Z, and this has been partitioned into two blocks, X and Y. In figure 3.2 the partitioning is firstly shown as a division of the network, then secondly as an order, where blocks are ordered by set inclusion of the underlying nodes. A nested partition when so ordered will always be a tree.

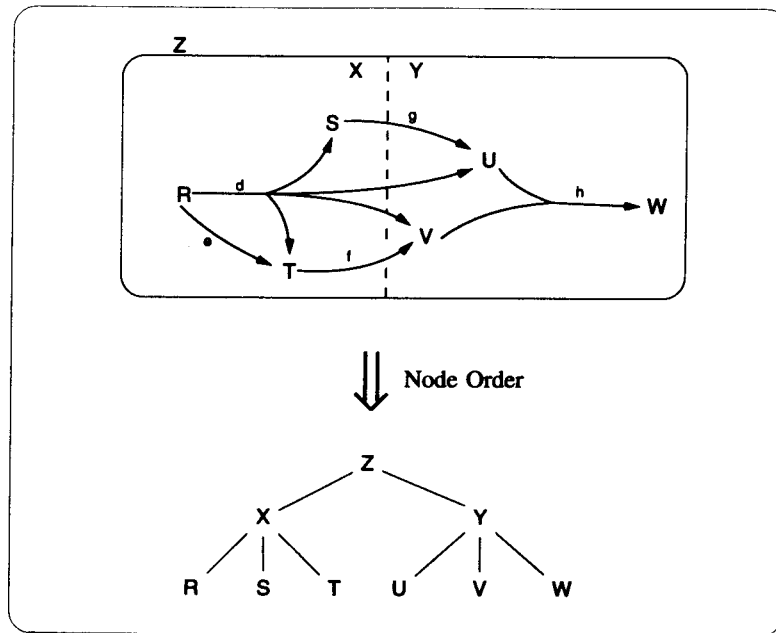


Figure 3.2 A partitioned network

Sometimes more than one partitioning may be appropriate, so the final node hierarchy which is the superposition of these trees is not a tree, but an ordered set. Figure 3.3 shows this. The network is represented by two alternative partitionings, Z and Q. The Q partition comprises the blocks M and N. In general an arbitrary number of alternative partitions can be used, with each partition giving rise to a tree.

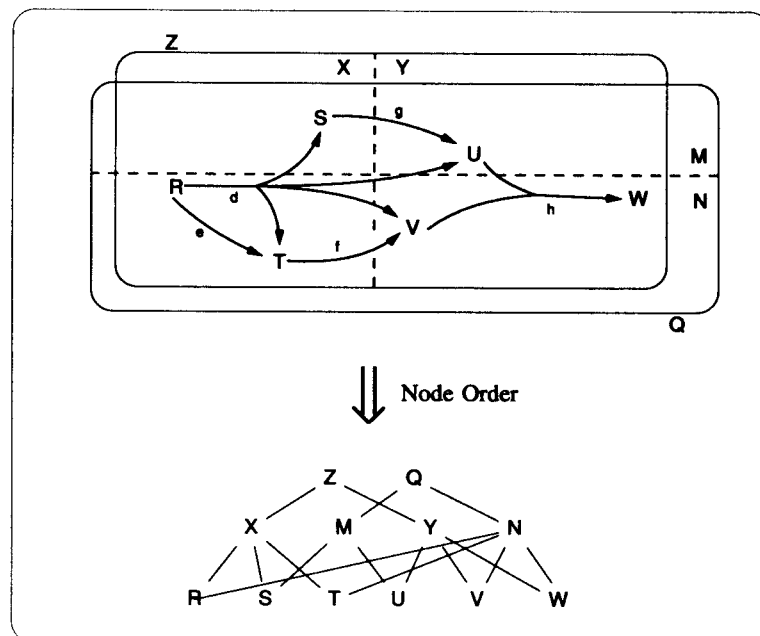


Figure 3.3 A multiply partitioned network

We have seen how partitioning the network into blocks results in networks which contain a comprehensible number of abstract nodes, but the situation for links has got worse. As blocks get bigger, the number of links between blocks increases. This is

alleviated by replacing groups of links by summary or abstract links. When this is done recursively the result is a link order. This is shown in figure 3.4.

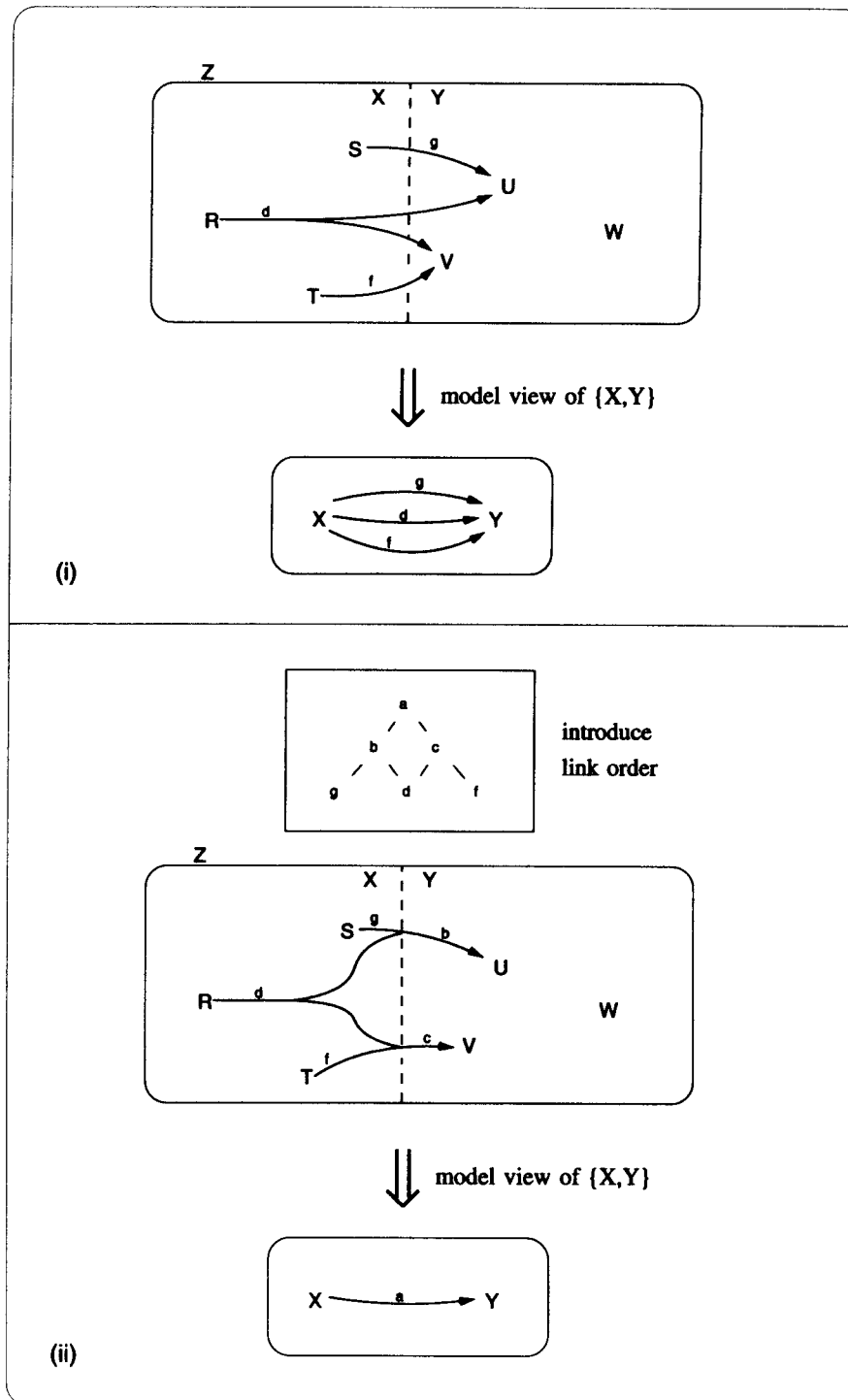


Figure 3.4 A partitioned network with summarised interfaces

Once a link order is established as in figure 3.4 there is a choice of interface, though all choices will have the same underlying links. In 3.4 the choices are {a}, {b,c}, {b,f}, {g,c} and {g,d,f}. The choice used {a}, was the most summarised interface. The most summarised (maximum) interface is used because this can be completely

derived from the underlying links and the link order. No additional user input is required. This minimises the number of input and output links appearing in a model view when the link order is a tree or tree like.

Future work could allow additional information to be added so that specific model views have non-maximal interfaces, but this could impact on the scalability of editing operations. This thesis does not address the generation of a model view's graph layout. It is assumed to be automatic by default, with a user having the option to modify a layout. However, allowing layout modification would add layout information for specific views to models, and this could also impact on the scalability of editing operations. There is a substantial body of research on automatic graph layout (Di Battista et. al, 1994) and this should be considered in any future full graphical implementation of this work.

We have started with a network of nodes and links, and after partitioning the network and summarising interfaces, node and link orders have been generated. With this approach, it is the underlying network which captures all dependency information between nodes and links. Abstract nodes and links are just summaries of the underlying nodes and links, and networks of abstract nodes and links are just summaries of the underlying network itself. It is this perception of models which results in all models having convexity of producer nodes: that is, if a node and its ancestor node produce a link, then any nodes between these also produce the link, and the same applies to consumers.

Traditional CASE tools, for example structured analysis tools, support a different perspective of a model. They represent a model as a collection of DFDs which are networks of nodes (processes) with summarised link (data flow) interfaces. Each process appears in only one DFD with its siblings (the other children of the process's parent). Because such tools directly implement the original paper based system, no networks of nodes from different diagrams are available for display to a user. However, using our perspective of an underlying network there is no reason to restrict the abstract networks which a user can see to just abstract networks containing sibling nodes. Networks can be generated from the underlying network, and node and link orders. Such networks are shown in figures 3.5 and 3.6. These figures also show how a user selects model views, by first selecting the domain (the submodel of interest), then selecting the nodes and links of interest.

If the model is very large the user first selects the submodel to be viewed, the domain. All nodes in the selected domain are shown shaded. In the example shown in figure 3.5 the whole model is chosen. Next a cross-section of nodes is chosen; this would be done while the user is looking at the node order, keeping in mind each node in this cross-section represents some portion of the underlying network. Nodes X,U,V and W

are selected. Finally the user may restrict which links appear in the resultant view. All links are selected in the example. Note while the node selection is flat (that is all nodes are non-comparable), the link selection must be down complete (that is, for each link included, its descendants are included as well). An example where the user does restrict the domain and links is shown in figure 3.6.

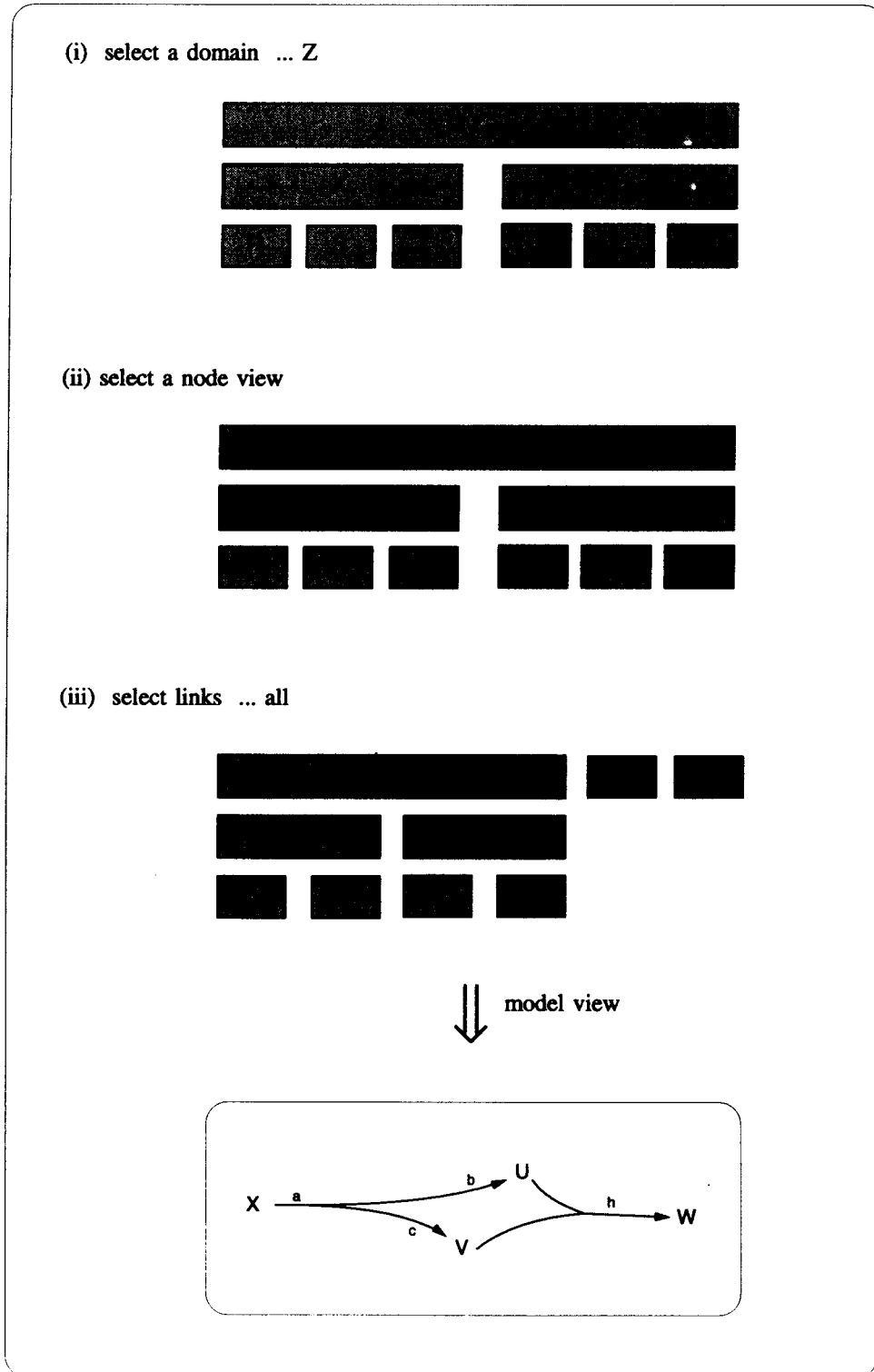


Figure 3.5 Selection of a model view (I)



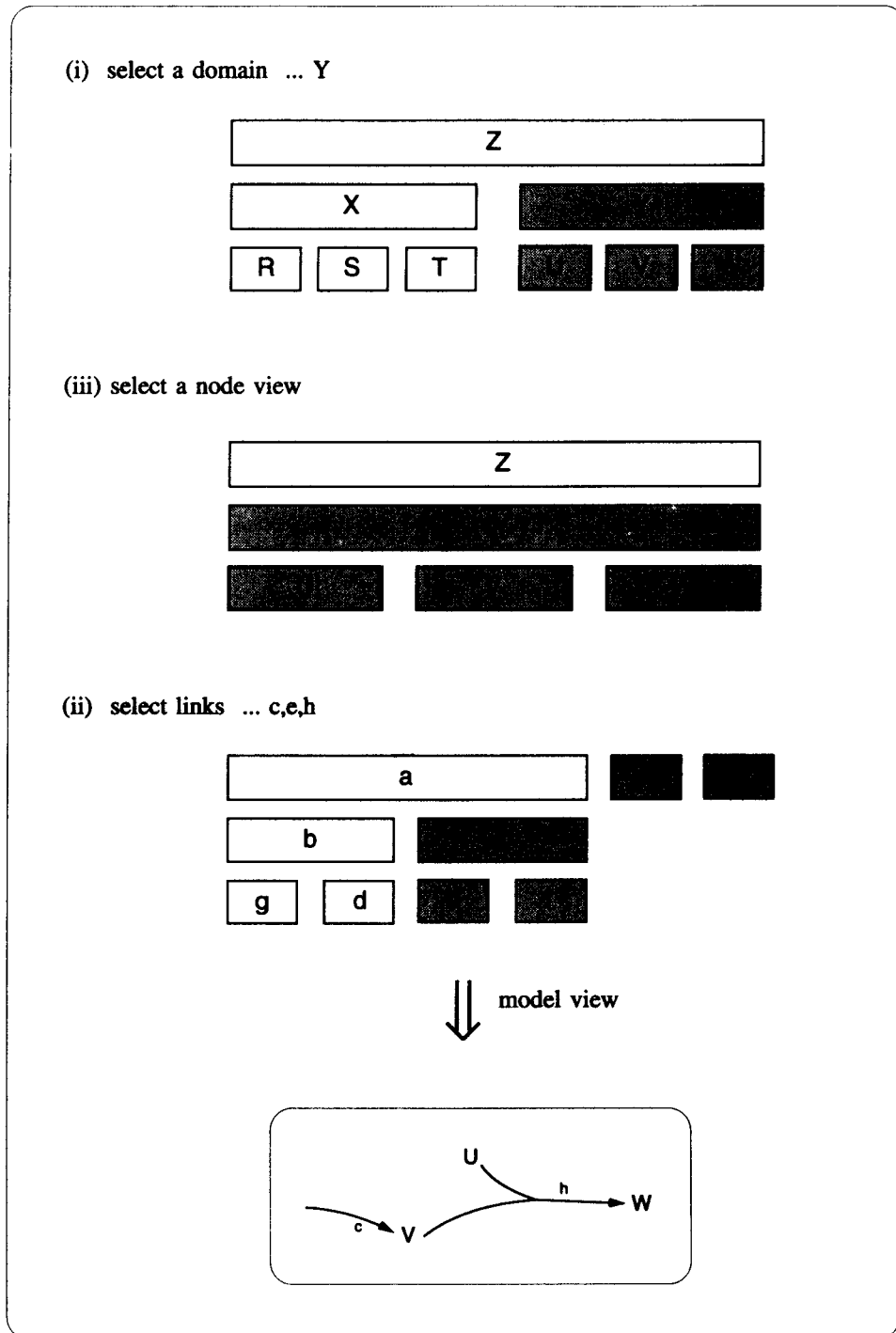


Figure 3.6 Selection of a model view (II)

A model is an underlying network of nodes and links with a node and link order. This is different to the user perspective of a model. Their perception comes from how they interact with the model. They are able to browse a model by looking at model views. So for a user a model could be the collection of all model views. Figure 3.7 shows such a model. Each node is shown with its summarised interface and links. A desired property for models is that node interfaces should be context free as this will provide the least surprise to a user. This has been achieved by our network layout style, which

allows the merging and splitting of links, combined with net interfaces. This will be explained in more detail in the next section. Given that the local interface of each abstract node is context free, a model view can be constructed by collecting nodes with their local interfaces, then joining an output link to an input link when they have some common underlying links. In this way figure 3.7 shows enough information to construct all model views, and because these local interfaces are already summarised the link order is not further required.

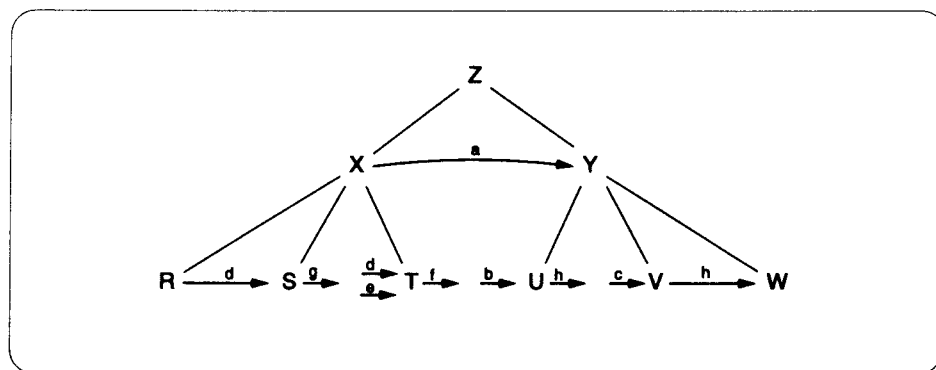


Figure 3.7 The local interfaces of a model

### 3.2 Browsing Models

This section will look at model browsing in more detail. A major challenge in generalising the model browsing in Sifer and Potter (1995) was determining the interface between two abstract nodes which overlap. Consider figure 3.8 which shows examples of this. Firstly (from left to right), it shows a node order above an underlying network, then it shows the underlying network partitioned into  $P$  and  $Q$ , and rightmost it shows the model view of  $\{P, Q\}$ . The rule used in the figure is: an abstract node produces a link as a net output when the link is produced within the portion of the underlying network that the abstract node represents, and the link is consumed by a node in the underlying network which is outside the network represented by the abstract node.

Figure 3.8 (i) shows a straightforward example where the node order is a tree. Link  $a$  is produced in  $P$  by node  $V$  and consumed outside  $P$  by  $X$  so it appears as an output of  $P$ . Also, link  $a$  is consumed in  $Q$  by node  $X$  and produced outside  $Q$  by  $V$  so it appears as an input of  $Q$ . In case (ii) a link appears between  $P$  and  $Q$  for similar reasons except the underlying nodes involved are  $V$  and  $Y$ . Case (iii) is different however, abstract node  $P$  produces link  $a$  but  $Q$  has no input in the model view of  $\{P, Q\}$ . Here, link  $a$  is a net output of  $P$  but not a net input of  $Q$  because, though link  $a$  is consumed within  $Q$  it is not produced outside  $Q$ . In case (iv) the model view of  $\{P, Q\}$  contains no links, as the link  $a$  is internal to both abstract nodes  $P$  and  $Q$ .

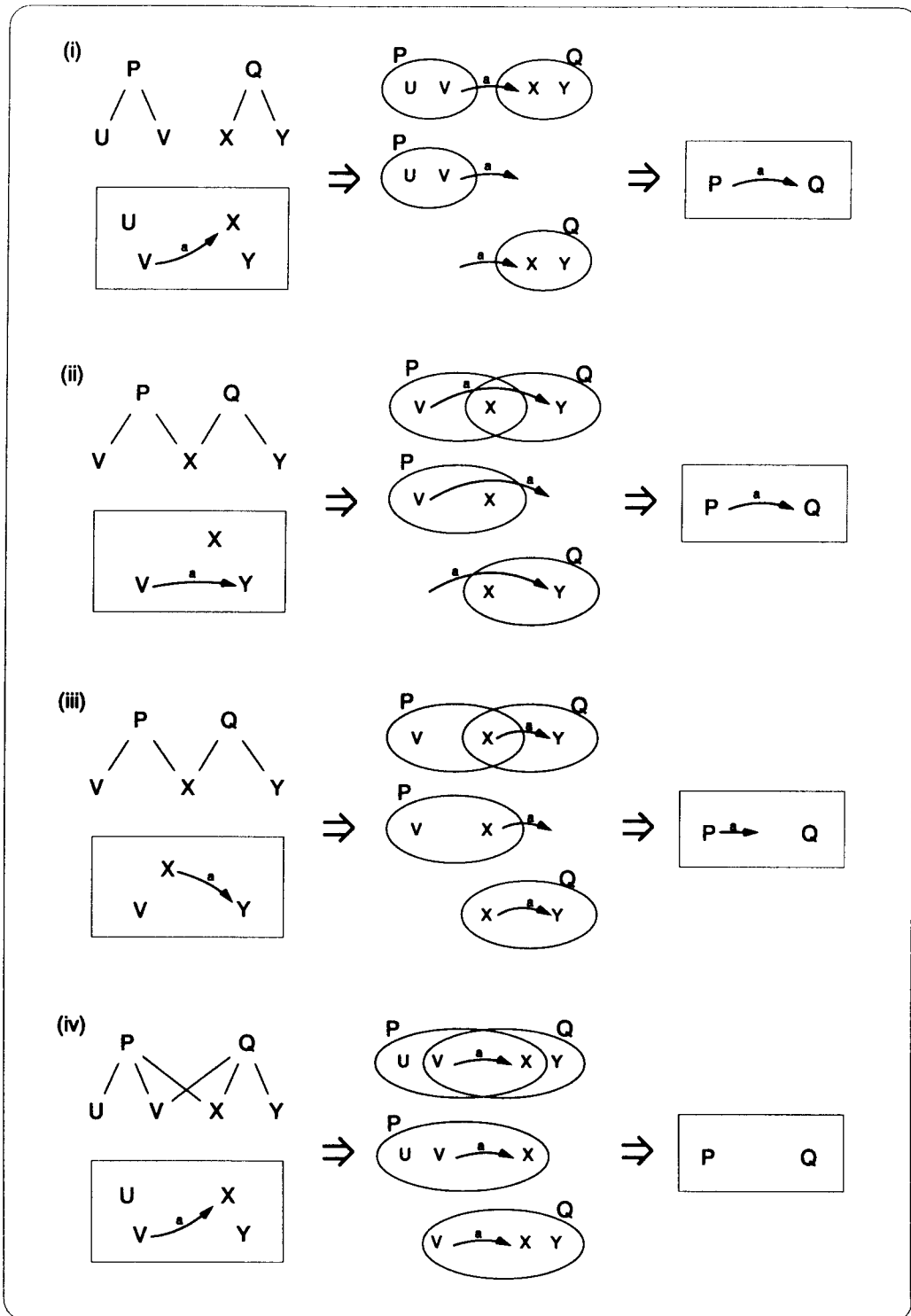


Figure 3.8 A comparison of rules for generating a net interface

Another consequence of our definition of the underlying network is, there can not be two links with the same name but which have different producer and consumer sets. Once a link's producers and consumers are identified, we say that all producers connect via the link to all consumers in the underlying network, and given the definition of net interfaces for abstract nodes this will apply to abstract producers and consumers too.

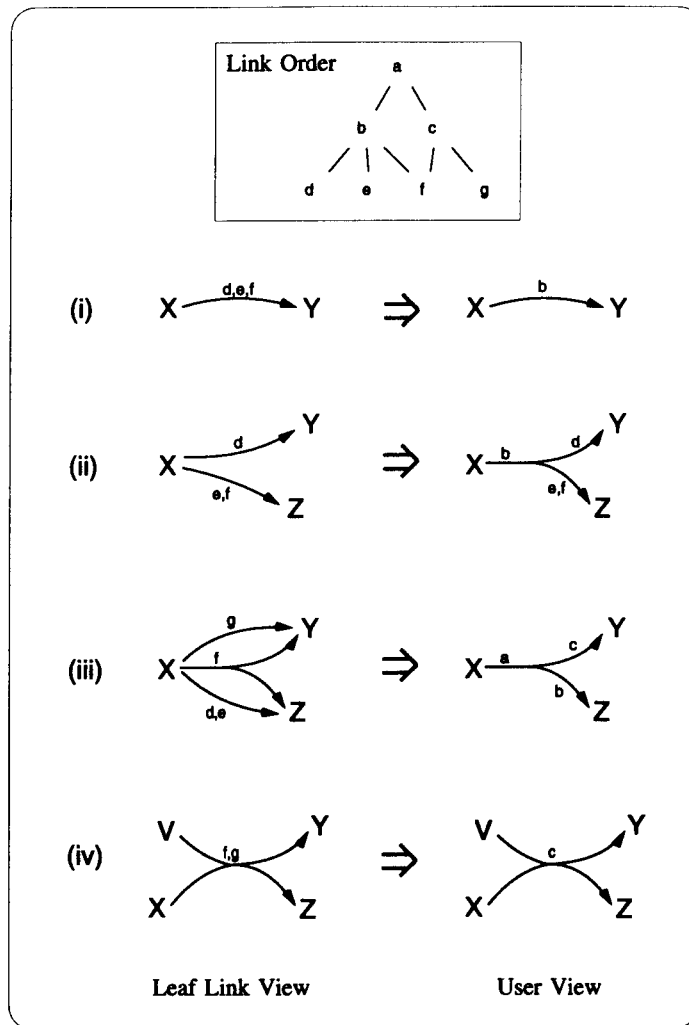


Figure 3.9 Networks with branching and splitting links

The network layout style used in this thesis was used in previous structured planning work (Wilson and Sifer, 1990). Figure 3.9 shows some examples of network layout. The left hand side of 3.9 shows underlying networks and the right-hand side shows the same network with summarised interfaces. Each node's inputs and outputs are summarised separately, so in (ii) the output of node X is summarised to *b*, while inputs to Y and Z are left unchanged as they cannot be summarised. The diagram should be read as: X produces link *b*, the *d* portion of *b* is consumed by Y, and the *e* and *f* portion of *b* is consumed by Z. In cases (iii) and (iv) all interfaces have been summarised. Case (iv) also shows an example with multiple producers. Again, each nodes inputs and outputs have been summarised separately then joined up when the leaf links they represent intersect. It is the combination of, this network layout style and net interfaces, which achieves the aim of a context independent node interface under model viewing. In this style of figure we sometimes show connections between producers and consumers but need not do so.

### 3.3 Incomplete Models

In a *complete* model all links have leaf producer and consumer nodes. A model which is not complete is *incomplete*. Incomplete models must be supported to allow flexible user editing, so final producers and consumers of a link can be left undefined. This allows a link's producers or consumers to be defined first. But, at all intermediate steps the model needs to be viewable, so any model viewing system must support browsing and editing of incomplete models. The formal description in the next sections will deal with incomplete models rather than complete models.

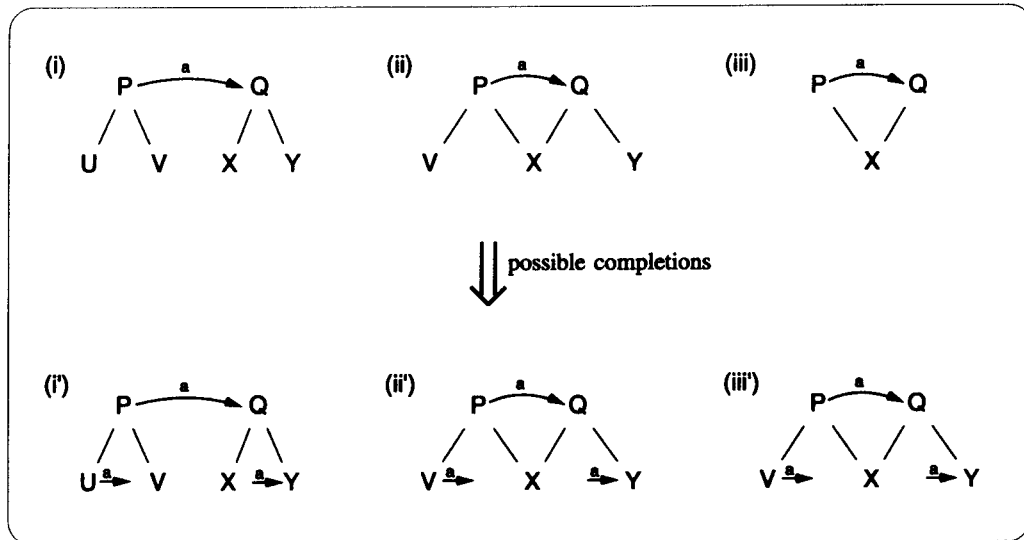


Figure 3.10 Three incomplete models which contain undistributed flows

A model may be incomplete in two ways. Firstly, some leaf flows may have producers and consumers but no leaf producers or consumers. These are called *undistributed flows*. Our interpretation of this model is, that the flow's producers and consumers will later continue to be made more specific, until all lowest producers and consumers are leaf nodes. But until this is done, the producer and consumer relationship between nodes and links is not fully captured by the underlying network.

Examples of incomplete models with undistributed flows are shown in figure 3.10 with some possible completions. In (i) link  $a$  has sources (lowest producers and consumers)  $P$  and  $Q$  which are not leaf nodes. This is read as, it is not known at the moment which nodes below  $P$  and  $Q$  will produce and consume link  $a$ . A possible completion for this incomplete model is shown in (i'). Case (ii) is similar except  $P$  and  $Q$  overlap. In case (iii) the node order itself must be extended to get a completion. Making node  $X$  the producer and consumer would not work for two reasons. Firstly, it would introduce a cycle into the underlying network as  $X$  would output and input the same flow. Secondly, even if a cycle was allowed, letting  $X$  produce and consume link  $a$  would leave  $P$  with no net output and  $Q$  with no net input so this change would not preserve the existing views and hence not be a completion.

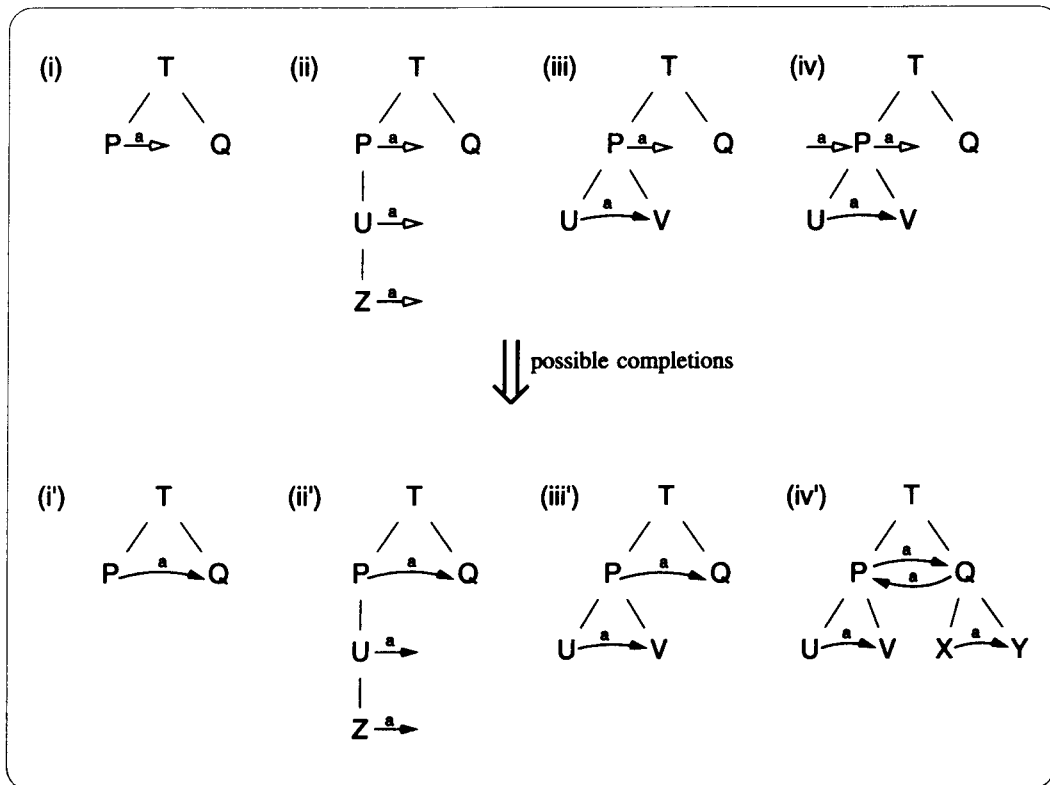


Figure 3.11 Four incomplete models which contain half flows

The second reason a model may be incomplete is: a link may have missing producers or consumers (half flows). Half flows make a model incomplete as the underlying network cannot capture what the flow's highest producers or consumers should be. Examples of models with half flows and possible model completions are shown in figure 3.11. Half flows are visually distinguished from full flows by having an empty arrow head, so their presence highlights a model's incomplete state. A non-leaf flow is shown as a half flow when it summarises at least one leaf half flow. In (i) and (ii) link  $a$  has only producers, but in (iii) link  $a$  has a full flow from node  $U$  to  $V$  and a half flow from  $P$ .

The property required of all half flows, as shown in (i)-(iv), is that there is no gap in the flow path. That is, a half flow's producer or consumer node set is convex. Completions for (i) to (iii) are straight forward but note in (iv) that the node order needed to be extended. This avoids having a cycle in the underlying network, the same problem encountered in figure 3.10 (iv). Also in (iv'), the view  $\{P, Q\}$  contains a link  $a$  from  $Q$  to  $P$ , as  $X$  in  $Q$  produces link  $a$  and  $V$  outside  $Q$  consumes link  $a$ . This is a reminder that for every flow, all producers connect to all consumers.

Recapping: an incomplete model is a model that cannot be fully described by an underlying network with node and link orders. The incomplete model will be described by: a network of nodes (underlying or abstract) and leaf links, the highest producers and consumers of half flows, and node and link orders.

### 3.4 Editing Models

This section will demonstrate some model editing: adding a flow and removing a flow. These operations must preserve the constraints which define a model. These are: node interfaces are to be context free, the maximum amount of abstraction is used, and all interfaces are to be net interfaces. Some examples of editing are shown in figure 3.12.

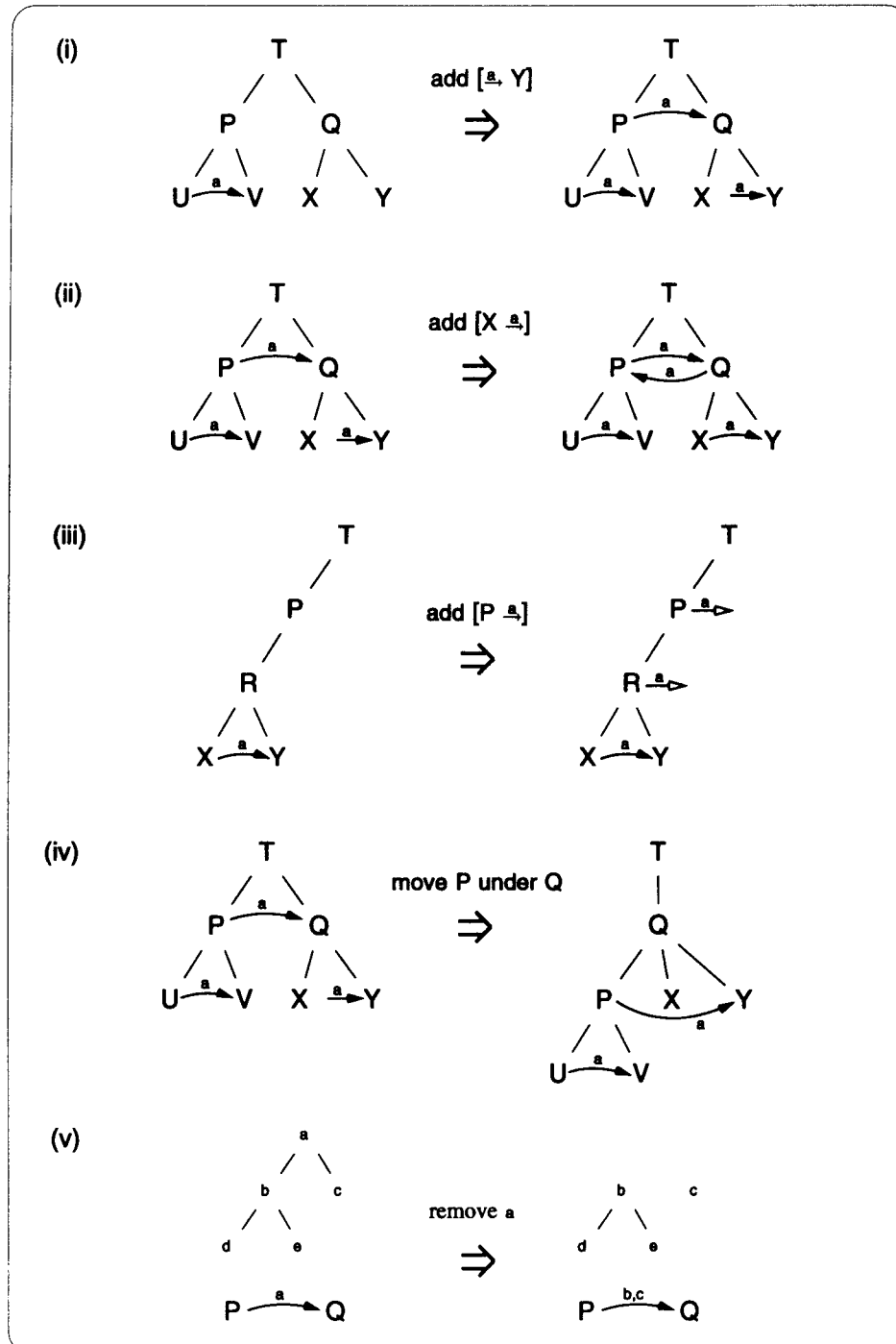


Figure 3.12 Examples of model editing

In (i) link *a* gains an additional consumer node *Y*, so *P* becomes a net producer of link *a* and *Q* a net consumer of link *a*. Case (ii) extends (i) by making node *X* a producer of link *a*, so *Q* becomes a net producer of link *a* and *P* a net consumer of link *a*. Now both

$P$  and  $Q$  output and input link  $a$ , but these are only apparent cycles, which indicate that link  $a$  must have multiple producers and consumers, with at least one producer and consumer in  $P$  and  $Q$ . Case (iii) shows  $P$  becomes a producer of link  $a$ , keeping the producer set convex,  $R$  also becomes a producer. In (iv) node  $P$  is moved under  $Q$ , note this has only changed the node order, there is no change to the underlying network. In (v) abstract link  $a$  is removed from the link order, so the summarised interface between  $P$  and  $Q$  becomes links  $b$  and  $c$ .

### 3.5 Model Limitations

It is a claim of this thesis, that structured graphs have scalable browsing and editing operations. In this section the model properties (or constraints) which ensure that browsing is scalable are given. Three limitations of structured graphs are then given: a lack of explicit external flows in some situations, the occurrence of apparent half flows in some model views, and the inability of some links to appear in any model view. These limitations can be partially resolved but at the cost of decreasing scalability.

Three properties allow model browsing to be scalable. Firstly, a node in a user's view always has the same set of link inputs and outputs independent of the other nodes which appear in the view. We call this the context free property for node interfaces. It allows a user to associate a fixed node interface with each node. The second property ensures each node's input and output links are shown at their maximum level of abstraction. The third property is the critical one: it defines the content of non-leaf node interfaces. A non-leaf node is an abstraction of its collection of descendant leaf nodes. A non-leaf node produces a given link when one of its descendant nodes produces the link and an outside leaf node consumes the link. A non-leaf node consumes a given link when one of its descendant nodes consumes the link and an outside leaf node produces the link. Such non-leaf nodes have a *net interface*. Note the symmetry between producers and consumers. These three properties guide how a user perceives a model via browsing and editing.

Scalability of model browsing and editing for a user is a result of the maximum net interfaces constraint (the combination of properties two and three). However, given this constraint there are three intrinsic limitations of models. Two of these limitations became apparent only with the generalisation of the node and link hierarchies to ordered sets from trees, so they did not appear in our earlier structured planning models.

The first limitation is a lack of explicit external flows in some situations. This limitation is due to the context free property of maximum net interfaces. When a partial model view is selected, it does not abstract the whole underlying network so there are leaf nodes outside the view. Figure 3.13 (i) shows a model view containing nodes  $X$  and  $Y$  with node  $Q$  being outside this view. Whether a node produces a link which is only



consumed locally (by a node also in the model view), or by a local node and an external node there is no diagrammatic difference. In figure 3.13 (ii) an external consumer, node  $Q$ , is introduced but there is no change to the net interface model view of nodes  $X$  and  $Y$ . The key problem is: existing structured analysis tools do highlight the existence of an external consumer with a flow which has no consumer node as is shown in (iii). We would like model views to show these external links without violating the context free node interface property.

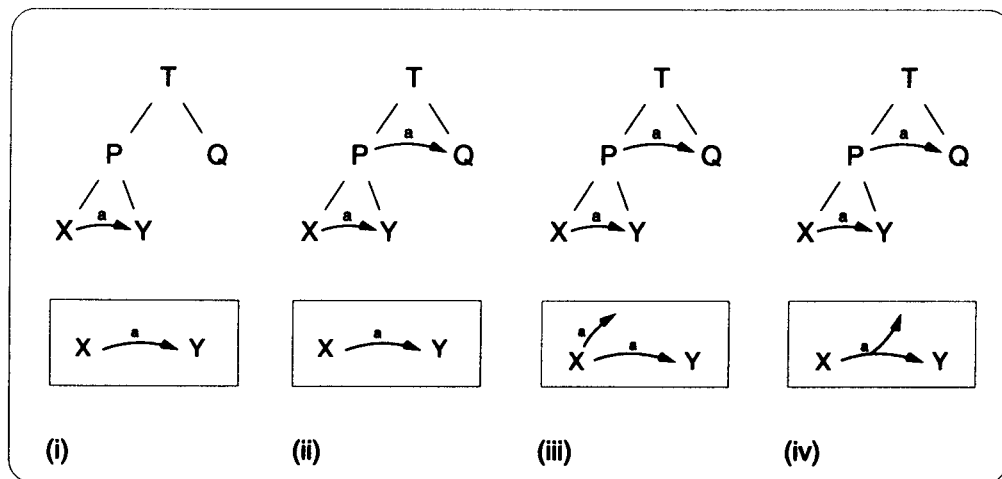


Figure 3.13 Models with and without external flows

To ensure node interfaces remain context free, external flows are shown as a flow to a hidden environment node. The environment node represents all the nodes outside the current view. For example, in figure 3.13 (iv) link  $a$  has a local consumer  $Y$ , and an external consumer indicated by the second arrow which has no local consumer node. The rules for adding an explicit external flow are:

- (i) Link  $a$  appears as an external input to the environment when: it is produced locally (in the current model view) and is consumed by an external node (which is non-comparable to all nodes in the model view).
- (ii) Link  $a$  appears as an external output from the environment when: it is consumed locally (in the current model view) and is produced by an external node (which is non-comparable to all nodes in the model view).

With these rules, external flows are added whilst context free node interfaces are retained, though the additional external flows are model view dependent. The formal description in Chapter four covers only the context free portion of the model, and does not discuss external flows.

The second limitation: the occurrence of apparent half flows in some model views, is due to the generalisation of models to include node ordered sets. When the node order is not a tree, a situation can arise where a model is complete, so all links have producer

and consumer nodes, but in some model views there are still half flows. An example of this is shown in figure 3.14. This is a consequence of the overall requirement for node input and output interfaces to be context independent, that is for a node's interface to be the same regardless of what model view it is in.

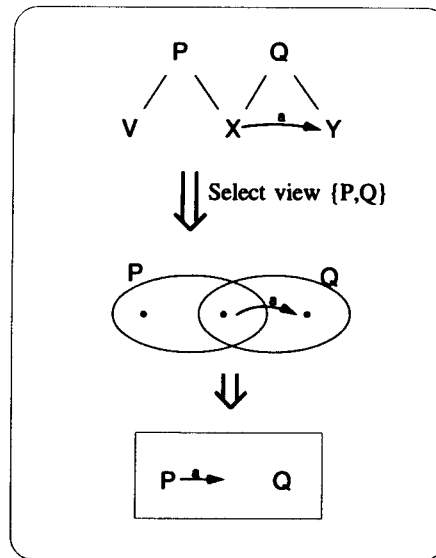


Figure 3.14 A model which contains apparent half flows

The problem is: when the user looks at the model view  $\{P, Q\}$  it appears in figure 3.14 that  $P$  produces link  $a$  and  $Q$  does nothing. Though there is a consumer of link  $a$  within  $Q$  there is no leaf producer of link  $a$  outside  $Q$ , so  $Q$  contains a consumer of link  $a$  (is a gross consumer of  $a$ ) but not a net consumer. If the model view is only  $\{Q\}$  then it is natural for no input of link  $a$  to be shown, but for the model view of  $\{P, Q\}$  we would like to see link  $a$  input into  $Q$ . Clearly doing this would violate the property of node interfaces being context free.

This limitation can be contained by the introduction of virtual half flows. A virtual half flow converts the apparent half flow to a full flow. To distinguish them, virtual flows are shown with a dotted line, as in figure 3.15.

In general a virtual flow for link  $a$  is added to a node  $Q$  when:

- (i)  $Q$  appears in a model view where link  $a$  is a net output of  $P$ , which is also in the model view and  $Q$  is a gross but not net consumer of link  $a$ .
- (ii)  $Q$  appears in a model view where link  $a$  is a net input of  $P$ , which is also in the model view and  $Q$  is a gross but not net producer of link  $a$ .

In situation (i) a virtual consumer is added, in situation (ii) a virtual producer is added. Again, these virtual flows are context dependent, and the formal model in Chapter four, which describes only context free interfaces, will not include virtual flows.

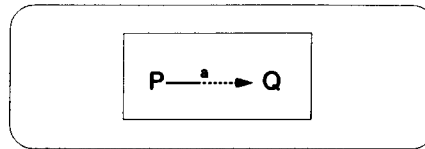


Figure 3.15 A model which contains a virtual flow

The third limitation: the inability of some links to appear in any model view, is due to the generalisation of models to include link ordered sets. When models are restricted to link trees, it is possible for every link to appear as a flow in some model view. However, when the link order is not a tree the situation can arise where, given a link order there are some links that could never appear in any model view of any model. An example of this is shown in figure 3.16. In case (i) the abstract link  $a$  is added, so link  $b$  no longer appears in the model view  $\{P, Q\}$ . The limitation is: regardless of changes to the underlying network, link  $b$  can never appear in a model view as link  $a$  is the maximum summary of link  $b$ . In case (ii) the interface between  $P$  and  $Q$  in the model view  $\{P, Q\}$  is links  $\{a, c\}$ . The limitation is that, regardless of changes to the underlying network, link  $a$  can never appear in a model view by itself, it will always appear with link  $c$ , as links  $\{a, c\}$  is the maximum summary of links  $\{e, f, g\}$ .

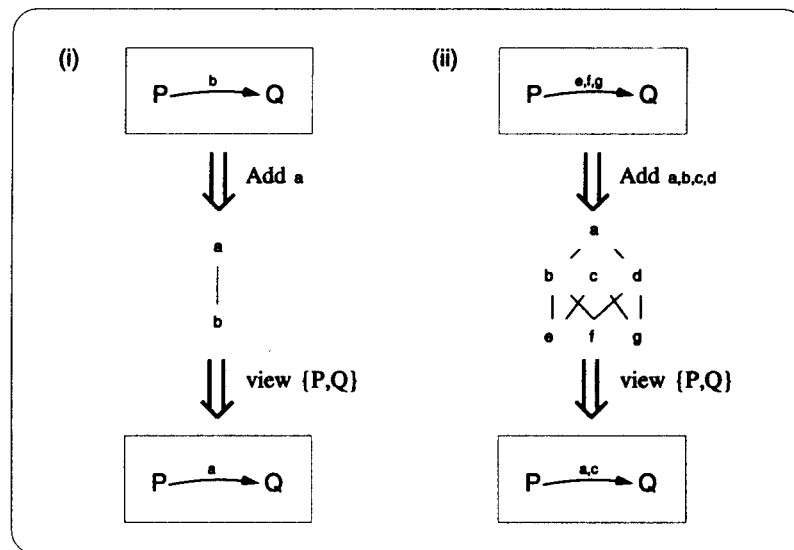


Figure 3.16 Two models which contain unviewable links

This third limitation is a consequence of node input and output interfaces in model views being maximum interfaces. Given an arbitrary link order, there may be some links which can not be part of any maximum link view or cannot appear alone. This limitation can only be overcome with the introduction of a model which allows non-maximum interfaces.

### **3.6 Summary**

This chapter has described with examples what complete and incomplete models are, and demonstrated model browsing and editing. The aim was to establish models as a structure which abstracts an underlying network, via node and link orders. This was further extended to allow incomplete models. Also, the intricacies caused by allowing non-tree node and link orders, and by allowing incomplete models have been covered. The following two chapters will provide the formal treatment of structured graphs.

---

## Ordered Sets : Background

---

This chapter introduces definitions and notation for ordered sets, required by the next chapter where structured graphs are formally defined. We have used the books by Davey and Priestley (1990) and Carre (1979) as our references.

### 4.1 Standard Terminology

**Definition 4.1** An *ordered set* (also referred to as *partially ordered set* ) is a set  $P$  equipped with a binary relation  $\leq$  on  $P$  such that, for all  $x, y, z \in P$ :

- $x \leq x$  (reflexive)
- $x \leq y$  and  $y \leq x$  imply  $x = y$  (anti-symmetric)
- $x \leq y$  and  $y \leq z$  imply  $x \leq z$  (transitive)  $\square$

**Definition 4.2** A order relation  $\leq$  on  $P$  gives rise to relations,  $<$  of *strict inequality* and  $\parallel$  of *non-comparability* such that, for all  $x, y \in P$ :

- $x < y$  if and only if  $x \leq y$  and  $x \neq y$
- $x \parallel y$  if and only if  $x \not\leq y$  and  $y \not\leq x$   $\square$

**Definition 4.3** Let  $\langle P, \leq \rangle$  be an ordered set and let  $x, y \in P$ . Then  $x < y$ , read as  $x$  is *covered by*  $y$  is given by:

$$x < y \text{ iff } x < y \text{ and } x \leq z < y \text{ implies } z = x \text{ for all } z \in P \quad \square$$

Finite orders can be diagrammatically represented as covering relations. These are called Hasse diagrams. If  $x$  is *covered by*  $y$  then  $x$  is placed below  $y$  in the diagram, and  $x$  and  $y$  are joined by a line. The node and link orders shown in figures 2.2 and 2.3 are Hasse diagrams. Once we are working with the covering relation we can use the terms parent and child so that,  $x$  is covered by  $y$  can be restated as  $x$  is a child of  $y$ . For finite ordered sets the reflexive transitive closure of the covering relation gives back the original ordering relation.

**Definition 4.4** Let  $P$  be an ordered set and let  $e \in P$  and  $S \subseteq P$ . Expressions  $\downarrow e$ ,  $\uparrow e$ ,  $\downarrow\downarrow e$ ,  $\uparrow\uparrow e$  are respectively the *down set* of  $e$ , the *strict down set* of  $e$ , the *up set* of  $e$  and

the *strict up set* of  $e$ .

- $\downarrow e = \{ x : P \mid x \leq e \}$
- $\updownarrow e = \{ x : P \mid x < e \}$
- $\uparrow e = \{ x : P \mid e \leq x \}$
- $\uparrow\downarrow e = \{ x : P \mid e < x \}$

Expressions  $\downarrow S$ ,  $\updownarrow S$ ,  $\uparrow S$ , and  $\uparrow\downarrow S$  are the set versions obtained by distributed union of the pointwise operators over  $S$ . □

The strict down set of  $e$  can be read as all *descendants* of  $e$ , while the strict upset of  $e$  can be read as all *ancestors* of  $e$ .

Maximals are those elements at the top of an order, while minimals are those at the bottom of an order. Note that maximals and minimals may contain more than one element. Often we are interested in the local maximals and minimals of a subset of an ordered set.

**Definition 4.5** Let  $P$  be an ordered set and let  $S \subseteq P$ . Then *maximals*, *minimals*, *upper bounds* ( $S^u$ ), *lower bounds* ( $S^l$ ), *upper-lower bounds* ( $S^{ul}$ ) and *span* are subsets of  $S$  which satisfy:

- *maximals*  $S = \{ m : S \mid \updownarrow m \cap S = \emptyset \}$
- *minimals*  $S = \{ m : S \mid \downarrow m \cap S = \emptyset \}$
- $S^u = \{ m : P \mid (\forall s \in S) s \leq m \}$
- $S^l = \{ m : P \mid (\forall s \in S) s \geq m \}$
- $S^{ul} = S^u \cup S^l$
- *span*  $S = \text{minimals } \downarrow S$  □

The *span* of a subset  $S$  is the set of minimal elements in  $P$  below  $S$ .

Because we use the functions *maximals* and *minimals* frequently, we adopt a more compact notation:

- $\overline{S} = \text{maximals } S$
- $\underline{S} = \text{minimals } S$

**Definition 4.6** Let  $P$  be an ordered set and  $S \subseteq P$ . The predicates *chain*, *flat*, *leaf*, *siblings* and *convex* are given by:

- $chain\ S = \forall x, y : S \bullet x \leq y \text{ or } x \geq y$
- $flat\ S = \forall x, y : S \bullet x \neq y \Rightarrow x \parallel y$
- $leaf\ S = S \subseteq \mathcal{P}$
- $siblings\ S = \exists p : P \bullet \forall x : S \bullet x \prec p$
- $convex\ S = \forall x, y : S, z : P \bullet x < z < y \Rightarrow z \in S$  □

A chain is a linear suborder. A subset of  $P$  is flat when it is an anti-chain (all elements are non-comparable). A subset is leaf when all subset elements are minimal of  $P$ . A subset satisfies the siblings predicate when they have a common parent element. A subset is convex when there are no order ‘gaps’ in the subset.

**Definition 4.7** Let  $P$  be an ordered set and  $S \subseteq P$ . The convex closure of  $S$  in  $P$  is given by:

$$\uparrow S = \uparrow S \cap \downarrow S \quad \square$$

The convex closure of a set, adds all elements between the highest and lowest elements, to the set.

Some of the functions defined above are closure operators. In fact, they form complementary pairs of the form: compact (C) and abstract (A), giving different representations of the same ordered set. The compact operator reduces the size of the set, while the abstract operator increases the size. They are complementary in that, C reverses the effect of A, and vice versa. The complementary operators are:

- (i)  $A = \uparrow$   
 $C = \_$  (minimals)
- (ii)  $A = \downarrow$   
 $C = \bar{\_}$  (maximals)
- (iii)  $A = \uparrow$   
 $C = \bar{\_}$  (minimals union maximals)

Case (i) holds for ordered sets with no infinite descending chain, (ii) hold for ordered sets with no infinite ascending chain, and (iii) hold for ordered sets with no infinite

chain. These properties provide the underpinning, for the definitions used in the next chapter.

### 4.2 View Orders

Now views can be formally introduced. The terminology of this subsection is not standard.

**Definition 4.8** Let  $P$  be an ordered set and  $V \subseteq P$ . Subset  $V$  is a *view* of  $P$  when the predicate *flat*  $V$  is true. Let  $V, W$  be views of  $P$ . The view order  $\leq_v$  on views of  $P$  is given by:

$$V \leq_v W \text{ if and only if } \downarrow V \subseteq \downarrow W \quad \square$$

Furthermore, views and downsets are equivalent, as each view is the set of maximals of some downset. So views can be reasoned about using either representation. The set of all views (equivalently downsets) forms a complete lattice. A view  $V$  of  $P$  is *complete* when  $\text{span } V$  equals  $\text{span } P$ . Several orders with their complete view orders are shown in figure 4.1.

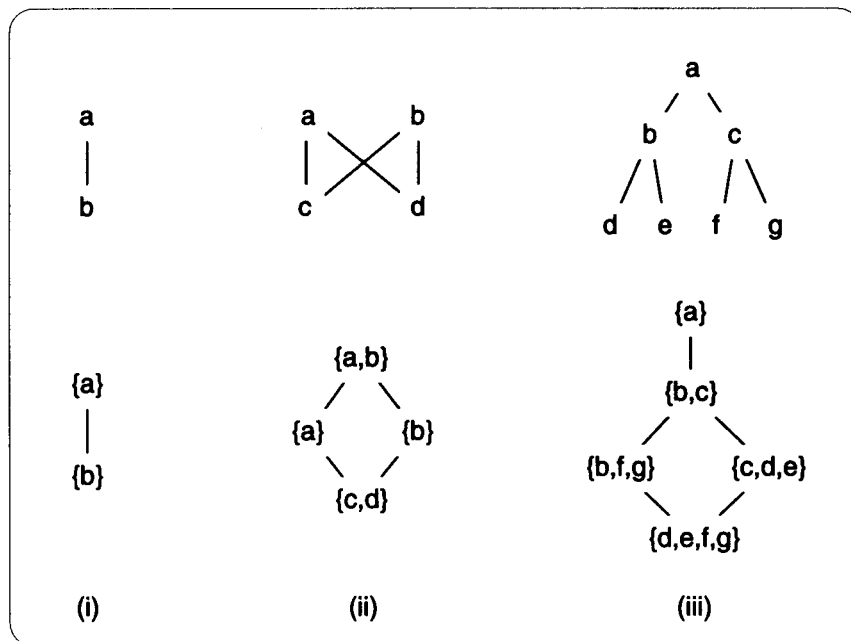


Figure 4.1: Three orders with their view orders

Orders (i) and (iii) have the maximum view  $\{a\}$ , while order (ii) has the maximum view  $\{a,b\}$  which contains two elements.

**Theorem 4.1** Let  $P$  be a finite ordered set. Let  $V$  be the set of all views on  $P$ . The function  $\text{span}$  is a lattice homomorphism from  $V$  to  $V$ . It partitions the set of all views



into equivalence classes (views having the same span), where each equivalence class has a maximum element (view).  $\square$

**Proof.** Appendix B Section B.1.

**Definition 4.9** Denote the function from views on  $P$  to the maximum element of their equivalence class as  $max_v$ .  $\square$

The function  $span$  factors the ordering on views into the ordering between views with the same span, and the ordering between the span equivalence classes. Now given any finite poset  $P$ , the map from each span of  $P$  to maximum spanning views can be constructed as follows:

**Proposition 4.1** Let  $P$  be an ordered set and  $S \subseteq P$ . The *maximum view* over  $S$  is called  $max_v S$ , and given by:

$$max_v S = \overline{\{ x : \uparrow S \mid span\ x \subseteq span\ S \}} \quad \square$$

In the next chapter some limitations in taking views of a model are discussed. The following definitions are required to support this discussion.

**Definition 4.10** Let  $P$  be an ordered set then  $P$  is an *inclusion order* if and only if it satisfies:

$$\begin{aligned} x \leq y &\equiv \downarrow x \subseteq \downarrow y \\ &\equiv span\ x \subseteq span\ y \text{ for all } x, y \in P. \end{aligned} \quad \square$$

So an inclusion order can be represented as a family of sets ordered by set inclusion. Note that:

$$x \leq y \equiv \downarrow x \subseteq \downarrow y$$

holds in an arbitrary ordered set. In this case the complete down sets are required for the order isomorphism to work, whereas for an inclusion order the minimal views are sufficient to characterise the order.

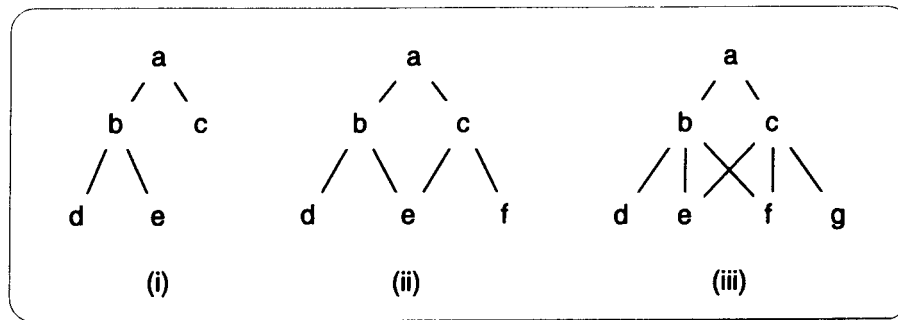


Figure 4.2: Three inclusion orders

Example inclusion orders are shown in figure 4.2. Order (i) and (ii) are semi-lattices while (iii) is not, as  $e$  and  $f$  have no least upper bound.

A chain is not an inclusion order, as all elements in a chain have the same span. In general, an order will be a non-inclusion order when: two elements have the same span, or there are two non-comparable elements whose leaf views are ordered. Some non-inclusion orders are shown in Figure 4.3.

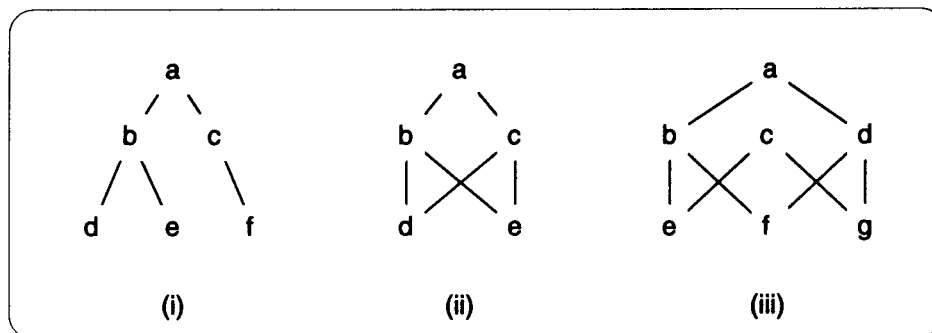


Figure 4.3: Three non-inclusion orders

Orders (i) and (ii) are non-inclusion orders because, in (i) both  $c$  and  $f$  have the same span:  $\{f\}$ , and in (ii)  $a$ ,  $b$  and  $c$  all have the same span:  $\{d, e\}$ . Order (iii) is a non-inclusion order because  $a$  and  $c$  are non-comparable and  $\text{span } c$  is a subset of  $\text{span } a$ . The maximum view over span  $\{e, f, g\}$  in order (iii) is  $\{a, c\}$ , not  $\{a\}$ . Lets look at the relationship between maximum views and inclusion orders.

**Proposition 4.2** Let  $P$  be an ordered set. Every singleton set of  $P$  is a maximum spanning view if and only if  $P$  is an inclusion order. □

**Definition 4.11** Let  $P$  be an ordered set,  $P$  is a *semi-inclusion order* if and only if every element of  $P$  appears in a maximum view. □

The following proposition provides a constructive definition of semi-inclusion orders.

**Proposition 4.3** Let  $P$  be an ordered set. Then  $P$  is an *semi-inclusion order* if and

only if it satisfies:

$$\text{span } x = \text{span } y \Rightarrow x = y \quad \text{for all } x, y \in P. \quad \square$$

In a semi-inclusion order every element has a unique span, but some elements; whose span is comparable, may not be comparable in the order. Semi-inclusion orders are a weaker form of inclusion order, so inclusion orders are semi-inclusion orders but some non-inclusion orders are semi-inclusion orders. Semi-inclusion orders do not contain chains or chain like structures. An example was shown in figure 4.3 (iii), where every element has a unique span but element  $a$  which has a span greater than element  $c$ 's span, does not cover  $c$ . Orders (i) and (ii) are not semi-inclusion orders because of the chain or chain like structure they contain.

Proposition 4.2 indicates when each element in an order can appear by itself in a maximum view. While definition 4.11 indicates just when each element can appear in a maximum view, possibly never by itself. This classification of orders will be important when we use leaf views as a representation for maximum views in the next chapter.

---

---

## Structured Graph Formalism

---

Formal representations for models are important. In Chapter two, browsing and editing a structured analysis model was demonstrated. A user was able to see the node order, the link order and select any arbitrary cross-section of a model as a graph, then directly edit it. From a user perspective, the model comprises the node and link orders and all possible cross-section networks of the model. Clearly trying to define editing operations on this highly redundant structure would be complex. A representation of this structure which contains no redundant information is needed to make the definition of editing operations straightforward and allow a better understanding of how viewing works; such a canonical representation assists visual formalisms to be simple and understandable.

This chapter presents the definition of a structured graph and two important representations. The representation which contains no redundant information is the *compact model*, and the representation which contains all derivable information is the *abstract model*. A tool user views cross-sections of abstract models, *model views*, which represent summary networks. The relationship between these representations and user browsing and editing is shown in figure 5.1.

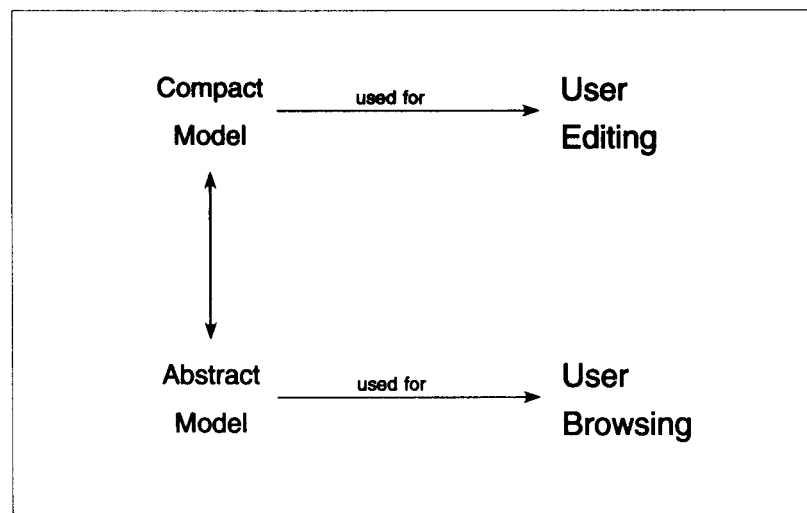


Figure 5.1 Compact and abstract models

Section 5.1 defines structured graphs (models). Section 5.2 presents the two canonical representations of structured graphs: compact and abstract models.

### 5.1 Structured Graphs

A structured graph (model) is a bi-partite graph whose vertices are nodes and links with edges characterised by producer and consumer relations. The structure is provided by relations on nodes and links.

**Definition 5.1** Given two sets Node and Link, a *structured graph* is a tuple  $(\leq_l, \leq_n, prods, cons)$  where:

- $\leq_l$  : is a finite ordering relation on *links* :  $\wp$  Link
- $\leq_n$  : is a finite ordering relation on *nodes* :  $\wp$  Node
- $prods, cons$  : Link  $\rightarrow \wp$  Node

An alternative representation is  $(\leq_l, \leq_n, out, in)$  where:

- $out, in$  : Node  $\rightarrow \wp$  Link
- $out\ n = \{ l \mid n \in prods\ l \}$
- $in\ n = \{ l \mid n \in cons\ l \}$

□

### 5.2 Compact and Abstract Models

We present a series of definitions which culminate in the definition of the *compact* and *abstract* functions  $C$  and  $A$ , which map structured graphs to their compact and abstract forms. Figure 5.2 depicts their relationship. We will prove that this diagram commutes (in appendix B).

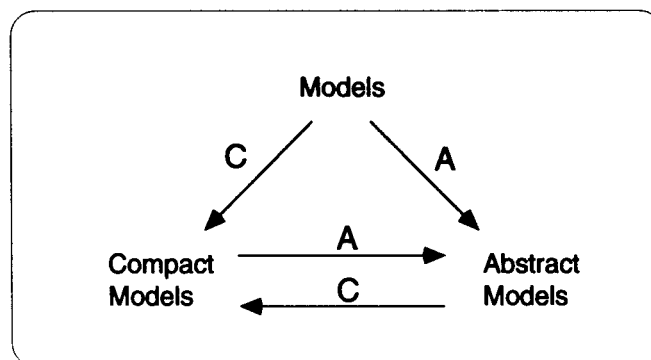


Figure 5.2 The mappings between models, compact models and abstract models

The first definition takes care of equivalent models which may have different levels of link summarisation.

#### Definition 5.2

- a) Conversion to an equivalent model with leaf links only.

$$C_l (\leq_l, \leq_n, out, in) = (\leq_l, \leq_n, span \circ out, span \circ in)$$

b) Conversion to a model with all possible links.

$$A_l (\leq_l, \leq_n, out, in) = (\leq_l, \leq_n, \downarrow \circ max_v \circ out, \downarrow \circ max_v \circ in) \quad \square$$

The next definition provides the crux of our approach. The key is to be able to capture the ideas of network abstraction presented in the previous section: the essential thing here is to capture the way in which full flows can be derived from an underlying network. We need to cater for cases when flows are not fully distributed, and when they are not complete (i.e. missing producers or consumers). Our presentation is simplified by restricting these preliminary definitions to models with leaf links only. The mappings in Definition 5.2 were provided to allow us to use such leaf link models as an intermediate stage in the construction of  $C$  and  $A$ .

For an underlying network we think of a leaf link as participating in a (full) flow just when it has non-comparable leaf producers and leaf consumers. We can generalise this to allow for the case when flows are not fully distributed, by replacing “leaf producers” by “minimal producers”, and similarly for consumers. A first attempt at defining an abstraction of a full flow might be to lift the producer and consumer relations through the node order, while ensuring that the flows still correspond to the flows in the underlying network. This would lead to the following definition for the set of full producers of a given leaf link  $l$ : links

$$prods_f l = \{ n : nodes \mid \exists p : \underline{prods} l, c : \underline{cons} l \bullet \begin{array}{l} p \parallel c \\ n \geq p \\ n \parallel c \end{array} \}$$

and similarly for  $cons_f$ . The minimal consumer  $c$  in this definition acts as the basis for the abstraction of the full flow for  $l$  from  $p$ . Our actual definition is very close to this, and is equivalent to it for complete models. However, for incomplete models, there is a desirable convexity property of full flows, which this first attempt at a definition does not yield. We relax the requirement that a full flow producer be connected to the *same* minimal consumer as the minimal producer which it abstracts.

$$prods_f l = \{ n : nodes \mid \exists p : \underline{prods} l, c1, c2 : \underline{cons} l \bullet \begin{array}{l} p \parallel c1 \\ n \geq p \\ n \parallel c2 \end{array} \}$$

The algebraic definition below is equivalent.

### Definition 5.3 Full Flows

Full flow producer for leaf links:  $prods_f: \text{Link} \rightarrow \wp \text{Node}$

For given  $prods$  and  $cons$ , and  $l : \underline{links}$

$$prods_f l = \uparrow(\underline{prods} l - (\underline{cons} l)^{ul}) - (\underline{cons} l)^{ul}$$

Full flow consumer for leaf links:  $cons_f : \text{Link} \rightarrow \wp \text{ Node}$

is defined analogously. □

The abstract form of a structured graph is constructed as a series of transformations: restricting the model to leaf links only (using  $C_l$ ), filling in any missing flows for those links, then for each node including all possible links (using  $A_l$ ). This is formalised below.

**Definition 5.4 Abstract models**

The *abstraction mapping*  $A$  for structured graphs is defined as

$$A = A_l \circ A_n \circ C_l$$

where

$$A_n (\leq_l, \leq_n, prods, cons) = (\leq_l, \leq_n, prods_a, cons_a)$$

$$\forall l : \text{links} \bullet$$

$$prods_a l = \downarrow (prods l \cup prods_{fl})$$

$$cons_a l = \downarrow (cons l \cup cons_{fl})$$

□

The set of producers in the abstract model is formed as the convex closure of the full producers together with any other producers in the original model.

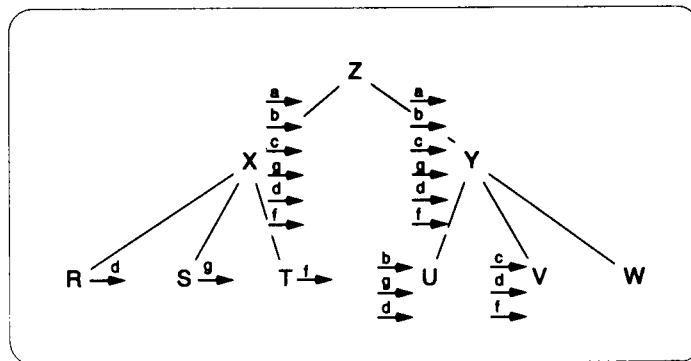


Figure 5.3 The abstract form of the model shown in figure 3.7

The model shown in Figure 5.3 is the abstract form of that in Figure 3.7. Note that each node interface is a downset of links, rather than just the most summarised links. Also, node Z hides link  $a$  and its descendants, because their producers and consumers are contained within  $Z$ . An abstract model producer which is not a full flow producer is a half flow producer, and similarly for consumers.

The compact form of a structured graph is constructed by restricting the model to leaf

links only (using  $C_l$ ), then extracting minimal producers and consumers, and maximal half flow producers and consumers.

### Definition 5.5 Compact models

The *compact mapping*  $C$  for structured graphs is defined as

$$C = C_n \circ C_l$$

where,

$$C_n (\leq_1, \leq_n, prods, cons) = (\leq_1, \leq_n, prods_c, cons_c)$$

$\forall l : links \bullet$

$$prods_c l = \underline{prods_a l} \cup (\overline{prods_a l} - prods_f l)$$

$$cons_c l = \underline{cons_a l} \cup (\overline{cons_a l} - cons_f l) \quad \square$$

The compact form is the smallest (leaf link) model which generates the same abstract model. The definition can be understood by relating it back to the abstract form: in the abstraction mapping, full flows are derived just from the underlying network (the minimals), but half flows are derived via a convex closure, which requires that maximal half flows be recorded in the compact form, as well as the minimals.

Examples of compact models are given by Figures 3.11 (i), (iii) and (iv); figure (ii) can be made compact by removing node  $U$  as a producer of link  $a$ .

### 5.3 Properties

**Theorem 5.1** Figure 5.2 is a commuting diagram, that is:

$$A \circ C = A \quad \text{and} \quad C \circ A = C \quad \square$$

**Corollary 5.1** The equivalence of compact and abstract models

$$A m_1 = A m_2 \Leftrightarrow C m_1 = C m_2 \quad \square$$

**Corollary 5.2** Closure

$$A \circ A = A \quad \text{and} \quad C \circ C = C \quad \square$$

Proofs are given in Appendix B. The equivalence property ensures that the two forms of model representation actually identify the same collection of models. The closure property ensures that further abstraction/compaction has no effect.

**Lemma 5.1** Convexity. An abstract model  $(\leq_1, \leq_n, prods, cons)$  satisfies:

$\forall l : links \bullet$

$$convex (prods l) \quad \text{and} \quad convex (cons l) \quad \square$$



For an example consider the model in figure 3.11 (ii) which is an abstract model with a trivial link order. If node  $U$  was removed as a producer of link  $a$  as suggested before, the model would not satisfy flow continuity. Nodes  $P$  and  $Z$  would produce link  $a$ , while  $U$  which is between  $P$  and  $Z$  would not.

#### 5.4 Model Editing

The compact model will be used to define editing operations because it is the smallest canonical representation, whilst the abstract model will be used for browsing. The editing operations given here are intended to be the basic building blocks for user editing operations, while the browsing operation given here provides the context free portion of a model view; the context dependent external and virtual flows would need to be added to form a model view for a user. Definitions for these now follow.

**Definition 5.6** Let  $\leq_x$  and  $\leq_y$  be order relations. Then  $\leq_x$  and  $\leq_y$  are *composable* if and only if:

$$(\leq_x \cup \leq_y)^* \text{ is an order relation} \quad \square$$

The above check is needed to avoid introducing cycles with the following edit operations.

**Definition 5.7** Let  $X$  and  $Y$  be compact models where their node and link orders are composable. Union, intersection and difference operations on them are given by:

$$X \cup Y = C((\leq_{xl} \cup \leq_{yl})^*, (\leq_{xn} \cup \leq_{yn})^*, \text{prods}_x \cup \text{prods}_y, \text{cons}_x \cup \text{cons}_y)$$

$$X \cap Y = C((\leq_{xl} \cup \leq_{yl})^*, (\leq_{xn} \cup \leq_{yn})^*, \text{prods}_x \cap \text{prods}_y, \text{cons}_x \cap \text{cons}_y)$$

$$X - Y = C((\leq_{xl} \cup \leq_{yl})^*, (\leq_{xn} \cup \leq_{yn})^*, \text{prods}_x - \text{prods}_y, \text{cons}_x - \text{cons}_y) \quad \square$$

The edit operations include a compact closure, to remove any producers or consumers that are now derivable. This ensures the result of these operations is always a compact model. However, in some situations this can lead to a lack of associativity, which we discuss in section 5.6.

#### 5.5 Model Viewing

A user browses a model by selecting model views. Both nodes and links in a model view can be restricted, with a final model view showing nodes and their net maximum summarised interfaces.

**Definition 5.8** Let  $m$  be an abstract model. Let  $N \subseteq \text{nodes}$ , and  $L \subseteq \text{links}$ , where  $N$  is flat and  $L$  is a downset. The model view of  $m$  restricted to  $L$  and  $N$  is depicted as:

$$\text{modelV } m \ L \ N = F ( L \triangleleft m \triangleright N )$$

where the abstract model node and link restriction operation on  $m$  is given by:

$$L \triangleleft m \triangleright N = A ( L \triangleleft \leq_1 \triangleright L, N \triangleleft \leq_n \triangleright N, L \triangleleft \text{prods} \triangleright N, L \triangleleft \text{cons} \triangleright N )$$

and the flat maximum model view operator  $F$  is given by:

$$F m = ( \leq_1, \leq_n, \text{max}_v \circ \text{out}, \text{max}_v \circ \text{in} ) \quad \square$$

Symbols  $\triangleleft$  and  $\triangleright$  are domain and range restriction operators.

Model views have context free interfaces under arbitrary node selection only with respect to a common link selection. For example, if model views are always selected with all links, the views will have context free interfaces.

## 5.6 Limitations

Some limitations in viewing and editing a model will now be described.

**Definition 5.9** Let  $L$  be a link order. Let  $M$  be the set of abstract models with link order  $L$ . Then  $L$  is *viewable* if and only if:

$$\forall l : L \bullet \exists m : M \bullet \exists n : \text{nodes}_m \bullet \\ l \in \text{max}_v (\text{out}_m n) \text{ or } l \in \text{max}_v (\text{in}_m n) \quad \square$$

That is, a link order is viewable when each link can appear as a flow in the flat interface of some node in some abstract model. If a link order is not viewable then there will be some links which can never appear in any model view of any model.

**Conjecture 5.1** Let  $L$  be a link order. Then  $L$  is viewable if and only if  $L$  is a semi-inclusion order. □

The major impact of this occurs when editing a model. Consider adding a flow  $l$  between two nodes which have no flows between them. Only if the link order is viewable will the user be guaranteed that this operation will result in the flow  $l$  appearing between the two nodes. Further, to guarantee the result is exactly the flow  $l$  with no extra flows, the link order must be an inclusion order.

Another limitation is that structured graph composition is not associative. This means that: if a user performs a collection of edit operations in a different order, the result may be different. This is undesirable. Figure 5.4 shows an example where associativity fails.

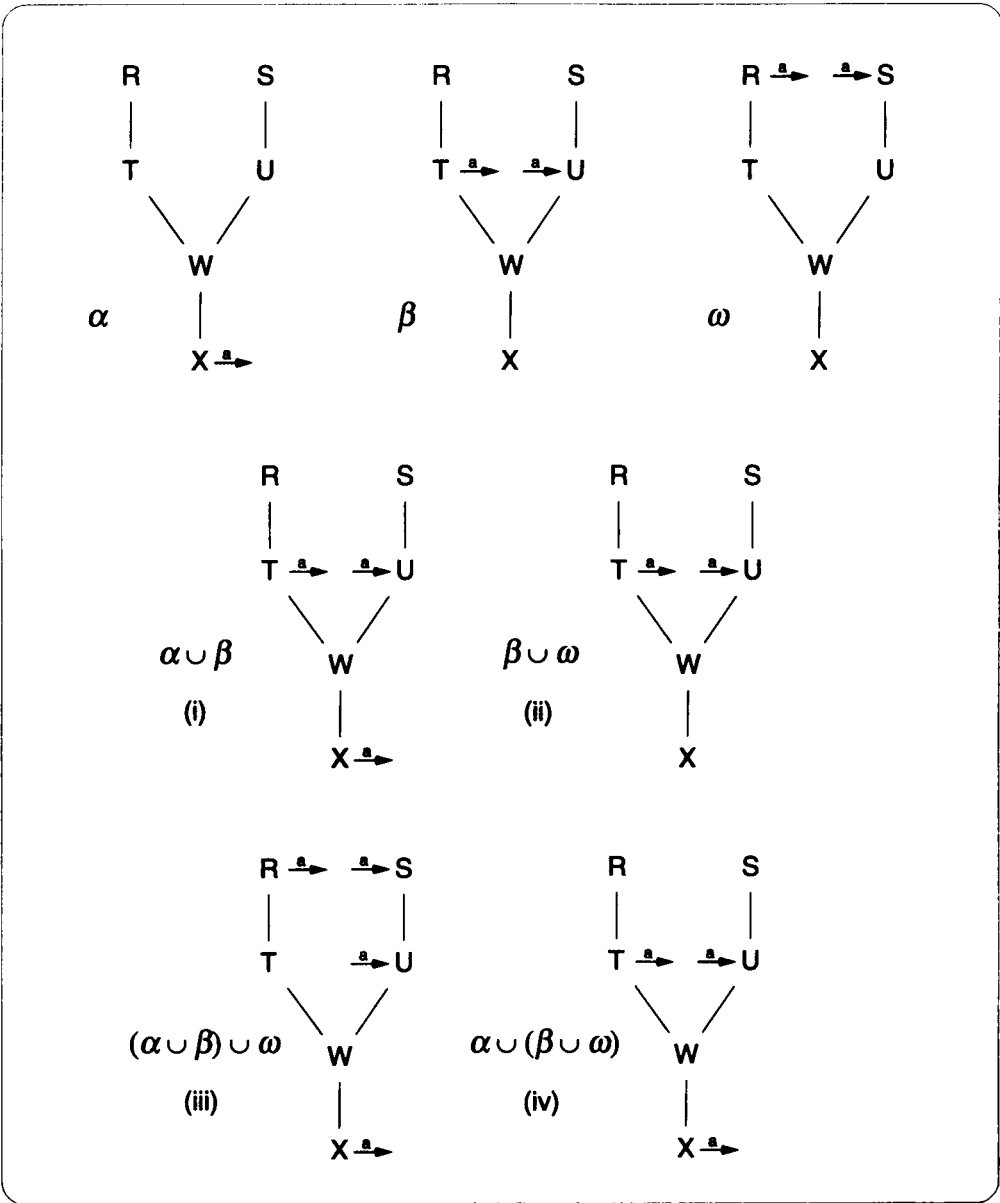


Figure 5.4 A demonstration of non-associative structured graph composition

The results of the composition of the three models are shown in (iii) and (iv), which are different. Note all models in the example are compact models, as the definition of composition includes a compact completion. The reason for associativity failing, is that composition does not preserve full flows; for example in (ii) there is a full flow between nodes T and U but in (iv) there are only half flows. The addition of model  $\alpha$  has collapsed the full flow.

Restricted families of structured graphs though, will still have associative composition. The situations where associativity fails, require an unlikely combination of half flows and node order. If we restrict, to structured graphs with no half flows, or to tree structured node hierarchies, for example, we recover associativity of composition.

## 5.7 Discussion

In summary, this chapter has presented two canonical representations of structured graphs: the compact and abstract models. These models are the fixpoints of two complementary closure operators, as shown in the commuting diagram of figure 5.2. This follows the same pattern noted in Chapter four, where complementary ordered set closure operators were identified.

The basis for browsing and editing operations has been given, and their properties and limitations identified. Two limitations were: (i) depending on the link order, some links may never appear in any flat model view, and (ii) model composition (union) is not associative. The latter limitation could be avoided by removing the compact closure step from the definition of model union (removing the  $C$ 's in definition 5.7), at the cost of leaving some derived flows in the model. Model browsing would not be affected, as this is done on the abstract completion of a model.

Proofs for the theorems are given in Appendix B. Implementations of structured graphs are given in Appendices C and D. Both these implementations are text based. The gofer code in Appendix C is a direct implementation of the compact and abstract closure operators, and the three editing operations: model union, intersection and difference. The C++ application introduced in Appendix D implements a collection of user level operations for editing and browsing such as: adding and deleting nodes and links, and displaying a table representation of selected model views.

A graphical implementation would be future work as it requires additional research to create suitable graph layout algorithms. A large body of work on graph layout, and on DFD layout already exists, but this is for single flat graphs. This is not suitable for structured graph viewing as a vast number of model views are possible on a single structured graph, and related model views (their nodes are related) should have a related layout to assist users to keep their perspective.

---

## Building with Structured Analysis Components

---

Part two of this thesis presents components and component composition. Components further improve the modularity of model construction. The editing operations discussed in part one, such as adding a flow between two processes are appropriate for an individual editing a model. Very large models are likely to be produced by teams. With existing structured analysis tools, team members operate within a global name space for processes and data flows, so team members must be carefully co-ordinated to ensure they don't use the same data flow names for different data flow instances. Further, team members' submodels are joined by merging together those processes and data flows which have exactly the same name. A less rigid system would improve the editing of such large models by a team, whilst still allowing submodels to be composed with one user operation. This chapter will demonstrate such a system.

An example of three components and their composition will be presented in this chapter. The example follows the theme used in part one; it is another portion of a jumbo simulator. Note that this example is an incomplete model. First, the three components are presented. Second, the composition of these three components is shown. Third, the new mechanism, *meshing*, which makes composition flexible is highlighted. Finally, a selected view of the composed model is shown.

### 6.1 Example Components

This section presents three components: `game`, `own_jumbo` and `other_plane`. Each component is shown in the same way as the model shown in Chapter two, that is as: the process order, a collection of DFDs, a data flow schema and the data flow order. A component is a structured graph which has been extended in two ways. Firstly, by allowing different links to have the same local name in different parts of a final model through the introduction of model qualifiers. Secondly, through the use of a link schema to define link types and subtypes, so actual links are then instances of these types.

Model qualifiers are used in component `game` shown in figure 6.1. Note every data flow ends with `[game]`. This is the model qualifier. For example `time[game]` can be read as the data flow `time` which appears in the submodel with root process `game`. Within the overall model there could be other data flows such as `time[own_jumbo]` which are distinct. Each component presented here uses its root node as the model qualifier for all its data flows. This means each component has its own data flow name

space, which gives greater naming freedom when each component is being developed by a different team member. In general, a data flow may have a set of model qualifiers. When all flows in a component have the same model qualifier (a simple component), the qualifiers can be abbreviated to []. This is adopted in later examples.

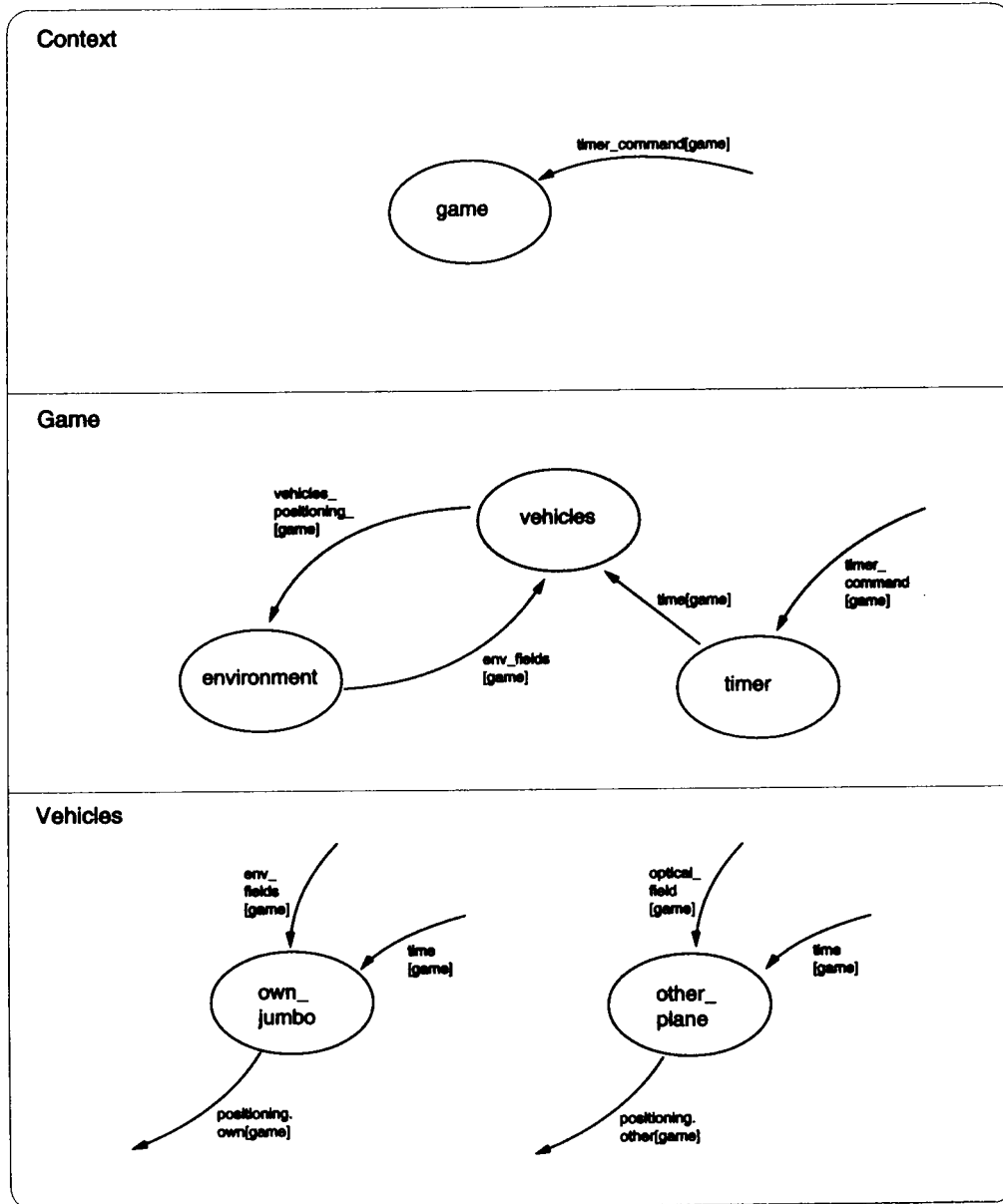


Figure 6.1 The DFDs of component game

The use of model qualifiers makes the time portion of `time[game]` a type scheme, as there may be multiple distinct data flows called `time[...]` but with different model qualifiers. However, within the one component we may also wish to have multiple flows with the same type. This is achieved by using *instance qualifiers*. An example is shown in the `vehicles` DFD in figure 6.1. There are two flows which both have the type `positioning`, which are: `positioning.own[game]` and `positioning.other[game]`. Both `own` and `other` are instance qualifiers. The process order is shown in figure 6.2.

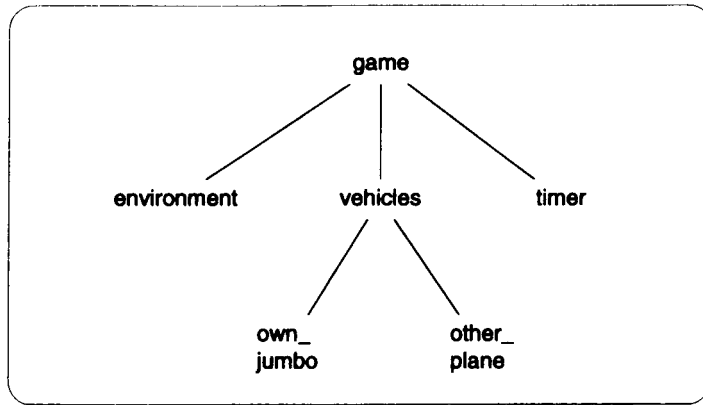


Figure 6.2 The process order of component game

Figure 6.3 shows the data flow type dictionary which is common to all three components presented in this chapter. It follows a similar format to the data dictionary shown in Chapter two, except for the use of instance qualifiers. The other additional feature is the notation (scope schema):

`positioning/ = position`

which states that the label `position` is shared within the scope of the type `positioning`.

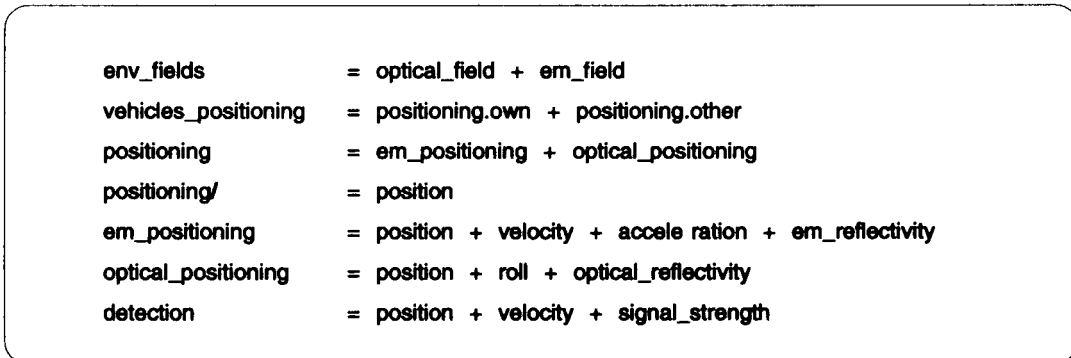


Figure 6.3 A data flow type dictionary

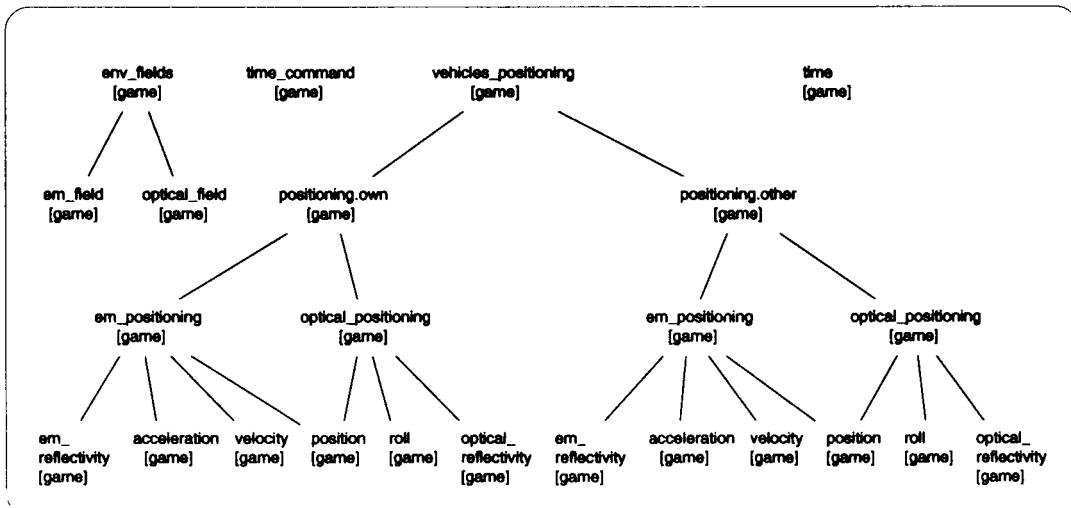


Figure 6.4 The data flow order of component game

The effect of scope is, within any data flow which has type positioning, there is only one occurrence of a data flow with the label position. Looking at figure 6.4 should make this clearer. Note that the data flows `em_positioning[game]` and `optical_positioning[game]` under `positioning.own[game]` share a common child: `position[game]`. It is the use of scope schema, which allows data flow hierarchies other than trees to be represented.

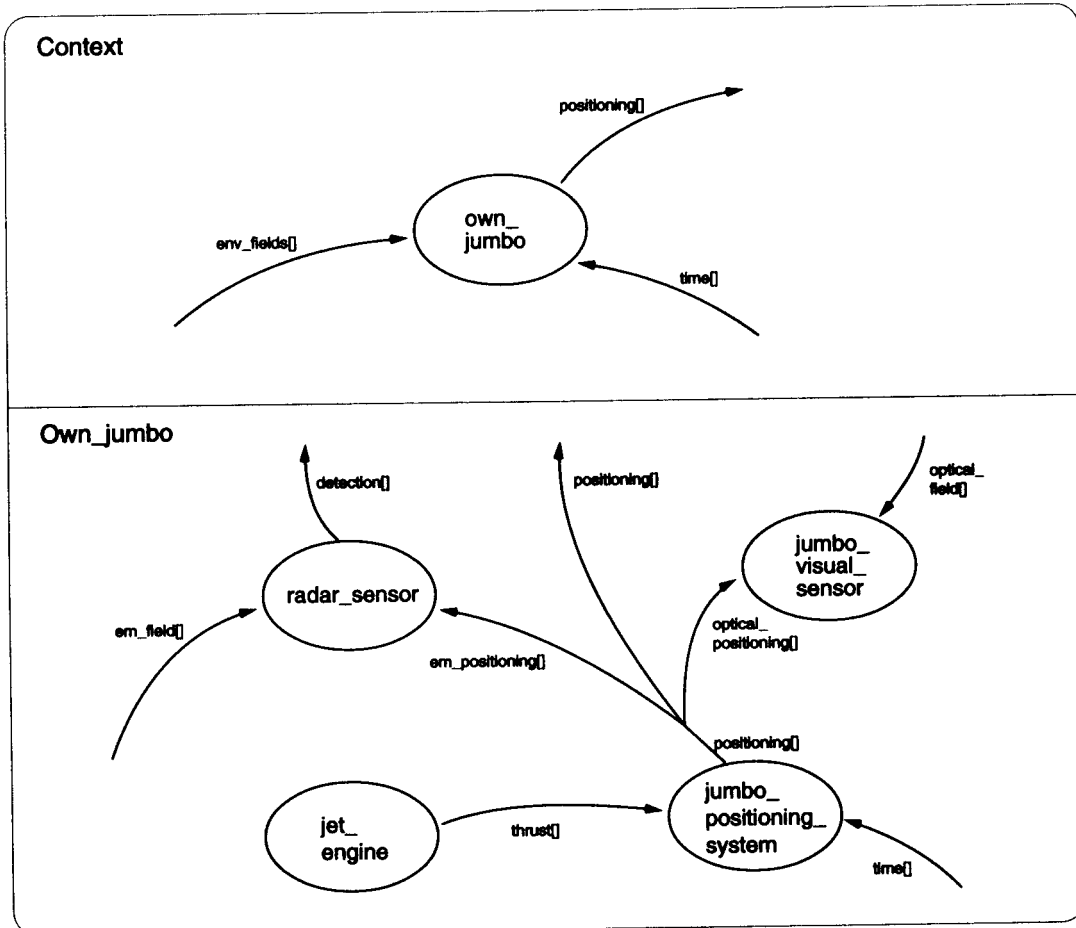


Figure 6.5 The DFDs of component `own_jumbo`

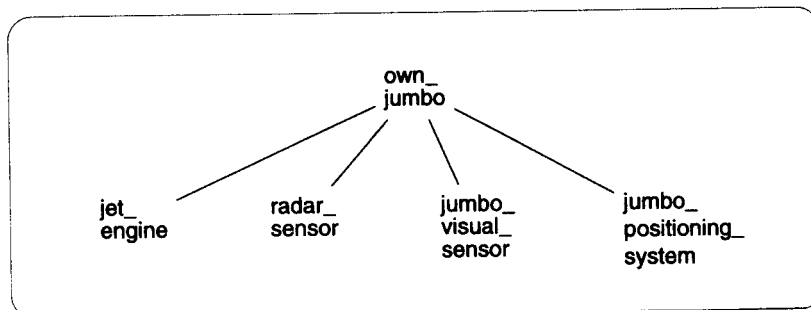


Figure 6.6 The node order of component `own_jumbo`

Figures 6.5 to 6.7 show the `own_jumbo` component, its collection of DFDs, process order and data flow order. In figure 6.5 an abbreviation is used for the model qualifier. Rather than placing `[own_jumbo]` at the end of every data flow in the `own_jumbo`



component, [] is used. This is just an abbreviation for the component-root-process model qualifier as stated earlier.

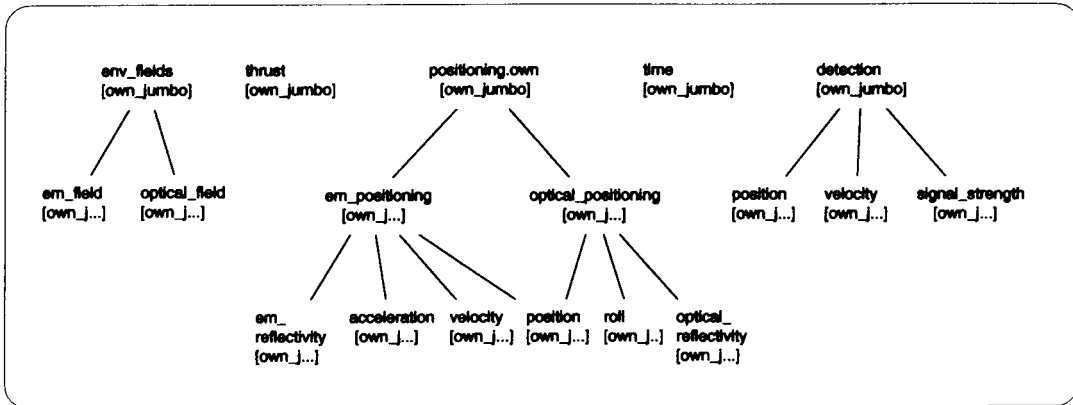


Figure 6.7 The data flow order of component own\_jumbo

A data flow type dictionary is not given here as all three components share the dictionary shown in figure 6.3. Figures 6.8 to 6.10 show the other\_plane component.

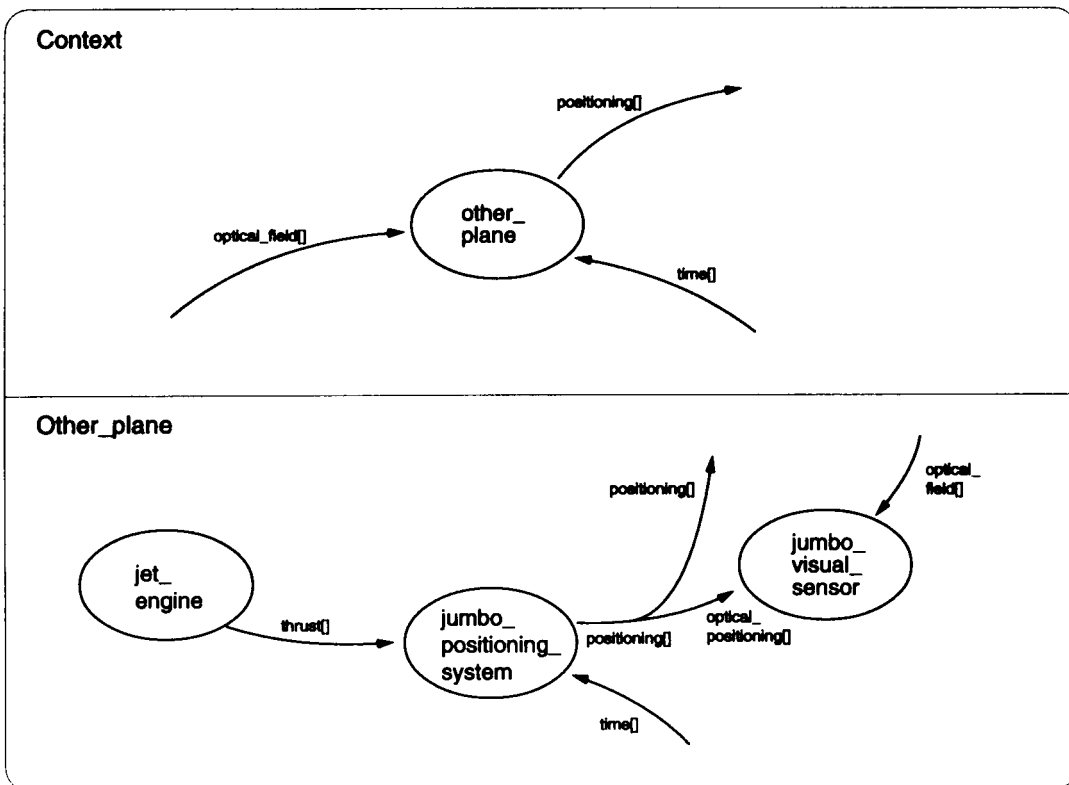


Figure 6.8 The DFDs of component other\_plane

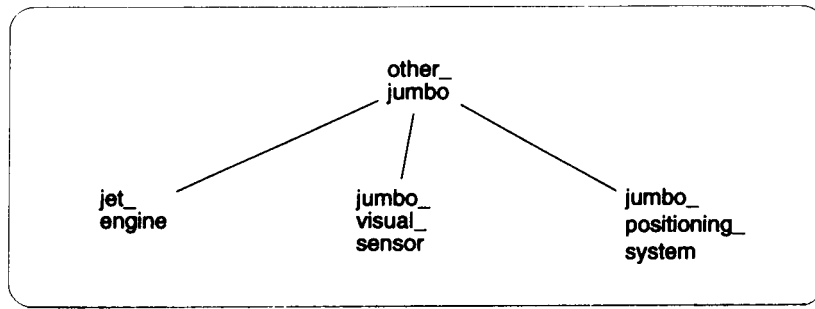


Figure 6.9 The node order of component other\_plane

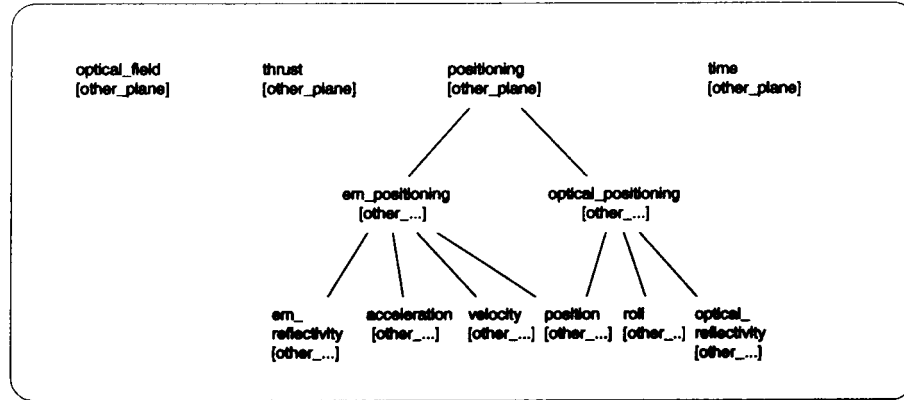


Figure 6.10 The data flow order of component other\_plane

## 6.2 The composed model

The composed model is shown in figures 6.11 to 6.13. The component process orders were combined by merging processes with the same process name together. The component data flow orders have been merged in a more flexible way. Data flows whose paths (producer nodes or consumer nodes) which intersect and satisfy a *meshability* relationship are mapped to the same data flow. This *meshability* relationship is discussed in more detail in the next section.

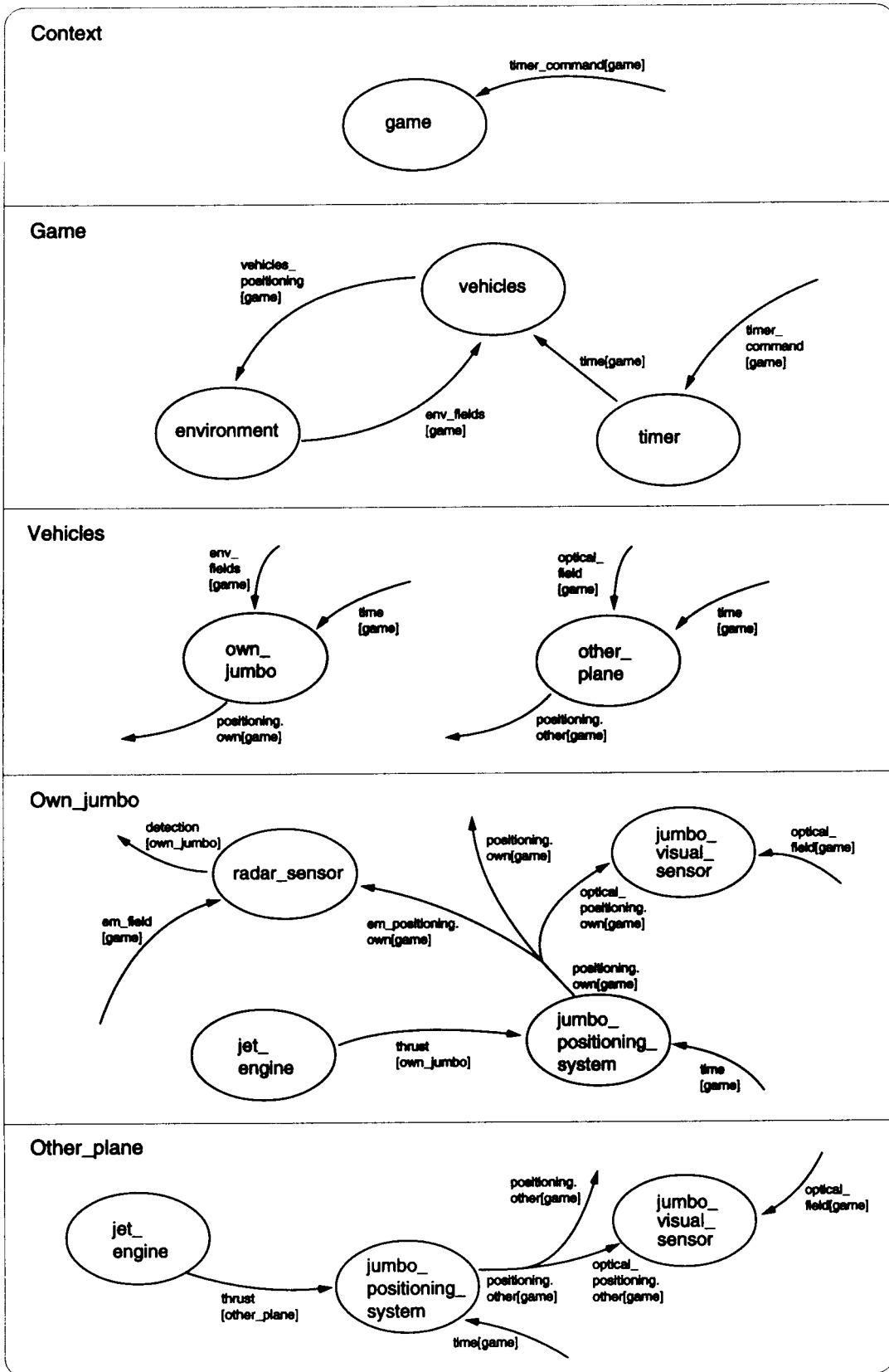


Figure 6.11 The DFDs of the composed component

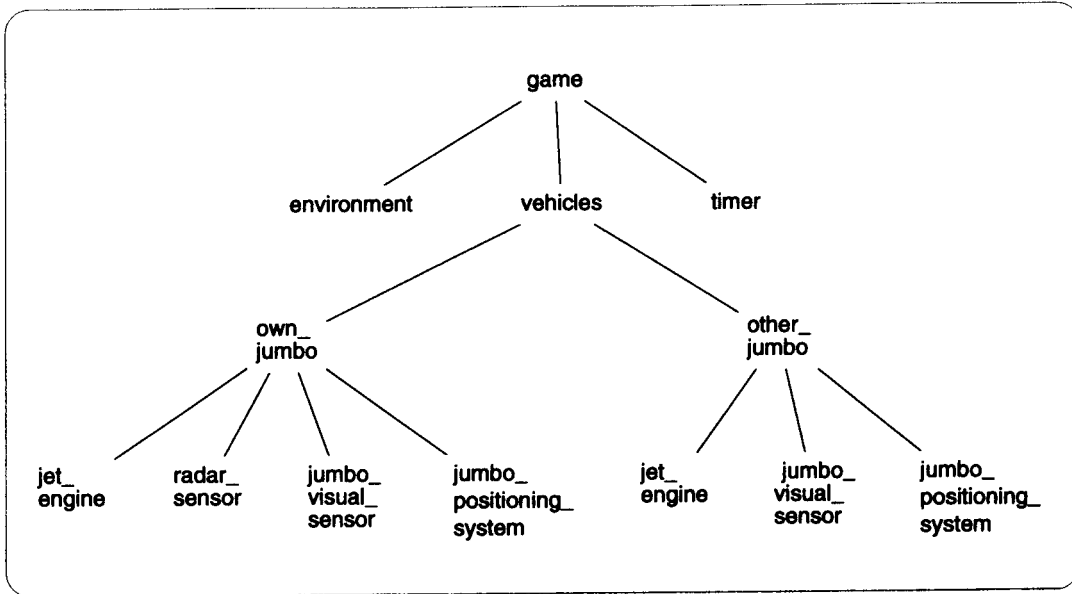


Figure 6.12 composed component node order

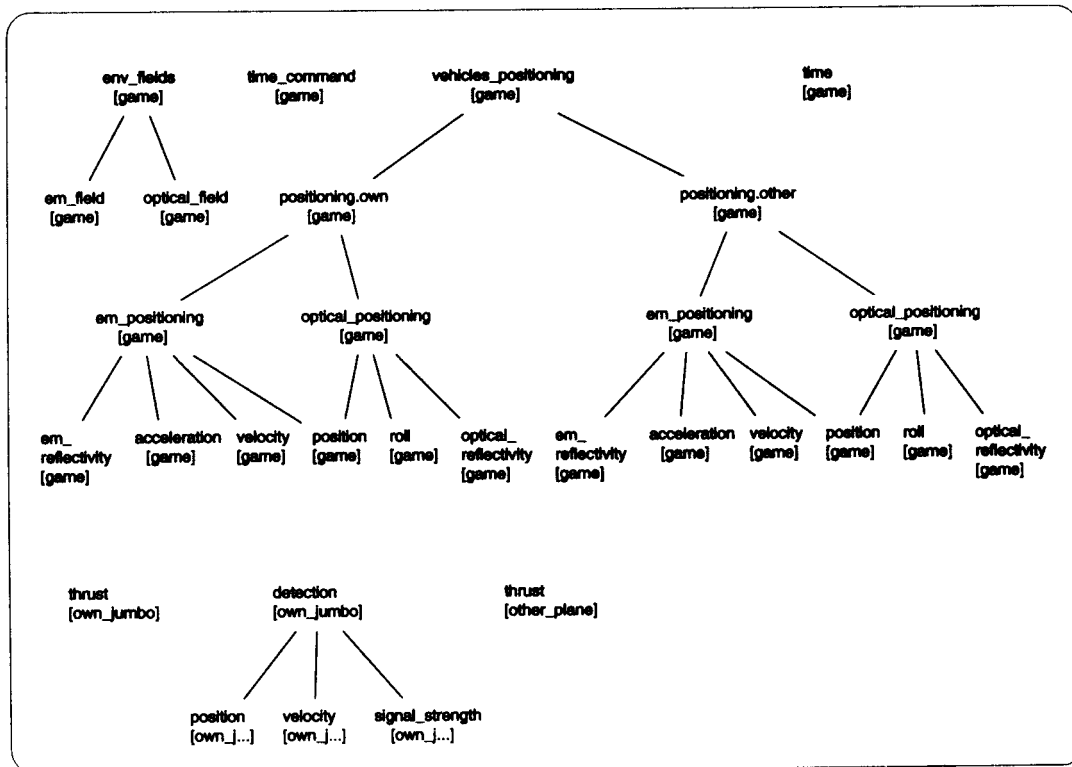


Figure 6.13 The data flow order of the composed component

### 6.3 Data flow meshing

The composition of the data flow orders, *meshing*, is shown in more detail in figures 6.14 and 6.15. Figure 6.14 shows two flows, one from the game component and one from the own\_jumbo component which are meshed together. Flows env\_fields [own\_jumbo] and env\_fields [game] become env\_fields [game] in the

composed component. Note that this affects not just the individual flow `env_fields[own_jumbo]` but its children `em_field[own_jumbo]` and `optical_field[own_jumbo]` as well.

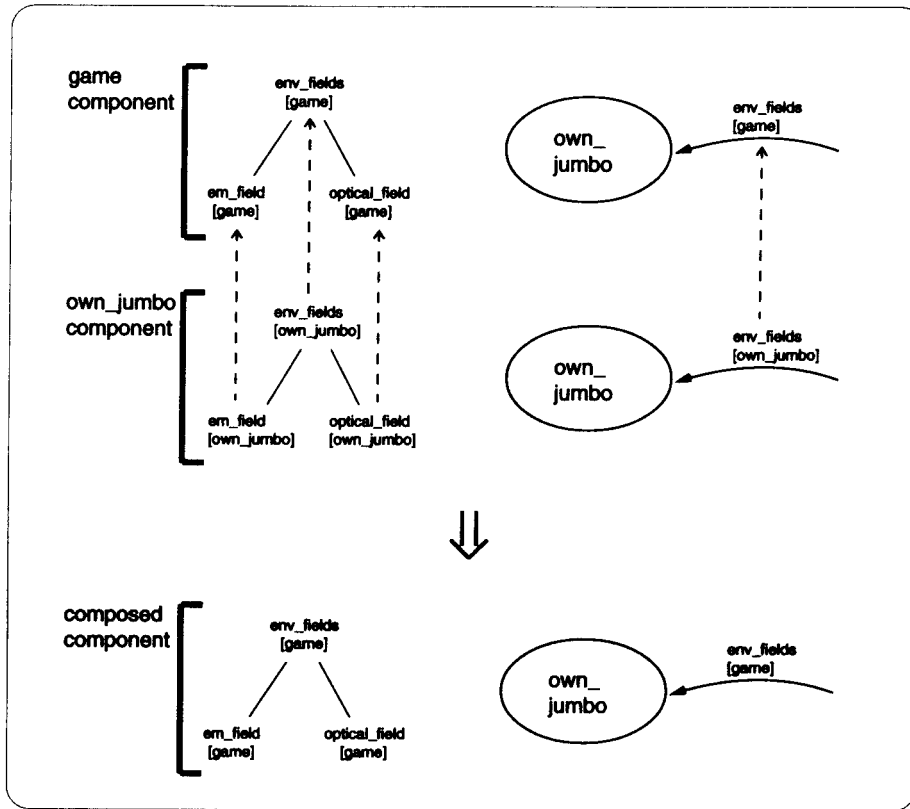


Figure 6.14 An example of meshing (I)

Figure 6.15 shows another example of meshing. Flow `positioning [own_jumbo]` is extended (i.e. its name) by `positioning.own [game]` and these flows have a common consumer, the process `own_jumbo`. As a result, `positioning [own_jumbo]` is replaced by `positioning.own [game]` within the `own_jumbo` submodel. The rough rule for identifying flow meshability is: a flow `x` meshes with another flow `y` when `x`'s name is a prefix of `y`'s name, and `x`'s model qualifier is less than `y`'s model qualifier.

In the preceding discussion minimal data flow names have been used. For example the full name of `positioning.own [game]` is `positioning.own.vehicles_positioning [game]`. The name `positioning.own [game]` was used, as it is smaller but still unique. The meshability rule given earlier refers to full names. The meshing shown in example 6.15 can be restated in terms of full names as: the flow `positioning [own_jumbo]` meshes with `positioning.own.vehicles_positioning [game]`; as `positioning` is a prefix of `positioning.own.vehicles_positioning` and `own_jumbo` is less than `game` in the process order. More details of meshing and component composition are given in later chapters.

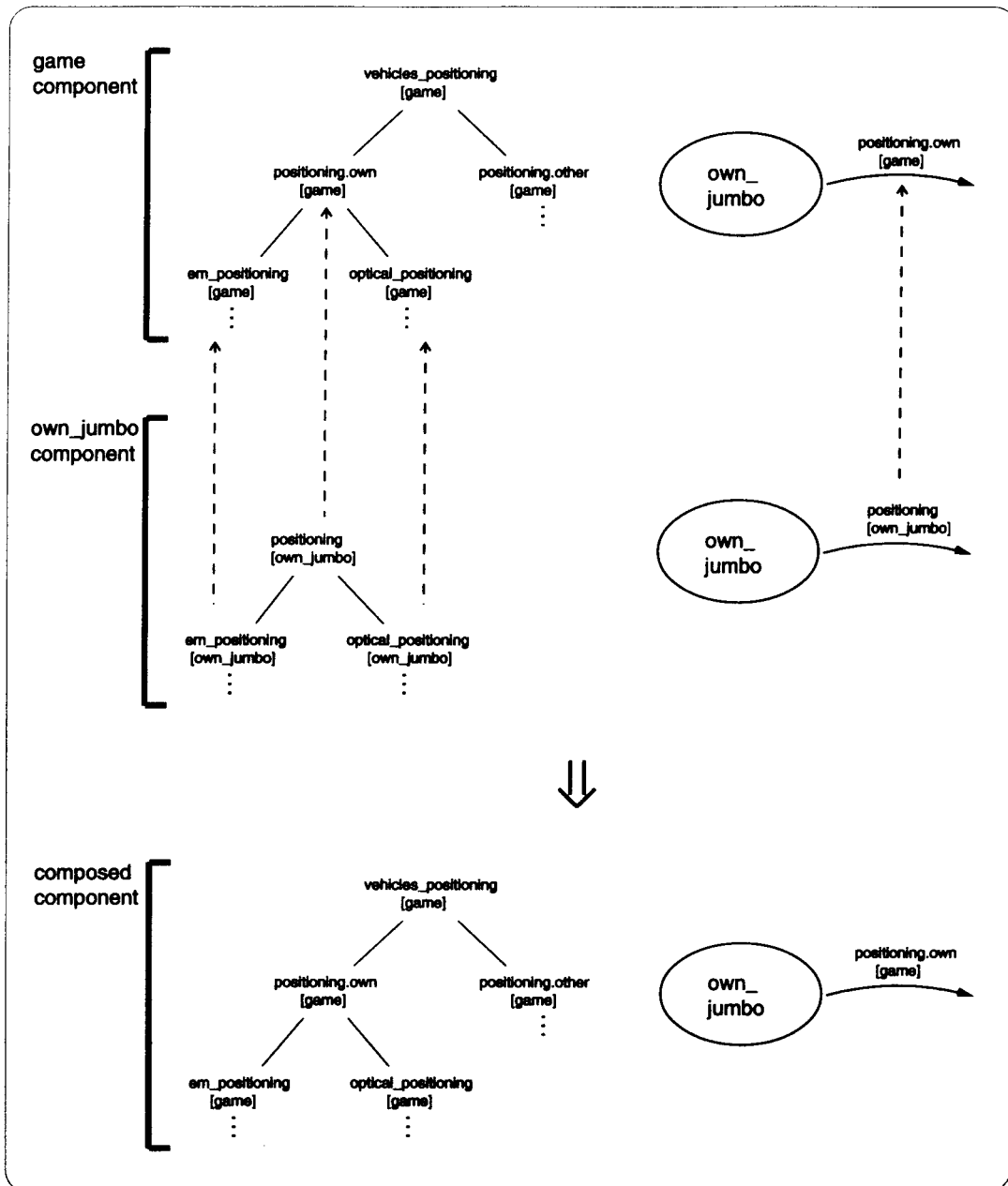


Figure 6.15 An example of meshing (II)

### 6.4 Viewing a component

The composed component is a model and can be browsed in the same way, as done in Chapter two. Figure 6.16 shows a process view being selected, the processes: jet\_engine, jumbo\_positioning\_system, plane\_engine and plane\_positioning\_system. All flows have been selected for this example. The corresponding model view is also shown. Note the presence of model qualifiers clearly distinguishes the two local flows thrust[own\_jumbo] and thrust[other\_plane]. Also, the flows positioning.own[game] and positioning.other[game] are distinguished by their instance qualifiers.

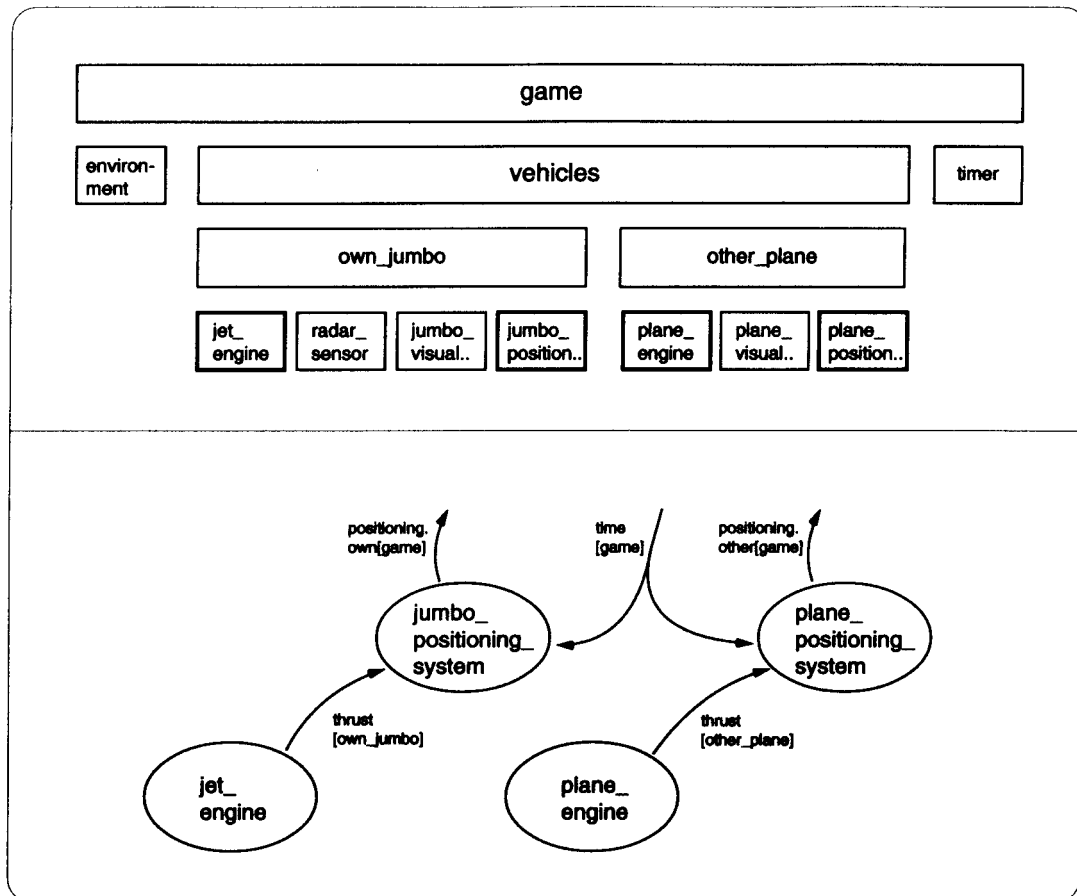


Figure 6.16 A view of the composed component

## 6.5 Discussion

This chapter has shown the key features of components and their composition. Components are models with the addition of: (i) model qualifiers for data flows, which allow each component to have its own local name space for flows, and (ii) typed data flows, which allow a clear distinction between data flows and data flow types. Component composition is flexible: we do not need to explicitly indicate which flows are joined during composition, but rather a meshability relationship between intersecting data flows is used. These features further enhance the scalable construction of large structured graphs, as two models, each containing many DFDs, can be composed with one user operation.

---

## Typed Link Orders

---

In this chapter an extension of link orders: typed link orders, is described. The main motivation of this extension, is to facilitate the definition of components given in the next chapter. However, the introduction of typed link orders into structured graphs are a useful extension in their own right.

Typing in this section does not refer to a data type but refers to the structure of a link order; every instance of a link type will have the same underlying structure of sublinks, though with different instances. In some applications such as structured analysis, leaf links in a final model may also have a data type, such as real or integer, which indicates the nature of the information carried by a link.

A typed structure is defined via schema definitions, on templates, which are then unfolded recursively and instantiated to give rise to instances or occurrences. This makes for economical definition of structures which have common substructure patterns. The resulting occurrences are structures which can be browsed in the usual manner. Consider the specific context of structured analysis.

Structured analysis models tend to contain far more data flows than processes. As the number of processes increases, the number of data flows increases even more. All the data flows require a unique name and a definition in the data dictionary. One way to cope with this name explosion, is to introduce a typing system, so some flows have the same type and hence share the same dictionary schema definition, but still have unique names, so they can be uniquely identified in DFDs for balancing.

The definition of a typed link order, where the order is restricted to a finite tree is straightforward. The same structure that is used for programming language data types can be used. The key properties this tree typed order have are: (i) each occurrence of the same type has the same structure, (ii) there is an intuitive naming scheme and (iii) each link always has a unique name. The second property can be expanded to: there is an association between the names of links which are subtypes and supertypes. Naming has become an explicit concern as names will highlight the structural relationships between link occurrences.

A major challenge of this thesis was to define a typed order which, could be an arbitrary finite ordered set, but have the properties which were identified above for a typed tree order. Also, the typed tree order should be a special case of the more general typed order.



For both the tree typed order and general typed order we are interested in how a user will define such orders through schema, and what the actual orders look like and their associated naming schema. The first section presents tree typed orders and tree order schema for flows, while the second presents general typed orders and order schema.

### 7.1 Tree typed link orders

The properties we wanted for tree typed orders of links were: each link occurrence with the same type should have the same structure, portions of a link with the same type should have the same structure, and the naming of links should indicate these relationships. Next, a link schema and some example models where this schema is unfolded into tree typed orders is demonstrated.

Here is an example link schema definition.

$$a = b.1 + b.2$$

$$b = c + d$$

Each LHS gives a link type. Each RHS gives a set of link labels, where each label is a link type with an optional qualifier. Some example models using these definitions are shown in figure 7.1.

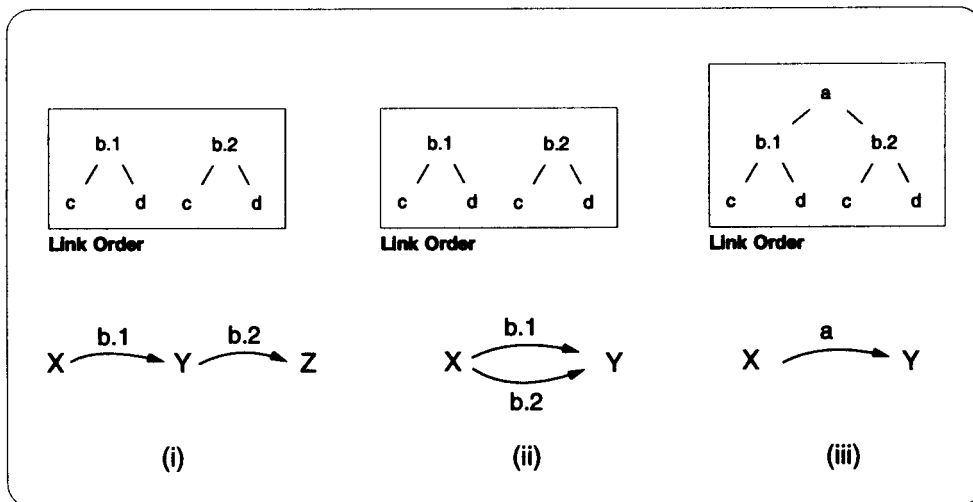


Figure 7.1: Three models with simple link types

Case (i) shows two occurrences of the link type  $b$ :  $b.1$  and  $b.2$ . Note the link schema alone gives no indication of the existence of actual link occurrences; link occurrences can only be found by inspecting the model. Also, in the link schema, link type  $b$  is comprised of  $c$  and  $d$ , so there are two occurrences of link type  $c$  and  $d$  in (i), one for each occurrence of  $b$ .

Cases (ii) and (iii) show links  $b.1$  and  $b.2$  are both produced by node  $X$  and consumed

by node *Y*. In (iii) the link order allows these two links to be further abstracted to *a*. The full names of the links are:

(i),(ii) *c.b.1, d.b.1, b.1, c.b.2, d.b.2, b.2*

(iii) *c.b.1.a, d.b.1.a, b.1.a, c.b.2.a, d.b.2.a, b.2.a, a*

These names are formed simply as the chain of a link's parent labels, with link types and qualifiers being distinct. Note the *a* from the schema, and the *a* from the model in (iii) are distinct, as the former is a link type, while the latter is a link occurrence. The above link names can be shortened, to produce minimal names.

(i),(ii) *c.1, d.1, b.1, c.2, d.2, b.2*

(iii) *c.1, d.1, b.1, c.2, d.2, b.2, a*

Minimal names are the collection of shortest link names which preserve: label sequencing and the initial type of each full name, whilst preserving the uniqueness of names. Clearly, the degree to which full names can be abbreviated is context dependent, that is, depends on what other link names exist.

Full names use a bottom-up style of naming; the leftmost portion of the name refers to the most specialised label. If a top-down style of naming were used then *c.b.1.a* would become *a.b.1.c*; the highest type would appear on the left. The bottom-up style was chosen because it suits the structured analysis application better, and it will make meshing (part of component composition described later) appear better.

In even larger models, ensuring that all links have a unique name would require many qualifiers to be generated, as there is a single name space. The alternative is to violate the unique link name constraint, as often happens in large structured analysis models. Another alternative is given in the next chapter on components, where the additional name qualification mechanism, model qualifiers is introduced.

## 7.2 General typed link orders

General typed orders should be an extension of tree typed orders. Recall tree typed orders satisfy three properties: (i) each occurrence of the same type has the same structure, (ii) there is an intuitive naming scheme and (iii) each link always has a unique global name. General typed orders will be required to satisfy the first two properties, but not the third. Clearly property (iii) is desirable (and typed orders with this property will be used in the definition of components in the next chapter), so we will identify the subset of typed orders which satisfy it later.

The main difference between a tree and an ordered set is that an element of an ordered set may have more than one parent. The challenge is to extend typed order schema to

allow multiple parents whilst preserving the existing tree typed order properties. In particular, preserving the constant structure constraint, each suborder whose root has the same type has the same structure and local naming.

The full name of an element in a tree typed order is the sequence of the element label and all its ancestors labels. That is, an element name is a chain of its ancestor labels. The full name of an element in a general typed order will also be a chain of its ancestor labels.

These requirements can be met by the use of overlapping structures. An element  $n$  with two parents  $p$  and  $q$  is interpreted as two suborders with roots  $p$  and  $q$  which overlap on  $n$ . However, how are the overlapping portions of suborders identified, in a way which meets all the constraints which have been laid out? A first example is shown in figure 7.2.

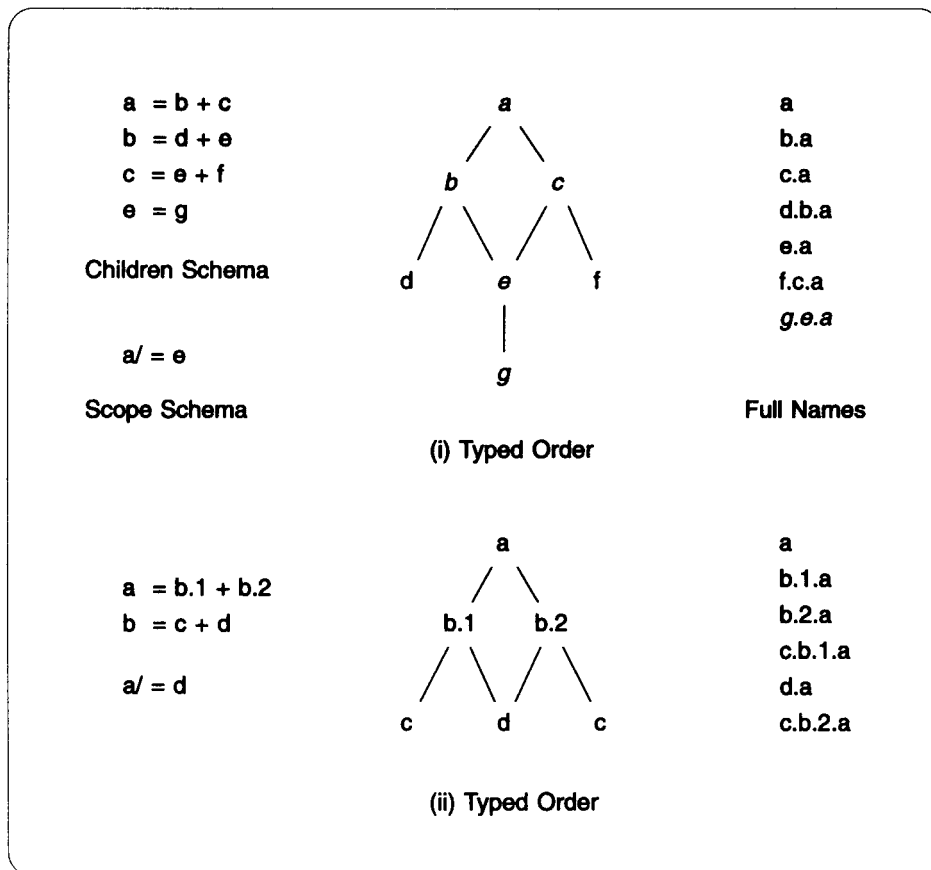


Figure 7.2: A typed link order

The typed order schema has two parts: the children schema and scope schema. The children schema is the same schema used to generate tree typed orders. It is the scope schema which allows overlapping structures. In (i) the scope schema is read as, under any occurrence of an element with type  $a$  there is at most one occurrence of an element with label  $e$ . So the suborders of  $b$  and  $c$  overlap at the occurrences of  $e$ , with the result that  $e$  has parents  $b$  and

*c*. The full names of those elements with recursively one parent, are the same as for tree typed orders. Applying the same rule to element *e* there are two possible full names: *e.b.a* and *e.c.a*. Element *g* has the single parent *e*, so its name should be *g* followed by *e*'s name.

Recall we would like links to have unique global names, so there is an arbitrary choice between the two names for *e*. Clearly, arbitrary choice is unacceptable. This is resolved by noting that, as a link with label *e* occurs at most once under link *a*, given the scope schema, the full name *e.a* is guaranteed to be unique and so can be used. In general, a full name can be constructed from the chain of scoping ancestors, however as we will see in the next section such full names are not always unique. This will be made more precise as more examples are considered. Figure 7.2(ii) shows an example where the overlapping suborders of *b.1* and *b.2* have the same type. Note, though they overlap, each occurrence of *b* has the same labelled structure below it.

### 7.3 Link schema constraints

In the previous section the focus was on the typed order, and their corresponding schemata. In this section the focus is on how to constrain schema so that they only give rise to typed orders. First constraints that ensure a tree order schema always produces a tree order are presented. Then constraints that ensure a general order schema always produces a typed order (sub-orders with the same type have the same local structure), and an order which has unique full names, are presented. The precise definitions of these constraints are given in Chapter nine on typed orders.

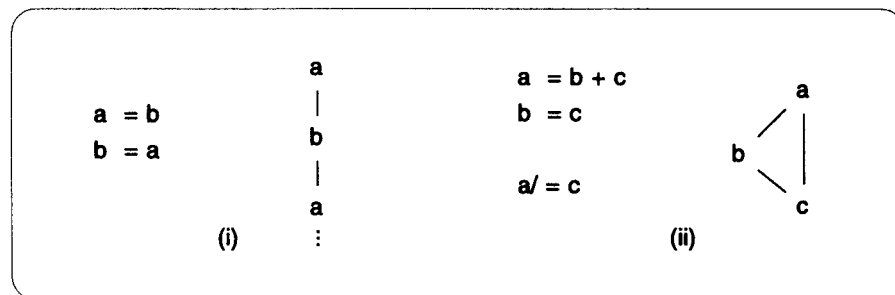


Figure 7.3: two “schema” which do not unfold into ordered sets

Recall, the first constraint for a typed order is that it is an ordered set. Figure 7.3 shows two schemata which do not produce ordered sets. In case (i) a label with type *a* is below type *a*, so it is a graph containing a cycle. In case (ii) link *c* has two parents which are related, so this graph is not the covering relation of an ordered set. To exclude these cases two schema constraints are needed: *no type cycles* and *no redundant transitive links*.

Consider the example shown in figure 7.4. Typed order (i) contains only one occurrence of *e*, because the scope schema has defined label *e* to have scope of type *a*. The typed order (ii) contains two occurrences of links with label *e*, as there is no scope schema for type *b* or

any type below it. The occurrence of  $b$  in the larger context of type  $a$  has a different structure to the typed order with root  $b$ , so the structure of a suborder with root type  $b$  is dependent on its context. Clearly this is in violation of the requirement that occurrences of a type have the same local structure, that is, be context independent.

This lack of constant structure for type occurrences can be resolved in figure 7.4 by the addition of the scope schema:

$$b/ = e$$

With this schema included, the typed order with root  $b$ , has a single occurrence of  $e$ , and hence the same downwards structure as the other occurrences of  $b$  under type  $a$ .

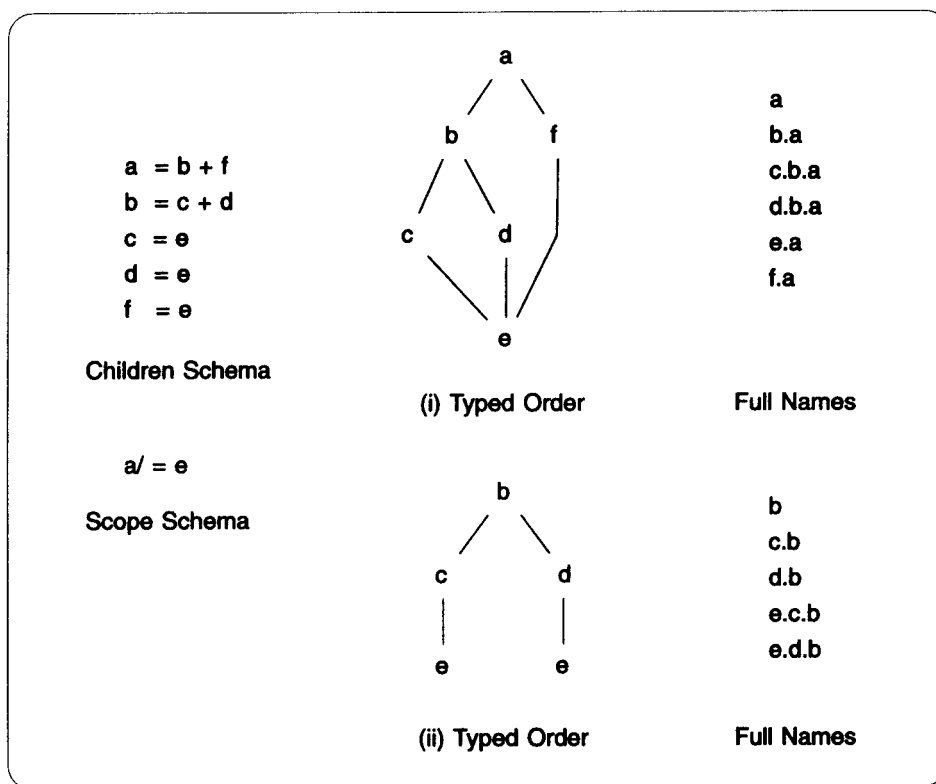


Figure 7.4: A “typed order” schema which lacks the constant structure property

In general, if a type  $t$  scopes label  $x$  then all types which can occur between an instance of  $t$  and  $x$  must also scope the label  $x$ . In figure 7.4, links with types  $b, c, d$  and  $f$  appear between an instance of type  $a$  and a link with label  $e$ , so these types should all scope label  $e$ . This constraint on the scope schema is called *scope continuity*.

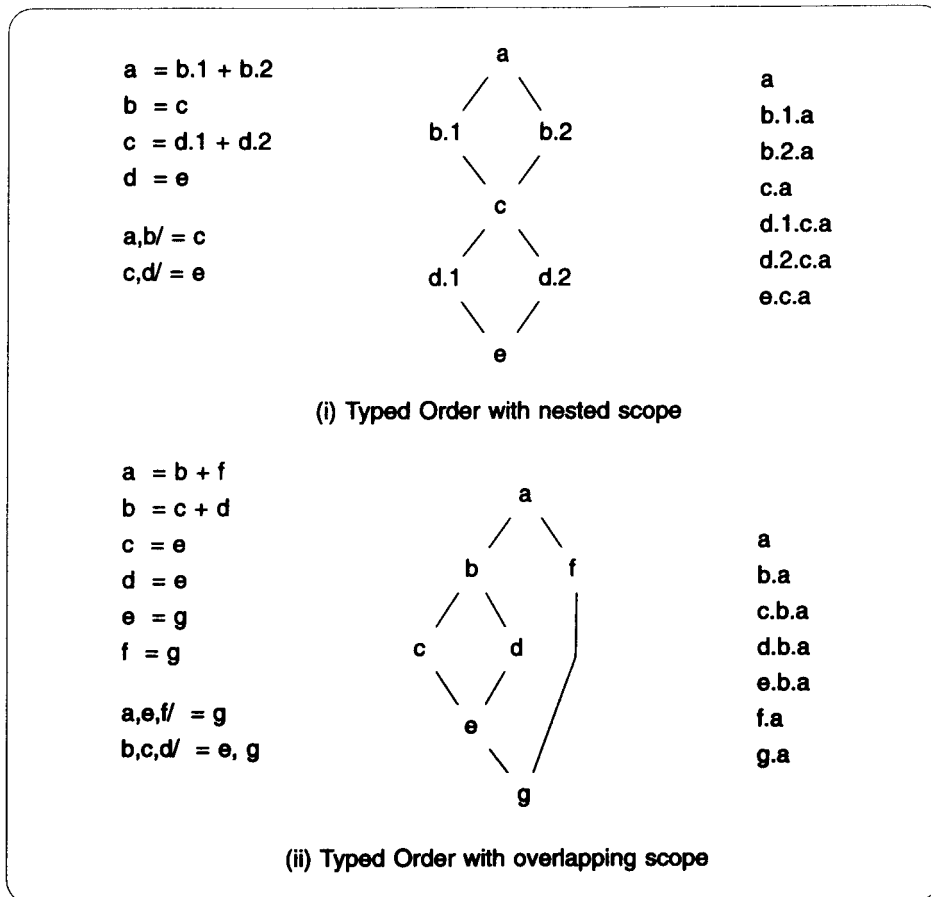


Figure 7.5: Two typed orders with nested and overlapping scope

When a scope schema satisfies scope continuity a smaller representation can be used; root scope schema, where only the highest types with a given scope are shown. The root scope schema for the example shown in figure 7.5(ii) is:

$$a/ = g$$

$$b/ = e$$

The scoping types of  $g$  are  $a, b, c, d, e$  and  $f$ . The highest of these is  $a$ , so it is the root scope of  $g$ . The scoping types of  $e$  are  $b, c$  and  $d$ , so its root scope is  $b$ . In the rest of this chapter all scope schema satisfy scope continuity, allowing the root scope schema representation to be used.

Next we consider the naming of elements in a typed order. In figure 7.4(i) link  $e$ 's full name starts with label  $e$ . The next label should be the highest link above  $e$  with scope of  $e$ ; that is the root scope link of  $e$ , link  $a$ . The resulting full name is then  $e.a$ . In general, the subset of ancestor labels used in a links full name, are those ancestors which are the successive root scope links, or where there is no root scope link, the parent link is used. So in the absence of any scope schema, this rule gives the successive chain of parents of a link as its full name, as for the tree typed order case.

Figure 7.2 and 7.4 showed examples where only one link had multiple parents. Figure 7.5(i) shows an example with nested scoping. Label  $e$  has root scope type  $c$ , and label  $c$  has root scope type  $a$ , so link  $e$  has a full name  $e.c.a$ . Figure 7.5(ii) shows an example with overlapping scope. Label  $e$  has root scope type  $b$  and label  $g$  has root scope type  $a$ . So the links with labels  $e$  and  $g$  have full names  $e.b.a$  and  $g.a$ .

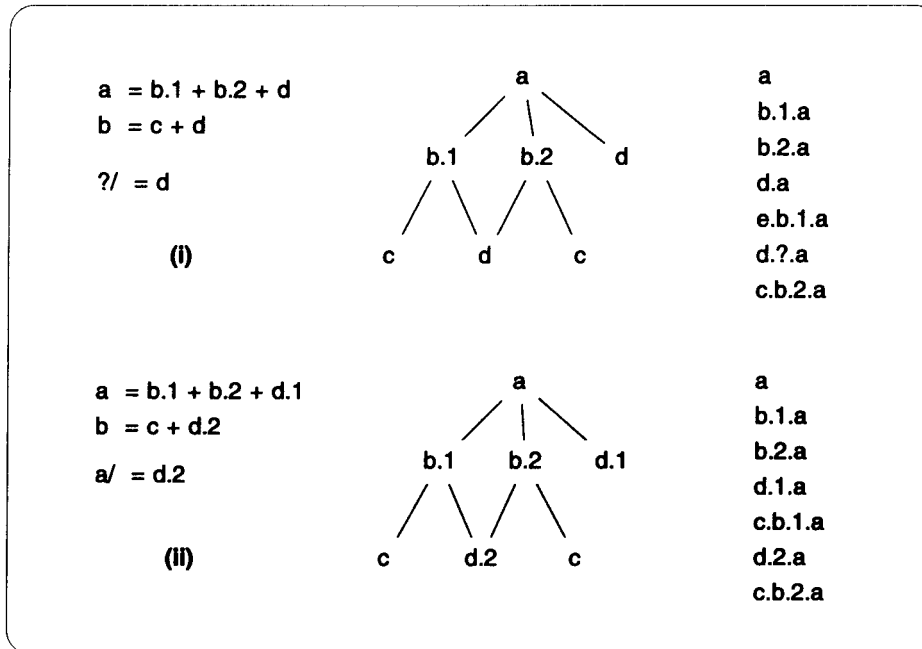


Figure 7.6: A typed order lacking unique names

Consider the example shown in figure 7.6(i). There are two links with label  $d$ , one is shared by  $b.1$  and  $b.2$  while the other has the single parent  $a$ . This is a valid typed order as it is an ordered set and satisfies the constant structure constraint. However, using the rule already given for forming full names, the full names of the two occurrences of  $d$  are,  $d.a$  and  $d.a$ . They are not unique. So this typed order does not have the additional property of having a unique full name for each link. The remedy for 7.6(i) is to distinguish the two occurrences of  $d$  with qualifiers so one occurrence can have  $a$  as its scope. This is done in figure 7.6(ii) where label  $d.2$  has scope of  $a$ , so the two full names for links with type  $d$  are now:  $d.2.a$  and  $d.1.a$ .

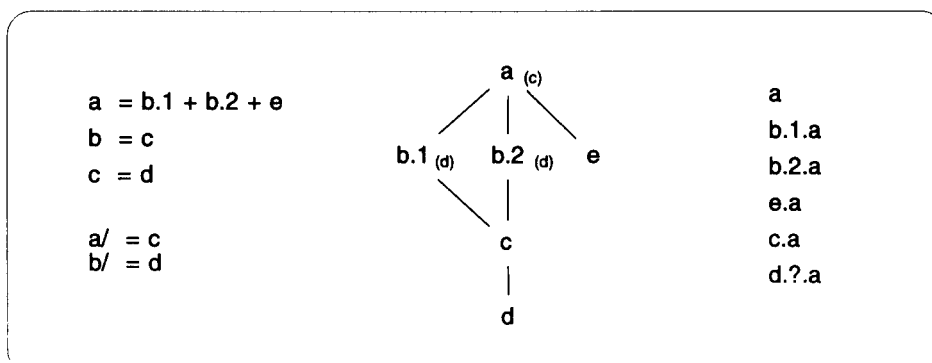


Figure 7.7: A typed order lacking scope based chain names

Figure 7.7 shows a typed order; it is an ordered set, has constant structure for each type occurrence and additionally has unique chain names for each link. However, using the schema given and the rule given before for constructing link names with scope schema, link  $d$  does not have a chain name. Its root scope links are  $b.1$  and  $b.2$  whose parent is  $a$ . Directly using the full name rule  $d$ 's name is  $d.(b.1,b.2).a$  which is not a chain. The problem is that  $d$  has two root scope links, that is  $b.1$  and  $b.2$ . A more extreme example of this problem is shown in figure 7.8, where the full name of the link with label  $n$  is the tree whose paths are:  $n.k.c.a$ ,  $n.k.d.a$ ,  $n.l.g.a$ , and  $n.l.h.a$ .

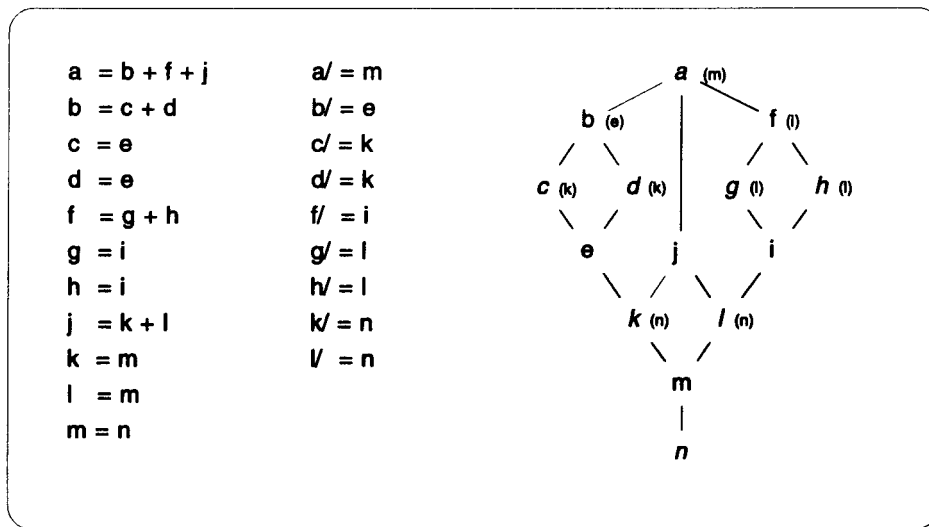


Figure 7.8: A typed order which contains an element with a tree name

Root scopes are shown in figures 7.7 and 7.8 as labels within brackets on the diagram of the order. In figure 7.7 the root scope of the element with label  $c$ , is the element  $a$ . The root scopes of the element with label  $d$ , are the elements  $b.1.a$  and  $b.2.a$ . In figure 7.7 element  $c$  has a unique root scope, but element  $d$  does not. In figure 7.8 there are several elements without unique root scopes: elements  $n$ ,  $k$ , and  $l$ .

Typed orders without chain names can be avoided. The typed orders shown in figures 7.7 and 7.8 fail to have chain names because some links have multiple root scope elements, which can only occur when the typed order is not a tree. If all links in a typed order have *unique root scope* links, then all links in the typed order will have a chain name.

### 7.4 Summary

In summary, a typed order is: an ordered set, and satisfies constant local structure for occurrences of the same type. To ensure a typed order schema unfolds into a typed order which has unique chain names for each element, it must have no type cycles, no redundant transitive links, scope continuity and unique root scopes.



---

## Component Composition

---

To cope with building very large models, a system where medium sized models (*components*) are created by individuals, which are composed to form a final large model is desired. However, to be scalable the dependencies between components must be minimised while the ability to flexibly couple components is maximised. Component composition itself should not introduce new constraints on editing that are not present in structured graph editing. In particular, arbitrary forms of composition should be supported, not just a top-down form.

Components are models with additional structure, so all scalable browsing and editing operations which apply to models also apply to components. Components have additional structure to further improve scalable editing beyond that available for models: the use of link orders which are also typed orders. In a large model many links may have a common structure. Naming links manually is a problem, if their common structure is to be maintained. The use of link schema addresses this maintenance problem. From a user perspective, a link order schema is first defined then instantiated as many times as required.

Components are primarily a building block for making large models. This implies that components can be built separately then plugged together. With this in mind, this chapter is perhaps more about component composition than components themselves. Models can already be plugged together, as this was how some model editing operations were specified; that is as the union of two models. What distinguishes component composition from model composition is its flexibility. Flexibility of composition means the ability to use one component in a range of contexts with the component adapting to each context. The purpose of this chapter is to present component composition and demonstrate its flexibility.

Minimizing the interdependencies between components makes their separate construction easier. So when a person is choosing the name for an internal link for his/her component, he/she should not need to check what link names are being used in other components. However, there can not be complete freedom in the design of a component, as components are intended to be coupled together, and so there must be some interface constraints.

Flexible component coupling should minimize interface constraints between components. A component composition which requires links from different

components to have the same name to couple under component composition is too rigid. For example, when constructing, a lower component (lower in the final node order), may have a link, and this link has a name. In the higher component (the one which will be composed above), the same name cannot be used as we have several such links. So we wish to support the coupling or joining of links in different components which have different names.

The next section looks at existing model composition, and its limitations, which motivated the need for component composition. Then the definition of components and their composition will be presented in stages. Firstly, components are presented. They are models containing typed link orders and links with local name scopes. Secondly, basic component composition is presented, where model composition has been extended to cope with the extra structure of components but with no extra flexibility. Then, component composition, where components are flexibly coupled via the mechanism of *mesh* is presented. Finally, a limitation of components is presented. The formal definitions of these structures and operations are given in Chapters nine and ten.

### **8.1 Limitations of model composition**

In the formal description of model editing in Chapter five, the addition of nodes and links to a model was accomplished by model composition. When two models are composed: the node orders and link orders are merged by mapping nodes and links with the same name to the same node or link in the composed model, producer and consumer sets are made by the union of the source model producer and consumer sets. Clearly these models are sharing one name space for node and link names. This approach is fine when the final composed model is small and a large proportion of links are coupled links, that is, links from different models which are intended to map to the same link in the composed model. However, once the final composed model becomes large, there is a need for independent naming of links within each model to be composed.

The major concern is with the link name space and not the node name space, because in a large structured analysis models there may typically be several hundred nodes but several thousand links. Also, as a model gets larger there is a tendency for the number of links to grow faster than the number of nodes, so the situation gets worse for very large models. Here, a global node name space is used to join component nodes together, while a global link schema will allow component links to be joined together.

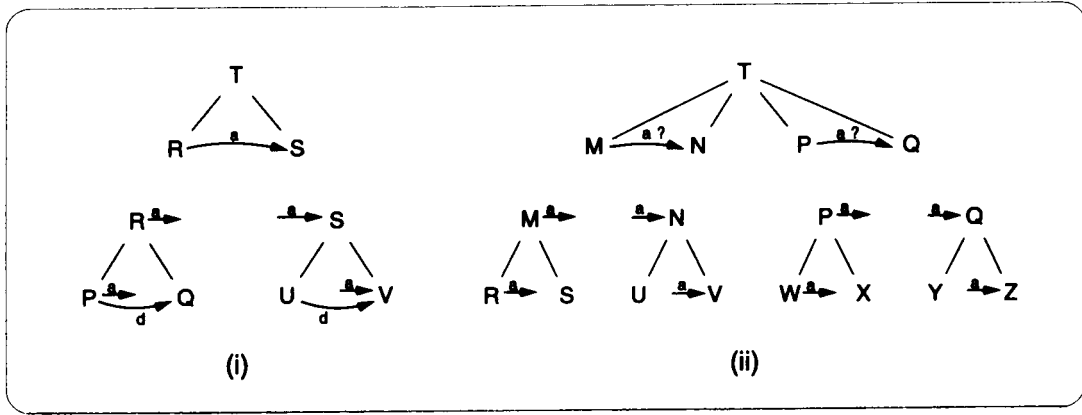


Figure 8.1 Groups of models to be composed

Consider figure 8.1. Case (i) shows three models which are to be composed. Using the rule that nodes and links with the same name map to the same node and link, with names being preserved the result would be just two links. So if the intention was to merge the *a* links but leave the *d* links as distinct, this can not be achieved. Case (ii) shows a related problem, we wish to choose which *a* links will couple. Model composition would result in there being just one link *a* in the composed model. So if the intention is to merge the link *a* which occurs in models *M* and *N*, and separately merge the link *a* which occurs in the models *P* and *Q*, this can not be achieved. To facilitate a composition which allows these intended merges to occur, a model with additional structure is required: components.

### 8.2 Components

A component is a model which has been extended in two ways. Firstly, the link order is a typed order. Secondly, a link name comprises: a link order name and a model qualifier (a set of node names). The latter allows local name scopes for links. A component's links are shown as  $\alpha[\beta]$  where  $\alpha$  is a typed link order name, and  $\beta$  the model qualifier is a set of node names. However, in typical components, the model qualifier is a single node name. Figure 8.2 shows some example components.

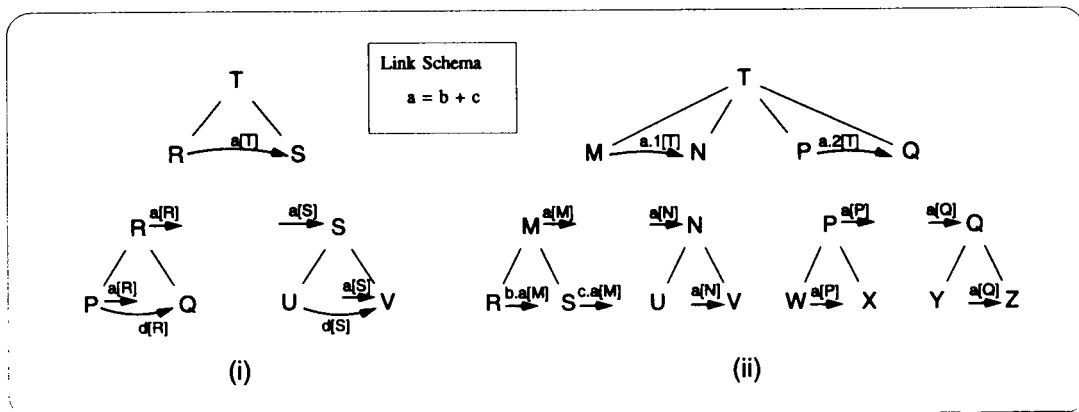


Figure 8.2 Groups of components to be composed

The example shown in figure 8.2 is similar to that of figure 8.1, but uses components. Figure 8.2 differs as it introduced a link schema used in the submodel with root node *M*. In (i) there are the links *a*[*T*], *a*[*R*], *b*[*R*], *a*[*S*] and *b*[*S*]. In (ii) there are the links *a.1*[*T*], *a.2*[*T*], *a*[*M*], *a*[*N*], *a*[*P*] and *a*[*Q*]. When these models are composed, we will want some of these links to merge. This will be the subject of the next section, component composition.

We now indicate some constraints for components, the precise constraints are given in Chapter 10. A component in addition to being a full model typically satisfies:

- all producer and consumer nodes of link  $\alpha[\beta]$  are below or equal any node in  $\beta$
- a link's model qualifier is the intersection of its descendent link model qualifiers
- links can be distinguished by their model names (link name and model qualifier)

The first constraint ensures that a user can read  $\alpha[\beta]$  as the link  $\alpha$  in the submodel with root nodes  $\beta$ . The second constraint gives the rule for deriving non-leaf link model qualifiers from descendent leaf link model qualifier. This builds on the same concept used in the first constraint; a link can only appear within its model qualifier submodel. A non-leaf link should only appear (as a producer or consumer) where, all of its leaf links can appear. This is given by the intersection of model qualifier descendants, where a flat representation of this is used, i.e. the highest such qualifier nodes.

### 8.3 A component composition example

In this section we introduce component composition with an example. The details of component composition will be presented in sections 8.4 and 8.5. Figure 8.3 shows the composition of the components shown in figure 8.2.

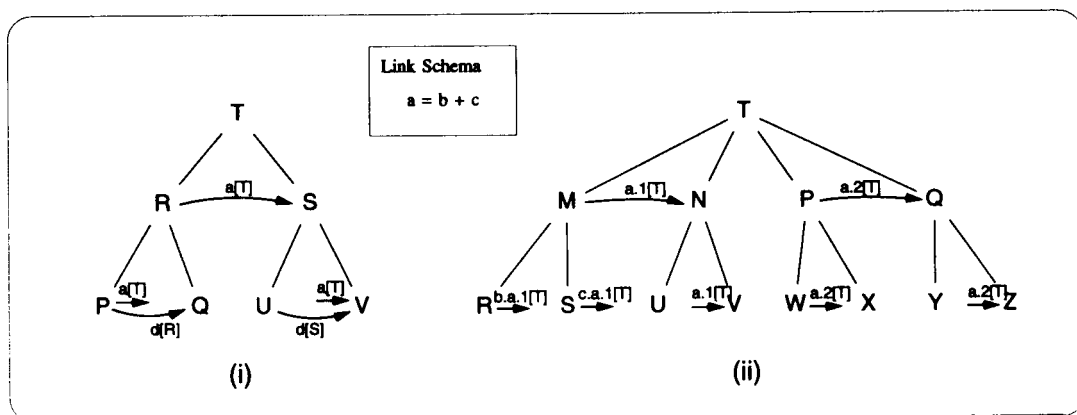


Figure 8.3 The composition of the component groups shown in figure 8.2

Figure 8.3 (i) shows three models have been composed into the one model. The links

$a[R]$ ,  $a[S]$  and  $a[T]$  have merged into  $a[T]$  in the composed model, while the links  $d[R]$  and  $d[S]$  have remained distinct. In (ii) the links  $a[M]$  and  $a.I[T]$  have meshed together, so that the children of  $a[M]$ ;  $b.a[M]$  and  $c.a[M]$ , have become the children of  $a.I[T]$ ;  $b.a.I[T]$  and  $c.a.I[T]$ .

### 8.4 Component merge

This section describes a basic joining of components. The term component merge is used to distinguish it from component composition which is more general.

Component merge maps a collection of components which have a common link schema and composable node orders, to a single merged model, by mapping links and nodes in each component to links and nodes in the merged model.

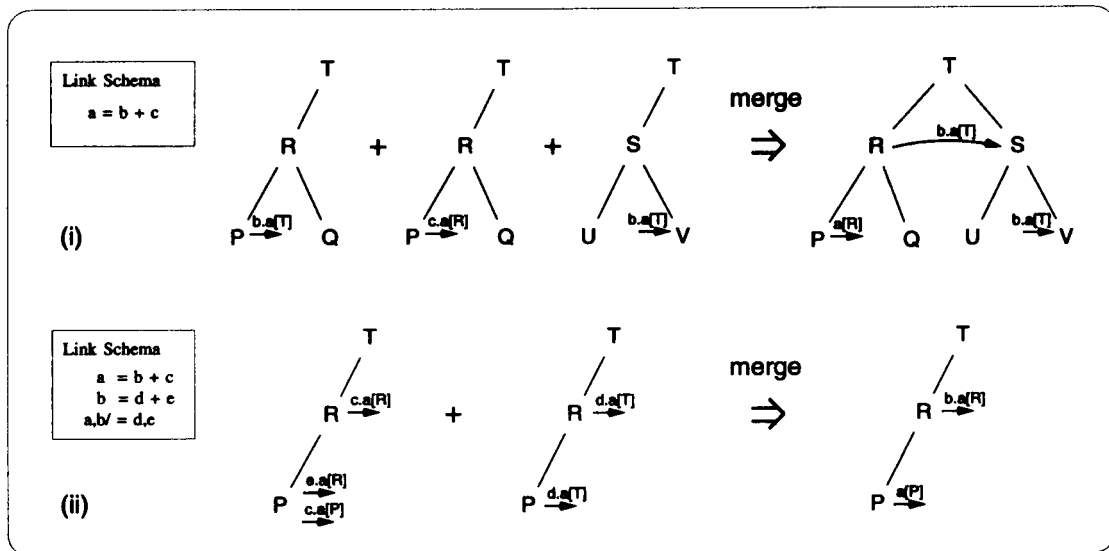


Figure 8.4 Examples of component merge

Components are merged by mapping nodes with the same name to the same node in the merged model, and by mapping leaf links with the same name to the same leaf link in the merged model whilst preserving names. Also, because the result of component merge is a component and a component is a full model, some filling-in of link producers and consumers, and abstracting of interfaces may be done. Some examples of component merge are shown in figure 8.4.

Figure 8.4 (i) shows three components being merged. Nodes with the same name, map together, but all parent-child relationships are maintained, so the nodes with name  $T$  merge, and nodes  $R$  and  $S$  map to children of the merged  $T$  node. Merging preserves producer and consumer relationships, so links  $b.a[T]$  and  $c.a[R]$  retain node  $P$  as their producer. As the merged component is a full model, all link interfaces must be maximum, so node  $P$ 's interface is abstracted to  $a.a[R]$ . Note the model qualifier is  $R$ , the intersection of model qualifiers  $T$  and  $R$ . In the first component link  $b.a[T]$  is produced

and in the third component  $b.a[T]$  is consumed. Recall all interfaces of a full model must be net interfaces, so in the merged component nodes  $R$  and  $S$  respectively produce and consume  $b.a[T]$ .

Figure 8.4 (ii) shows further examples link abstraction occurring as a result of component merge. Links  $c.a[R]$  and  $d.a[T]$  produced by node  $R$  are merged and abstracted to  $b.a[R]$ , while links  $e.a[R]$ ,  $c.a[P]$  and  $d.a[T]$  produced by node  $P$  are merged and abstracted to  $a[P]$ . In summary, component merge is the minimum extension of model composition which supports components, that is, models which have the additional structure of a typed link order and model qualifiers.

### 8.5 Component composition

Component composition is an operation which takes a collection of components and combines them into a single component, so node orders, link orders and producer/consumer relations are all combined. The major property we want for this operation is, that model views are preserved where possible. That is, whether a component is viewed in isolation, or as part of a larger component (after composition) the same structure is seen.

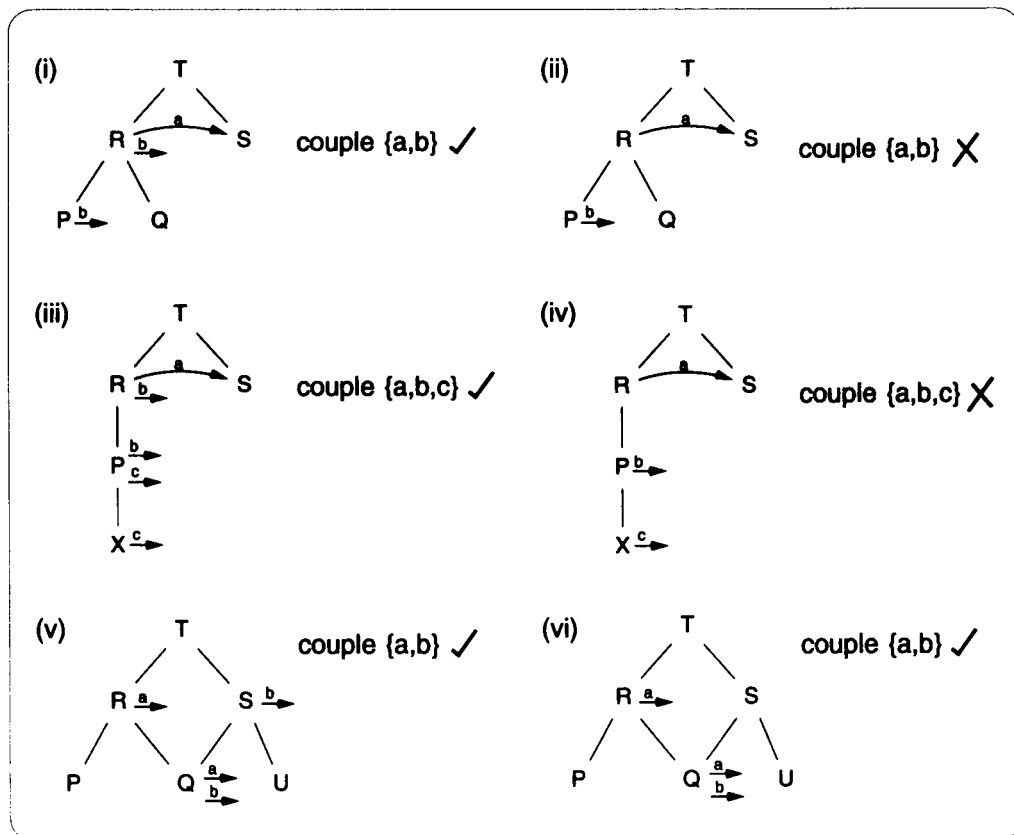


Figure 8.5 Examples of link coupling

Components are composed by mapping nodes from different models with the same name to the same node in the composed model, but for links a more flexible scheme is

used. In addition to mapping links with the same name from different models together (*merging*), two links may also map to the same link when they couple and their context is comparable (*meshing*). Link coupling and link context order are defined next.

Links couple when they share a common producer or consumer node. Examples of link coupling are shown in figure 8.5. In (i) links *a* and *b* couple as they share a common producer, node *R*, but in (ii) they do not couple as they have no common producer and no common consumer nodes. Case (iii) shows link coupling is transitive. Cases (iv) - (vi) provide additional examples of coupling and non-coupling.

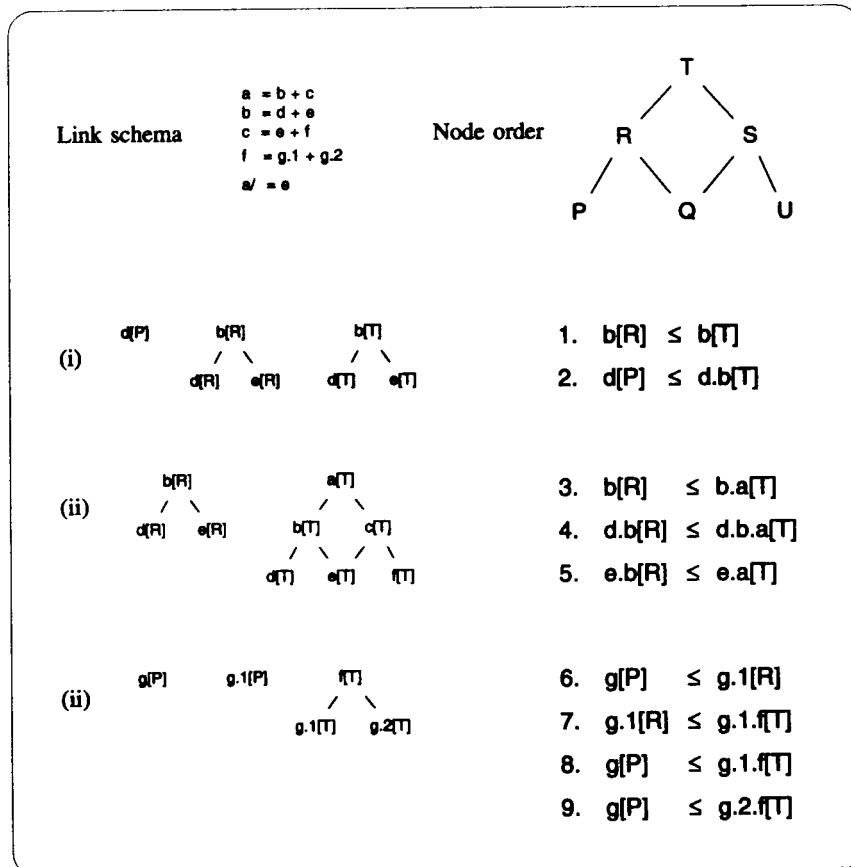


Figure 8.6 Examples of link context ordering

Link *x* is less than (w.r.t. the context order) *y* when: (i) *x* and its ancestors can embed into *y* and its ancestors, and (ii) *x*'s model qualifier is less than or equal to *y*'s model qualifier. The first condition is typically true when, *y*'s name (without the model qualifier) is an extension of *x*'s name. In figure 8.6 case 2, the link *d[P]* is less than (context) *d.b[T]* because the suborder of *d[P]* and its ancestors (just the single element *d[P]*) embeds into the suborder of *d.b[T]* and its ancestors (the suborder containing *d.b[T]* and *b[T]*). Also, the *d.b* in *d.b[T]*, is a extension of *d* in *d[P]*. Case 5 shows an example where this name extension rule does not satisfy condition (i). This situation arises when the typed link order is not a tree. The formal definitions of context ordering are given in Chapter nine.

When components are composed, all links which may merge (they have the same name) are combined, but some links which are meshable (they couple and are context ordered) may not be combined. There are three situations in which the latter occurs: mesh clash, arbitrary choice and unique leaf link name violation.

Mesh clash occurs when two links (which are not meshable with each other) are meshable with a link in another component. An example is shown in figure 8.7. Node  $R$  in the bottom component produces link  $a[R]$ , while node  $R$  in the top component produces links  $a.1[T]$  and  $a.2[T]$ . Link  $a[R]$  is meshable with both  $a.1[T]$  and  $a.2[T]$  but it cannot mesh with both of them, as this would result in  $a.1[T]$  and  $a.2[T]$  mapping to the same link.

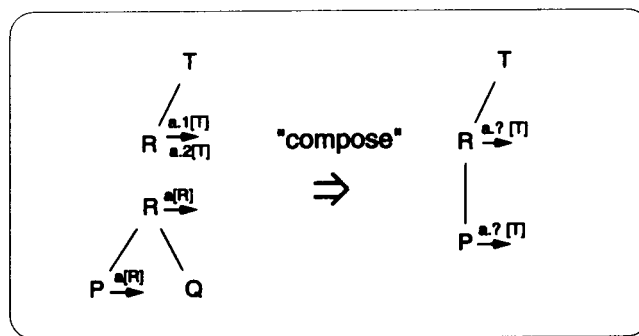


Figure 8.7 An example of mesh clash

Mesh clash may sometimes be resolved. So while the merging of two links may appear to introduce a mesh clash locally, the presence of other components may globally resolve this local clash. An example is shown in figure 8.8. In (i) four components compose and there is a mesh clash between link  $a[P]$  and links  $a.1[R]$  and  $a.1[S]$ . Links  $a.1[R]$  and  $a.1[S]$  don't mesh as their model qualifiers are non-comparable. In (ii) the mesh clash is resolved by the addition of a new top component which adds the link  $a.1[T]$ . Both  $a.1[R]$  and  $a.1[S]$  mesh with  $a.1[T]$ , and so  $a[P]$  meshes with  $a.1[T]$  without any clash. In general a mesh clash is resolvable when the two links which clash  $a[M]$  and  $b[N]$ , only fail to be meshable because their model qualifiers are non-comparable. The later addition of a link  $b[R]$  from a common node ancestor could resolve the clash.



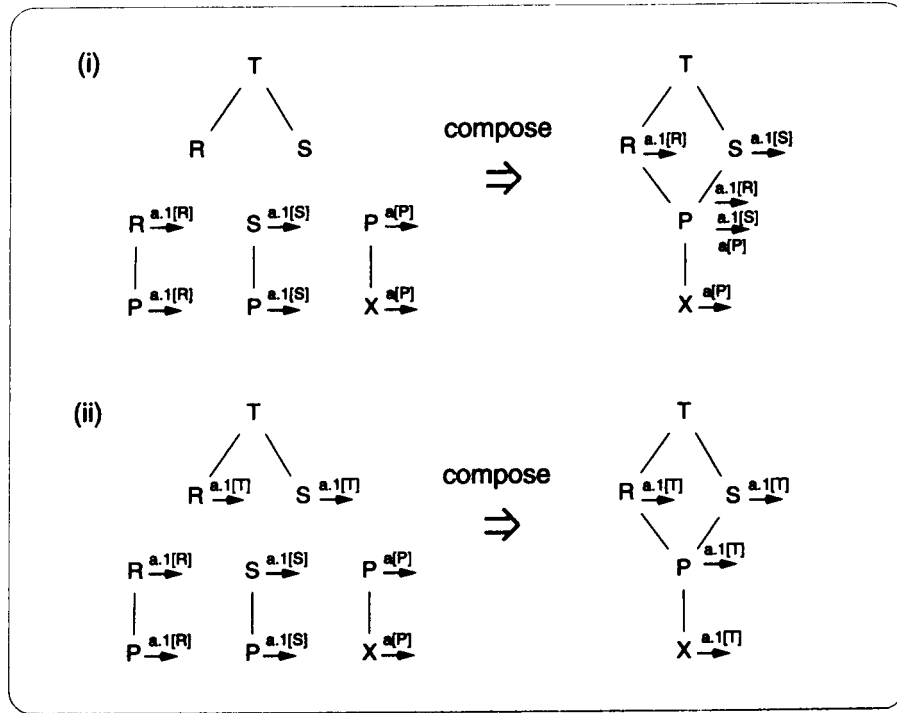


Figure 8.8 Examples of resolvable simple mesh clash

Arbitrary choices about which link to mesh, can occur when there are unresolved mesh clashes which involve three or more links, so that arbitrary subsets of a clash collection can still mesh together. For example in figure 8.7 link  $a[R]$  meshes with  $a.1[T]$  and not with  $a.2[T]$  is an acceptable mesh. However, the choice of which links to mesh in these cases was arbitrary. The example shown in figure 8.9 shows these arbitrary choices explicitly. Link  $b.a[R]$  may mesh with  $b.a.1[T]$ ,  $b.a.2[T]$  or none. These choices result in the components shown in (i), (ii) and (iii).

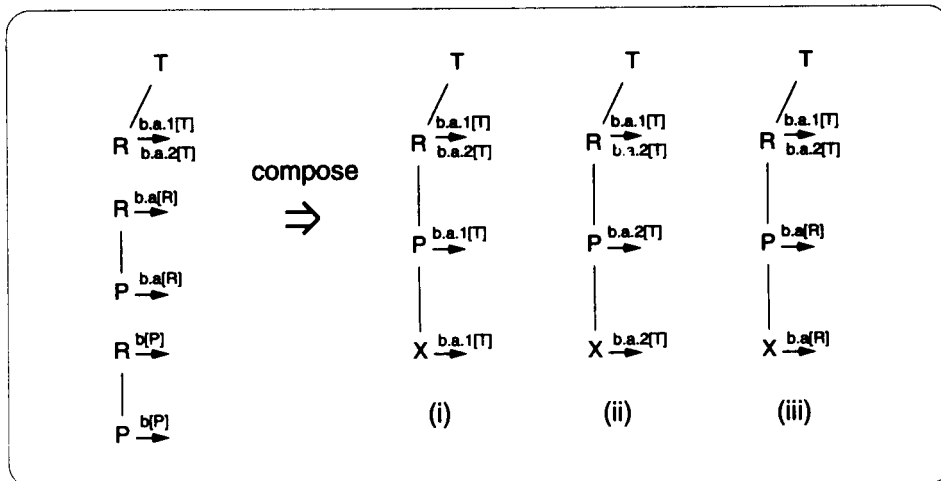


Figure 8.9 An example of arbitrary mesh choices

In figure 8.7 the best resolution of the mesh clash, is no meshing. In figure 8.9 the best mesh is  $b[P]$  meshing with  $b.a[R]$  but  $b.a[R]$  meshes with nothing, that is case (iii). In general, the best mesh is achieved when all links which are meshable are combined,

except where it would lead to unresolvable mesh clash. The formal definition of best mesh is given in Chapter ten.

In summary, this section has introduced component composition. Flexible composition was facilitated via the use of meshing, while constraints on the composition ensured that meshing occurred in a maximum but predictable manner. Independent naming of internal links within a component was facilitated via the use of model qualifiers. Together, these extensions to basic structured graphs provide better support for construction of large models by a team. Next, in the final section, a limitation of components is discussed.

### 8.6 A component limitation

A limitation of components and their composition, is the lack of unique names for non-leaf links in components. This section will present the situations where composition fails to attain this property.

Components may not have unique names for non-leaf links. This allows component merge and hence composition to be total functions defined on all components, rather than being undefined for those components whose merging introduces non-unique link names. Examples of components without unique link names are shown in figure 8.10.

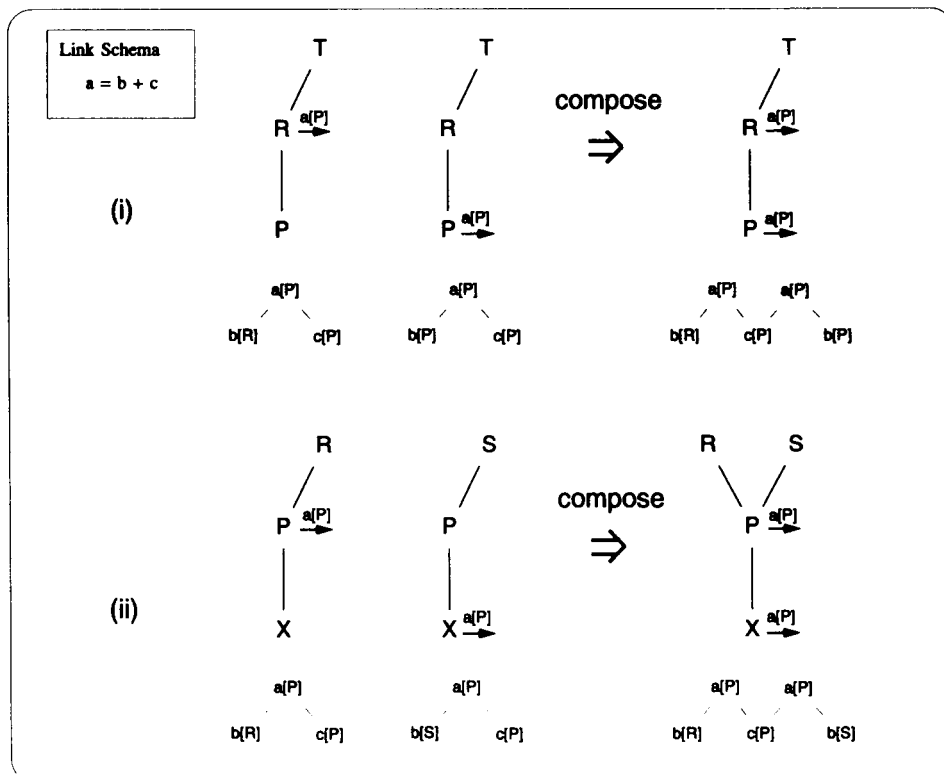


Figure 8.10 Examples of components without unique non-leaf link names

In figure 8.10 (i) two components are composed. Each component contains a non-leaf

link  $a[P]$ . The left component  $a[P]$  contains  $b.a[R]$  and  $c.a[P]$ , while the right component contains  $b.a[P]$  and  $c.a[P]$ . The model qualifier of the  $b.a$  link differs. The leaf links  $c.a[P]$  from each component maps to the same link as they have the same name. The links  $b.a[R]$  and  $b.a[P]$  map to distinct links. The result is the overlapping link order with two roots,  $a[P]$  and  $a[P]$  which are distinct as they represent different sets of leaf links but they have the same name. The example shown in (ii) is similar except the leaf link which have the same context  $b.a[R]$  and  $b.a[S]$  have model qualifiers which are non-comparable but still overlap.

Typically the cases which give rise to non-unique names are rare, so an implementation of this system could warn the user, indicating the problem links and allow the user to undo the composition and adjust the names of these links.

---

## Typed Order Formalism

---

This chapter presents the formal definitions of typed orders and the schemas for generating them. Typed link orders have been presented and motivated in chapter seven. The major concern was to support the construction of large orders, where many suborders have a common structure and common naming by allowing a user to specify the common parts: the schema. Major challenges were: to define a typed order which could be an arbitrary ordered set rather than just a tree; and to define an associated typed order schema.

In structured graphs the naming of nodes and links was not a concern as it was assumed each distinct node and link would have its own unique name. For typed orders naming is a major concern, because we would expect corresponding elements in suborders with the same type to have a related name. That is, we would expect the naming of elements to highlight their type relationship. Such a naming scheme for type ordered trees is straightforward. The other major challenge was: to define such a naming scheme for general typed orders which produces names which are also unique.

Typed orders will be presented in four stages. First, labelled orders are presented. These are an extension of ordered sets in which each element has an associated label, and global names are defined in terms of these labels. Second, typed orders are presented. These are labelled orders extended with type information. Third and fourth, typed order schema and the unfolding functions are presented for both trees and the general case; these functions map order schema to typed orders.

### 9.1 Labelled orders

A labelled order is an ordered set where each element has a label. The key design choice here is: what will the scope of each label be? There are two natural choices. Each label could have global scope: a label distinguishes an element from all other elements in the order. The other choice is that each label could have local scope: a label only distinguishes an element from all of its siblings.

Whether global or local labels are chosen, we wish to be able to identify each element amongst all other elements. *Names* will be used for this purpose. However we don't insist on names being unique, though we do try to choose a naming scheme which will deliver unique names in most cases of interest. Clearly, if globally scoped labels are used, they can also be used as names. However, if locally scoped labels are used, names must contain more than the local label.

In this thesis labelled orders with locally scoped labels are chosen, as this is more general and supports the style of typed order presented later. It is more general as a globally scoped order can be described as a locally scoped order with the additional constraint that each label is also globally unique. Next, labelled orders are defined; this is followed by the introduction of a naming scheme.

**Definition 9.1** Let  $(P, \leq)$  be a finite ordered set,  $L : P \rightarrow Label$ . The tuple  $(P, \leq, L, \leq_{label})$  is a *labelled order* when:

$$\forall x, y : P \bullet$$

$$L x = L y \Rightarrow \text{not siblings } \{x, y\} \quad \square$$

The constraint requires labels of sibling elements to be distinct. Note that  $L$  is a total function, so that all labels participate in the label order. The label order captures the style of labels used in this thesis; labels as a sequence of text strings separated by full stops, for example, *a.1.1* and *engine.fore.left*. The label order will play an important part in component meshing, described in the next chapter.

For labelled orders which are trees there is a natural naming scheme: for each element its name is a sequence of labels formed by the element's label and its ancestor labels. The ordering of labels in the name sequence matches the order of the corresponding ancestor elements in the labelled order, that is the chain of ancestor elements. Such names can be easily incorporated into text passages as they are just a list.

We wish to use a similar naming scheme for general labelled orders. But in general, ancestor sets are not always chains. For arbitrary ordering, we show how to extract chains which can be used as a basis for naming.

**Definition 9.2** Let  $(P, \leq)$  be an ordered set. Let  $S$  be a subset of  $P$ . The function *chainpoints* is given by:

$$\text{chainpoints } S = \{ a : S \mid \forall b : S \bullet a \leq b \text{ or } b \leq a \} \quad \square$$

Chainpoints are those elements of a set  $S$  which have no non-comparable elements in  $S$ . The chainpoints of  $S$  form a chain within  $S$ .

In figure 9.1 (i) all elements in the order  $d$ ,  $b$  and  $a$  are chainpoints, in (ii) elements  $d$  and  $a$  only are chainpoints, and in (iii) only element  $a$  is a chainpoint.

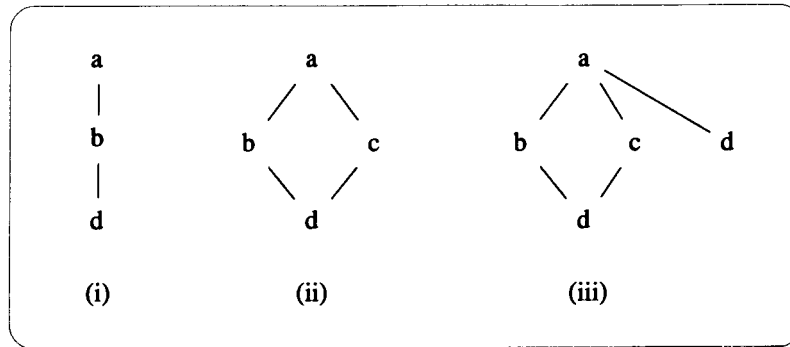


Figure 9.1 Labelled orders which lack unique full chain names

**Definition 9.3** Let  $(P, \leq, L)$  be a labelled order and  $a \in P$ . A labelled sequence  $N$  is a *chain name* for  $a$  if there is a corresponding increasing sequence  $X$  of elements in the chainpoints of  $\uparrow a$  such that:

$$X_1 = a \quad \text{and} \quad \forall i \in L \quad (X_i) = N_i \quad \square$$

Chain names are an extension of the naming scheme suggested for trees because in a tree all elements in an ancestor set are chainpoints. Note that a chain name must only contains the labels of chainpoint elements, but it is not required to contain all chainpoints. If all chainpoints are included in the chain name, then we call this a *full chain name*, which we will often abbreviate to *full name*.

In a tree labelled order full chain names are always unique. However, when the order is not a tree, even full chain names are not always unique. An example of this is shown in figure 9.1(iii). There are two elements with label  $d$ , and both have the same full chain name, the sequence  $\langle d, a \rangle$ .

A labelled order in which each element has a chain name and these chain names are all unique, will be described as having the property of *unique chain names*. We are most interested in labelled orders with this property. In the final section of this chapter on typed order schema we will identify those typed order schema which unfold into labelled orders with unique chain names.

Finally, to support the definition of typed orders in the next section, we need to define an equivalence of labelled orders. Two labelled orders will be equivalent when they have the same order structure and corresponding elements have the same labels.

**Definition 9.4** *Equivalence  $\approx_L$  of labelled orders.* Let  $A$  and  $B$  be labelled orders.  $A \approx_L B$  iff (i)  $\approx_L$  restricted to the underlying ordered sets yields an order isomorphism, and (ii) labels are preserved for corresponding elements. □

## 9.2 Typed orders

A typed order is an extension of a labelled order, in which type information is added and portions of the order with the same type are constrained to be similar. For typed orders we wish the name to also give some indication of its type. We require labels to be associated with only one type, though several labels may be associated with the same type. This is the style of typed order used in previous chapters and is formally defined next.

**Definition 9.5** Let  $(P, \leq, L, \leq_{label})$  be a labelled order and  $T : P \rightarrow Type$  a type function. A *typed labelled order* is the tuple  $(P, \leq, L, \leq_{label}, T)$  where:

$$(i) \exists \varphi : Label \rightarrow Type \bullet$$

$$\varphi \circ L = T$$

$$(ii) \forall x, y : P \bullet$$

$$Tx = Ty \Rightarrow (\downarrow x \triangleleft \leq \triangleright \downarrow x) \approx_t (\downarrow y \triangleleft \leq \triangleright \downarrow y) \quad \square$$

The typed labelled order constraint ensures that elements with the same type are the tops of equivalent labelled orders, that is, have the same order structure and labelling. A consequence is that a label can not appear in the same chain name twice, as this would imply an element has a descendant with the same type; as typed orders are labelled orders which are finite, this is not possible. Our convention for an element's label is to include its type as a prefix. A consequence is that, comparable labels have the same type. Now an equivalence relation for typed orders can be given.

**Definition 9.6** *Equivalence  $\approx_t$  of typed labelled orders.* Let A and B be typed labelled orders.  $A \approx_t B$  iff (i) A and B are labelled order equivalent, and (ii) types are preserved for corresponding elements. We denote corresponding elements  $a \in A$ ,  $b \in B$ , in equivalent typed orders, as  $a \approx_t b$ .  $\square$

Typed labelled orders are equivalent when they are label order equivalent and corresponding elements have the same type. Finally, in this section typed order consistency is defined. This is required for the definition of component composition in section 10.4. Consistent typed orders share a common pool of type definitions; that is, they could have been generated from the same typed order schema.

**Definition 9.7** Let A and B be typed labelled orders. Then A and B are *typed order consistent* iff they satisfy:

$$\forall x:A, y:B \bullet$$

$$T_A x = T_B y \Rightarrow (\downarrow x \triangleleft \leq_A \triangleright \downarrow x) \approx_t (\downarrow y \triangleleft \leq_B \triangleright \downarrow y) \quad \square$$

This constraint ensure that elements from A and B with the same type, are the root of equivalent typed sub-orders. In the remainder, *typed labelled order* will be abbreviated to *typed order*.

### 9.3 Typed tree schema

In this section the schema for generating a typed tree is given first, followed by the definition of the unfold function for converting a schema into a typed tree. The definitions given in this section are not directly used later, their purpose is to provide a restricted but simpler version of the definitions in the next section.

**Definition 9.8** A tuple  $(C_s, T_s)$  is a *tree schema* iff:

$$C_s : \text{Type} \rightarrow \wp \text{Label} \quad (\text{the children mapping})$$

$$T_s : \text{Label} \rightarrow \text{Type} \quad (\text{the type mapping})$$

$$(\text{map } T_s) \circ C_s \text{ is acyclic} \quad \square$$

The  $(C_s, T_s)$  pair is like a record data type; a type is made up of several components, each with their own type. In previous examples of order schema, those types which do not appear on the LHS of the schema, map to an empty set by default. The constraint states there are no type cycles, and so the schema will generate a finite order. Now we can present the partial function for converting a tree schema into a typed tree.

**Definition 9.9** A typed tree  $(P, \leq, L, \leq_{\text{label}}, T)$  is a valid *tree\_unfold* of the tree schema  $(C_s, T_s)$  with root label  $l$  iff:

$$(i) \quad L(\text{root } P) = l$$

$$(ii) \quad \forall x, y : P \bullet$$

$$y \prec x \Leftrightarrow L y \in C_s(T x)$$

$$(iii) \quad T = T_s \circ L \quad \square$$

### 9.4 Typed order schema

Two approaches to extending typed tree schema to typed order schema were discussed in Chapter seven. The approach chosen was, the introduction of a scope function. The scope function just indicates where the overlap of suborders occurs. Also, by using the scope function a natural extension to the naming scheme used for the typed tree was possible. Next define a schema for generating typed orders and its unfold function.



**Definition 9.10** A tuple  $(C_s, S_s, T_s)$  is an *order schema* iff:

- $C_s : \text{Type} \rightarrow \wp \text{Label}$  (the children mapping)
- $S_s : \text{Type} \rightarrow \wp \text{Label}$  (the scope mapping)
- $T_s : \text{Label} \rightarrow \text{Type}$  (the type mapping)

(i)  $(\text{map } T_s) \circ C_s$  is acyclic

Now let  $<_\tau$  be the strict partial order generated by this relation.

(ii)  $\forall x : \text{Label}; a, b : \text{Type} \bullet$

$$x \in C_s a \text{ and } x \in C_s b \text{ and } b <_\tau a \Rightarrow x \notin S_s a$$

(iii)  $\forall x : \text{Label}; a, b, c : \text{Type} \bullet$

$$x \in S_s a \text{ and } x \in S_s c \text{ and } a <_\tau b <_\tau c \Rightarrow x \in S_s b$$

□

Constraint (i) is as for tree schema. Constraint (ii) ensures the scope relation does not induce redundant transitive links in the resultant "order". An example of this is shown in figure 7.3(ii) which we repeat here as figure 9.2.

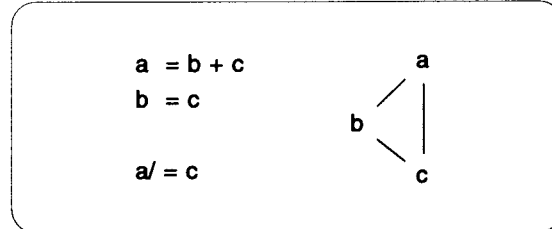


Figure 9.2 An "order" with a redundant transitive covering link

A typed order generated from the schema shown in figure 9.2 must satisfy:  $b$  and  $c$  are covered by  $a$ ,  $c$  is covered by  $b$ , and there is only one element with label  $c$  below  $a$ . The graph which satisfies this constraint, is not the covering relation of an order. The edge between  $a$  and  $c$  should not appear, it should be added when taking the transitive closure of this covering relation to form an order.

Constraint (iii); *scope continuity*, imposes a convexity condition on types scoping the same label. An example of this is shown in Chapter seven figure 7.4. This ensures that orders and suborders with the same type are always equivalent. Now we can present the partial function for converting an order schema into a typed order.

**Definition 9.11** A typed order  $(P, \leq, L, \leq_{label}, T)$  with a unique maximal, root of  $P$ , is a valid *unfold* of the order schema  $(C_s, S_s, T_s)$  with root label  $l$  iff it is the largest (has the most elements) order which satisfies:

(i)  $L(\text{root } P) = l$

(ii)  $\forall x, y : P \bullet$

$$y < x \Leftrightarrow L y \in C_s(T x)$$

(iii)  $T = T_s \circ L$

(iv)  $\forall a, b, c : P \bullet$

$$L a = L b \text{ and } a < c \text{ and } b < c \text{ and } L a \in S_s(T c) \Rightarrow a = b \quad \square$$

We think of unfold as constructing the order from the schema and label. Constraints (i) - (iii) are the same as for *tree\_unfold*. Constraint (iv) gives the conditions where elements must be overlapped; which allows non-trees to be generated, while the requirement the typed order is the largest, ensures no further overlapping is done. Figure 9.3 provides an example of the need to choose the largest order.

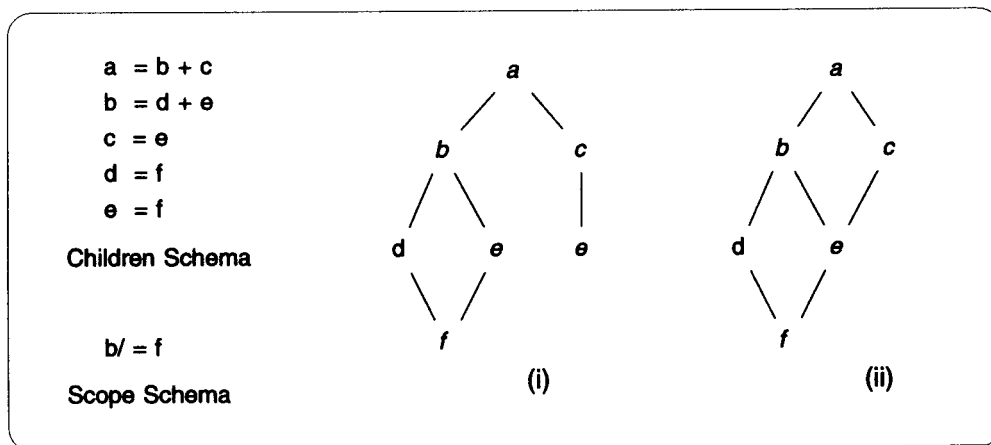


Figure 9.3 Order (i) is larger than order (ii)

The typed order shown in figure 9.3(i) is the unfold of the example schema. The typed order shown in (ii) was a possible unfold of the example schema. It satisfies constraints (i) - (iv), but it is not the largest such order. Element  $e$  has been shared by parents  $b$  and  $c$  resulting in a smaller order, even though there is no  $a/ = e$  in the order schema.

Unfold creates a single rooted order. When applied to a set, unfold generates a collection of disjoint orders.

**Definition 9.12** A typed order  $(P, \leq, L, \leq_{label}, T)$  generated from the order schema  $(C_s, S_s, T_s)$  satisfies the *unique root scope* constraint if and only if:

$$\forall a : P \bullet \frac{}{\{ x \in \uparrow a \mid S_s(Tx) = La \} \text{ is empty or a singleton}} \quad \square$$

Recall, in Chapter seven a desirable property for typed orders was identified: the possession of unique chain names for each element. Figures 7.7 and 7.8 gave examples of typed orders which did not have this property. Using the unique root scope constraint we can now identify those typed orders which do have this property.

**Proposition 9.1** A typed order generated from an order schema has unique full chain names for all elements, if it has unique root scopes. □

In summary, labelled orders then typed orders have been presented. The major concern of this chapter has been the naming of elements. A naming scheme for labelled orders which was a natural extension of the naming scheme for typed trees was given. The key, was mapping orders to chains through chainpoints. The schema representation of the type order was presented in two stages: the typed tree schema, then the typed order schema. The extension of typed trees to typed orders was enabled by the introduction of scope schema. Finally, those typed orders which satisfy unique chain names, and can be generated from a order schema were identified.

---

## Structured Graph Component Formalism

---

This chapter presents the formal definitions of components and component composition which were informally introduced and motivated in Chapter eight. The major concern is to support the construction of large models by a team. Each team member needs significant autonomy in constructing his or her part, yet all parts should contain sufficient information so they can be integrated easily. Components are intended to be such parts. Most of this chapter describes the composition of components.

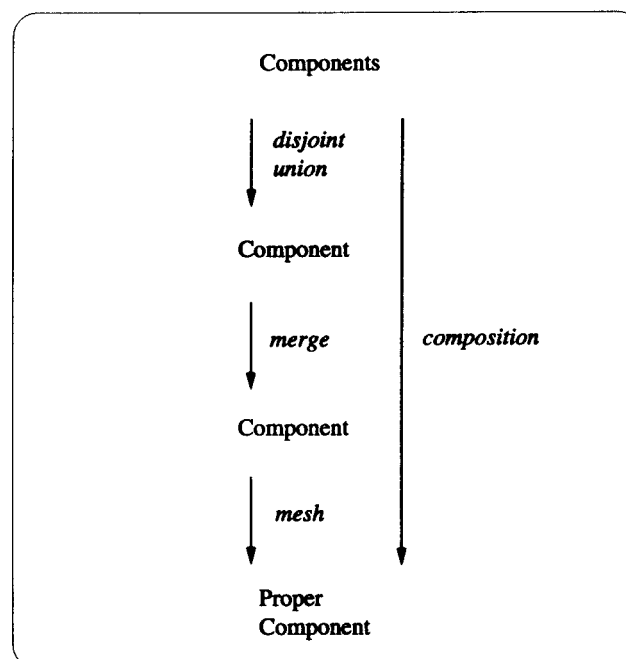


Figure 10.1 An overview of component composition

Figure 10.1 indicates how component composition is structured. The first step is to take the disjoint union of the underlying structured graphs. Second the resulting component is merged: links with the same name are combined. Third the resulting component is meshed: links which mesh are combined. Structures which can be self composed without change are of particular interest; we call them *proper components*.

### 10.1 Pre-components

Pre-components are structured graphs which have been extended in two ways: the addition of model scopes, and the use of a typed link order. Components, defined later, are a subset of pre-components. We present the definition of pre-components in two stages: structured graphs having a model qualifier function, then add a typed link order.

**Definition 10.1** The relation  $M$  is a *model qualifier function* for a structured graph  $(\leq_l, \leq_n, \text{prods}, \text{cons})$  if and only if:

$$M : \text{links} \rightarrow \wp \text{nodes}$$

$$(i) \forall a : \text{links} \bullet$$

$$M a = \downarrow (M a)$$

$$(ii) \forall a : \text{links} \bullet$$

$$M a = \bigcap \{ M b \mid b \in \text{span } a \}$$

$$(iii) \forall a : \text{links} \bullet$$

$$\text{prods } a \cup \text{cons } a \subseteq M a \quad \square$$

The model qualifier function attaches a model qualifier to each link. Model qualifiers limit where a link can appear. The model qualifier is a node downset, constraint (i). A link can only appear as a producer or consumer within the portion of the model given by its model qualifier. Non-leaf link model qualifiers are derivable from their descendent leaf link model qualifiers. Constraint (iii) requires that a link can only appear where all of its leaf links appear.

**Definition 10.2** A pre-component is a structured graph with a model scope function and typed link order. □

## 10.2 The merge operator and components

In this section component merge then components are defined. First, some preliminary definition are required: order parts and an ordering on pre-components.

**Definition 10.3** *Order parts* is the partition of an ordered set into its set of disconnected components (in graph theoretic terms). □

In figure 10.2(i) the link order before merging contains six parts and the node order one part, while after the merge, there are four link order parts. Chapter eight described merging and meshing in terms of individual links. In this chapter they are defined in terms of link order parts, to allow a simpler definition.

**Definition 10.4** Let  $X$  and  $Y$  be pre-components. Let  $\leq_{\text{pre}}$  denote an ordering on pre-components which have a common node order. Then  $Y \leq_{\text{pre}} X$  iff:

$\forall n : \text{nodes} \bullet$

$\forall a : \underline{\text{out}}_Y n, \exists b : \underline{\text{out}}_X n \bullet \uparrow a \approx_{\tau} \uparrow b$

and similarly for inputs □

A pre-component is smaller when its interface links sets are smaller, up to typed order equivalence (see definition 9.6). Since abstract model interfaces are downsets, it is sufficient to compare leaf links. Now the merge operator can be defined.

**Definition 10.5** *Merge* is a function on pre-components. For such a pre-component  $X$ ,  $\text{Merge } X = Y$  where  $Y$  is the smallest pre-component (w.r.t.  $\leq_{\text{pre}}$ ) satisfying:

(i)  $X$  and  $Y$  have the same node order

(ii) there is an onto mapping  $\psi$  from  $\text{links}_X$  to  $\text{links}_Y$ , and each order part  $P$  in  $\text{links}_X$  embeds into  $\text{links}_Y$  where  $P$  and its image are typed order equivalent

(iii)  $\forall a : \text{links}_X \bullet$

$\text{prods}_X a \subseteq \text{prods}_Y(\psi a)$  and  $\text{cons}_X a \subseteq \text{cons}_Y(\psi a)$  and  $M_X a = M_Y(\psi a)$

(iv)  $\forall a, b : \text{leaf links}_X \bullet$

$\uparrow a \approx_{\perp} \uparrow b$  and  $M a = M b \Leftrightarrow \psi a = \psi b$

(v)  $\forall a, b : \text{non-leaf links}_X \bullet$

$\uparrow a \approx_{\perp} \uparrow b$  and  $\psi(\text{span } a) = \psi(\text{span } b) \Leftrightarrow \psi a = \psi b$  □

Constraint (ii) requires a link order is made smaller by merging or embedding order parts, for example in figure 10.2(ii) each order part embeds into the final order. Constraint (iii) and the requirement that  $Y$  is the smallest such pre-component, ensures all link producers and consumers are preserved, while not adding any extra ones beyond those added by an abstract completion. Constraints (iv) and (v) in most cases require, leaf links to be combined when they have the same link name and model scope, and non-leaf links to be combined when they have the same link name and their leaf links merge together. Full upsets are used in (iv) and (v) rather than names, because some leaf links may not have unique full chain names.

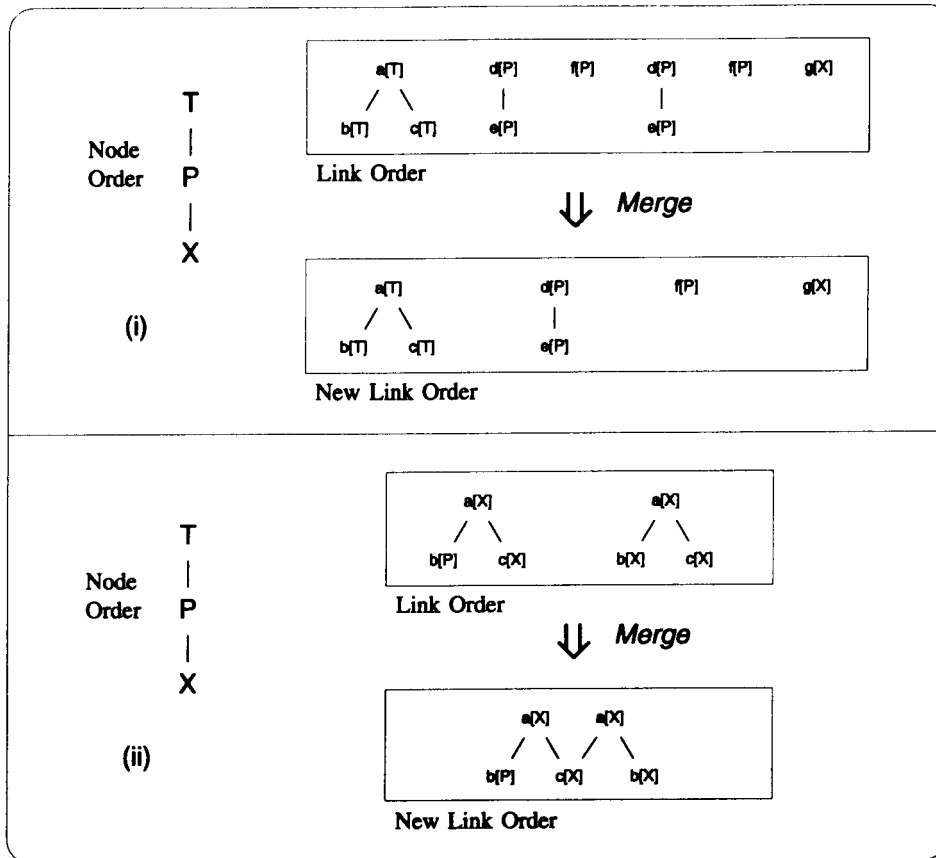


Figure 10.2 Examples of the effect of merge on link orders

Figure 10.2 shows two examples of merge. In (i) the link orders with equivalent leaf links and the same model qualifier;  $d[P]$ ,  $e.d[P]$  and  $f[P]$ , are combined. However, in (ii) the two occurrences of  $b.a[X]$  are not combined, resulting in a link order in which there are two  $a[X]$ 's. This situation was discussed in section 8.6, as a component limitation.

The alternative merge for example 10.2(ii), is to combine no links. The result would be worse, as in addition having two link with name  $a[X]$ , there would also be two leaf links with the name  $c.a[X]$ . This example highlights a limitation of the model qualifier rule used for non-leaf links: a link's model qualifier is the intersection of its descendant link qualifiers. Recall, a model qualifier is a downset of nodes, but in diagrams we only show the highest nodes, so in 10.2(ii)  $b[P]$  rather than  $b[P,X]$  is used. As indicated in section 8.6, a reasonable implementation would highlight this situation of having two links with the same name, allowing users to take some action.

Components were described in Chapters six and eight as being structured graphs with a typed link order and model qualifier. Components also typically have the property of unique link naming. Merge is a function which typically combines links with the same name. Now we can define components.

**Definition 10.6** A *component* is an element of the range of merge. □

Examples of pre-components are shown in figure 10.2, where the link orders prior to merge contain links with the same name.

### 10.3 The component mesh operator

In this section component mesh is defined. The constraints a component meshing function must satisfy are given. Then in the final part of this section a particular meshing function is chosen to be *the mesh* function.

**Definition 10.7** For a given component,  $\leq_{\text{mesh}}$  is an ordering between link downsets given by the reflexive transitive closure of  $\Delta_{\text{mesh}}$  where  $A \Delta_{\text{mesh}} B$  satisfies:

(i)  $A$  is an order part

(ii)  $A$  embeds into  $B$ ; let  $\varphi$  be the mapping from  $A$  to  $B$

(iii)  $\forall a \in A, b \in B \bullet$

$$\varphi a = b \text{ and } a \in \overline{A} \Rightarrow L a \leq_{\text{label}} L b$$

(iv)  $\forall a \in \underline{A}, b \in \underline{B} \bullet$

$$\varphi a = b \Rightarrow ((\text{prods } a \cap \text{prods } b \neq \emptyset) \text{ or } (\text{cons } a \cap \text{cons } b \neq \emptyset))$$

$$\text{and } M a \subseteq M b$$

□

A mesh ordering rather than mesh relation is required because: links are meshable only when they transitively couple as we discussed earlier in section 8.5. Constraints (i) and (ii) ensure that links in the same order part always mesh at the same time, that is, there is no partial meshing of an order part. Constraint (iii) allows links to be mesh ordered when a label prefix relationship holds between the links roots. Constraint (iv) requires that corresponding leaf links couple and are model qualifier ordered.

Figure 10.3 demonstrates why both order parts and downsets are required for the mesh order. The order part with root  $a[P]$  can mesh with either downset  $a.1[T]$  or downset  $a.2[T]$ . For this example, we choose  $a.1[T]$ . If only order parts were used, the presence of this choice would not be noticed.



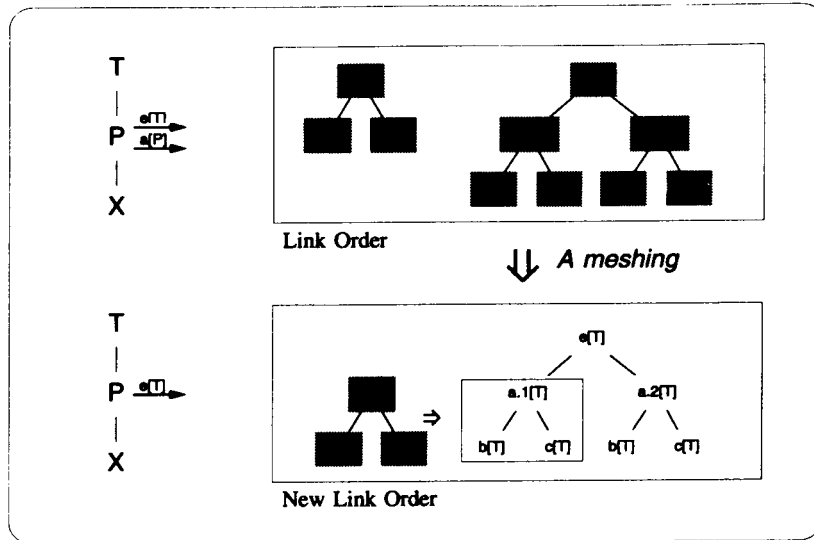


Figure 10.3 A possible result of meshing

**Definition 10.8** Component  $Y$  is a *valid mesh* of component  $X$  when  $Y$  is one of the smallest pre-components (w.r.t.  $\leq_{\text{pre}}$ ) which satisfies:

- (i)  $X$  and  $Y$  have the same node order
- (ii)  $\text{links}_Y \subseteq \text{links}_X$ , and there is an onto mapping  $\psi$  from  $\text{links}_X$  to  $\text{links}_Y$ , and each order part  $P$  in  $\text{links}_X$  embeds into  $\text{links}_Y$  where  $P$  and its image are typed order equivalent

- (iii)  $\forall a : \text{links}_X \bullet$

$$\text{prods}_X a \subseteq \text{prods}_Y (\psi a) \text{ and } \text{cons}_X a \subseteq \text{cons}_Y (\psi a)$$

- (iv)  $\forall A : \text{order\_parts } L \bullet$

$$A \leq_{\text{mesh}} \psi A$$

- (v)  $M_Y = \text{links}_Y \triangleleft M_X$

we call the partition on domain links induced by  $\psi$ , the *link mesh partition*.  $\square$

For a given component there may be many such mesh functions. One such function is the trivial mesh where no consolidation of links is done, leaving the component unchanged. Constraint (ii) requires a link order is made smaller only by merging or embedding order parts. Constraint (iii) and the requirement that  $Y$  is the smallest such pre-component, ensures all link producers and consumers are preserved, while not adding extra ones beyond those added by an abstract completion. Constraint (iv) requires that links are partitioned into groups, which contain order parts that mesh with the maximum order part in the group. Constraint (v) requires that model scopes are preserved for those links left in  $Y$ .

A collection of valid meshes is given in definition 10.8. We wish to choose one that does as much meshing as possible without arbitrary choice, as explained in Chapter eight. Meshing functions will be compared by how they partition links. Next we provide a method for choosing a best partition.

**Definition 10.9** Let  $W = \{\psi_i\}_{i \in I}$  be a set of partitions on some initial set, and let  $\psi_1$  be the trivial partition on this set, so each block in  $\psi_1$  contains one element. The best partition in  $\{\psi_i\}_{i \in I}$  is given by:

$$\text{best } \{\psi_i\}_{i \in I} = \text{maximum (chainpoints } \{\psi_i\}_{i \in I})$$

where,

$$\psi_1 \leq \psi_2 \equiv \forall B_i \in \psi_1 \cdot \exists A_j \in \psi_2 \cdot B_i \subseteq A_j \quad \square$$

Note there is at least one chainpoint; the trivial partition, as this is always the minimum partition. To aid understanding of how link mesh partitions work, an example follows. Figure 10.3 shows four components.

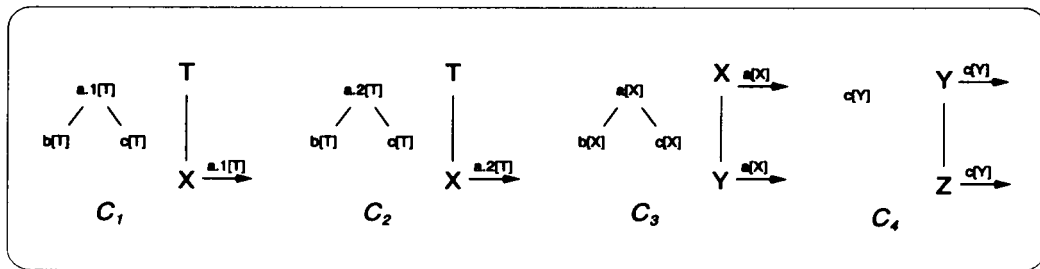


Figure 10.3 Four components

A merge of the four components in figure 10.3 results in the components shown in figure 10.4. Consider all possible mesh coupling of links in the example. Flow  $c[Y]$  can mesh with  $a[X]$  as they are context ordered and they have a common producer  $x$ , link  $a[X]$  could mesh with  $a.1[T]$  or with  $a.2[T]$ .

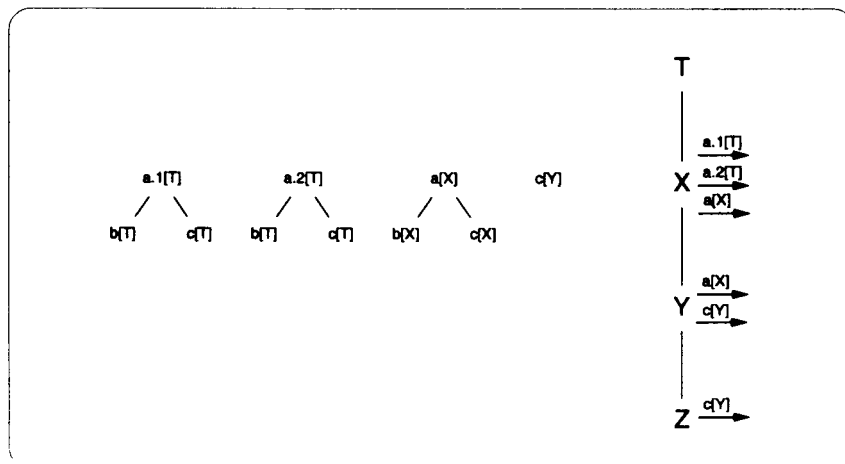


Figure 10.4 The merged four components

The choice of which links to combine is done by considering all possible link mesh partitions of the link order. Figure 10.5 shows the possible link mesh partitions.

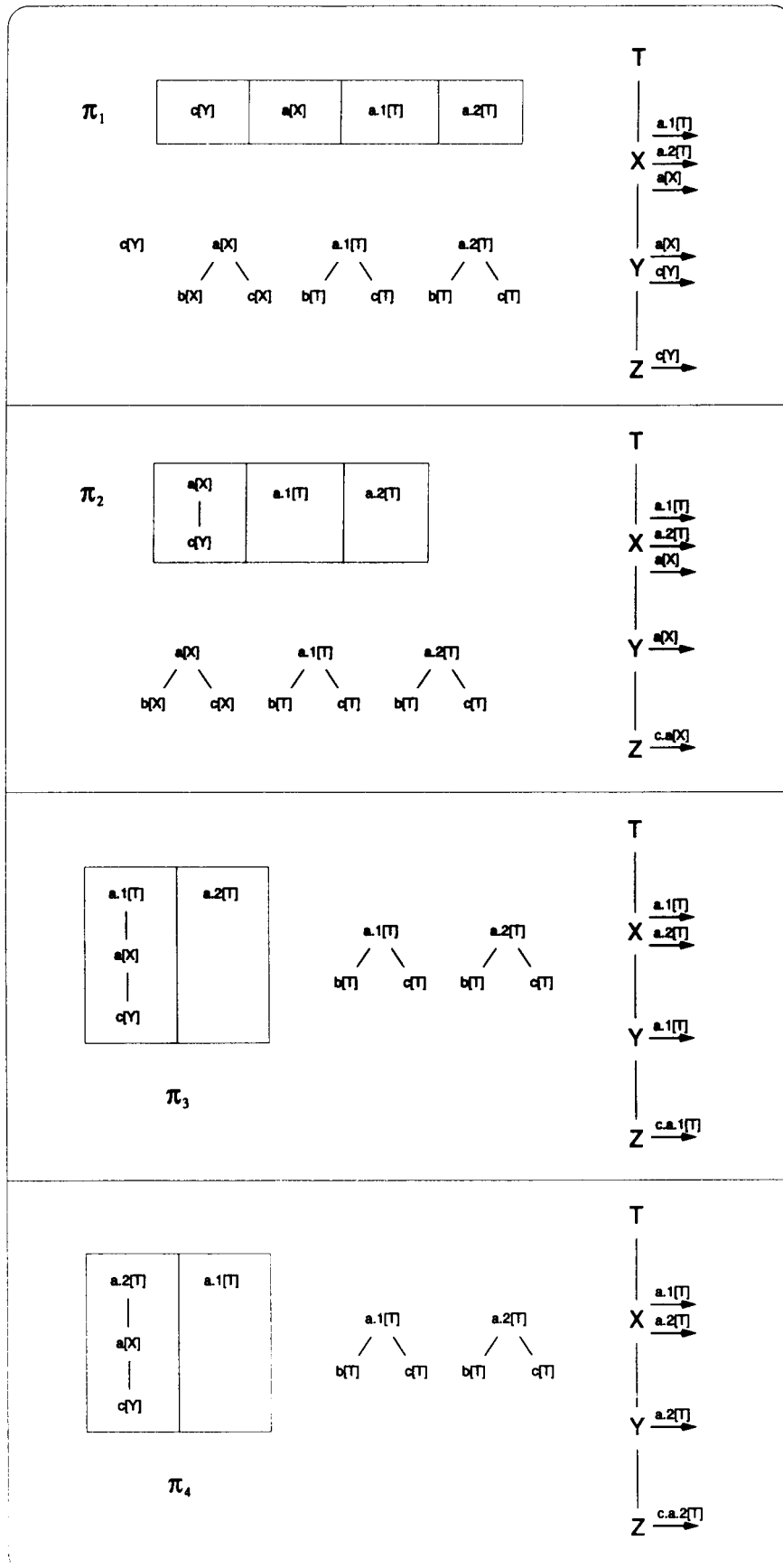


Figure 10.5 Four possible mesh partitions

The link mesh partitions can themselves be ordered, and this is shown in figure 10.6. The highest chainpoint (the highest element with all other elements either above or below it) is  $\pi_2$ , so this is chosen as the best partition.

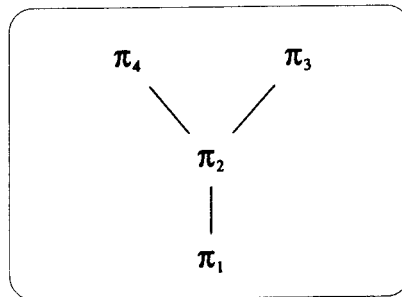


Figure 10.6 An ordering of the four mesh partitions

**Proposition 10.1** Let  $A$  be a component. Then there exists a unique meshing function on  $A$  which has the best link mesh partition. It is denoted by the function *mesh*.  $\square$

#### 10.4 The component composition operator and proper components

**Definition 10.10** Let  $C_1 \dots C_n$  be components which have consistent typed link orders, and have composable node orders. *Component composition* is a function which maps a set of components to a component and is given by:

$$C_1 \circ C_2 \circ \dots \circ C_n = \text{mesh} \circ \text{merge} \circ (C_1 \cup C_2 \dots \cup C_n)$$

where,  $\cup$  is disjoint model union.  $\square$

When the composition operator is applied to a single component, it removes merge and mesh redundant information. For example, if the component contains links with equivalent names and links which mesh without conflict, then self composition will reduce these sets of links to single links. If a library of components is used, a desirable property for composition is: for all views in each component there is an equivalent view (though with an extended context perhaps) in the composed component. Library components should then at a minimum, not contain redundant mesh information which would cause the loss of some views under composition.

**Definition 10.11** A *proper component* is an element of the range of component composition.  $\square$

We have presented definitions of components and their composition. Components were identified as the range of the merge operator, while proper components were identified as the range of the composition operator and as candidates for forming libraries.

---

## Conclusion

---

Two traditional ways of visually organising information are: hierarchies for abstraction, and graphs for arbitrary relations. The major contribution of this thesis is to present a visual formalism which is a generalisation of these two approaches, structured graphs. The treatment is general: arbitrary hierarchies are supported, and compositional techniques are supported with structured graph components. Because structured graphs are built upon such basic visual formalisms, hierarchies and graphs, they should be applicable in many areas.

The rest of this chapter will summarise the thesis contributions, describe existing limitations of this work, and indicate some directions for future work. The thesis has been presented in two parts: structured graphs and structured graph components. Part one is work which is relatively complete and future work is likely to be on tool support, whereas part two is work which requires further extension. Because of this we choose to present the contributions separately for each part.

### 11.1 The contribution of part one: structured graphs

Part one has presented structured graphs, with examples and mathematical definitions. Structured graphs support the abstraction of large and complex networks (of nodes and links), through the use of node and link hierarchies, and automatic derivation of high level views from an underlying network. This is a bottom-up perspective, which contrasts with the top-down perspective of structured analysis.

In Chapter two we showed that with structured graphs, the operations of adding a flow and moving a process only require one user step. Furthermore, as the model increases in size, the benefit of our approach increases. This addresses one of the major deficiencies of current graph based CASE tools: a lack of scalability for user operations which induces inertia in large developments.

The key thing which delivers scalability is the separation of a structured graph model into the three orthogonal parts: the node order, the link order and the network of leaf nodes and links (for a complete model). It is this separation which simplifies the editing operations. The separation has another potential benefit. Alternative node and link hierarchies can be used simultaneously provided they have the same leaves.

In Chapter three examples illustrated the complexity caused by allowing the node hierarchy to be an ordered set rather than just a tree. They also describe both bottom-up

and top-down construction of structured graphs. Also structured graphs may be incomplete models; lowest producers and consumers of a link may not be leaf nodes and there may be links which have only producers or consumers. Input and output interfaces are context free: regardless of what model view a node appears in, it has the same full interface.

The mathematical treatment defined structured graphs, then identified two canonical representations: compact and abstract. The major properties, equivalence of compact and abstract models, closure, and abstract model convexity were presented. Model browsing was defined on the abstract representation, while model editing was defined on the compact form. Proofs for the properties have been given in appendix B.

This thesis includes two implementations of structured graphs. The first is in Gofer a functional language, and the code and sample runs are given in appendix C. The Gofer code implements the compact and abstract completions on arbitrary structured graphs. The second implementation is in C++ and a sample run of this is given in appendix D. The C++ program allows a structured graph to be incrementally built and browsed. The Gofer code (with interpreter) and C++ application are on a floppy disc (Macintosh) that comes with this thesis.

The generic nature of structured graphs (in terms of nodes and links only) makes it application independent and suitable for application to other areas. This is shown by two recent papers in which structured graphs are used in hypermedia work (Lowe and Sifer, 1996) and (Lowe et al., 1996). Also, in the future, structured graphs may be used in the design of file systems which integrate file directories (hierarchies of files) with world wide web page style hyperlinks (arbitrary relations between files).

## **11.2 Limitations of structured graphs**

Structured graphs have a number of limitations. Two major ones are: (i) structured graph composition is not always associative, and (ii) some links may not be viewable. A lack of associativity means that if the collection of updates to a model is done in a different order, the result may be different. Clearly, for a user the order of updates to a model should not matter. However, the failure of associativity only occurs with an unusual arrangement of half flows and so is unlikely to occur often in practice. Also, if the structured graph is, either restricted to node and link trees, or restricted to full flows only, associativity is guaranteed. So associativity is likely to hold for restricted forms of structured graphs which will be sufficient for some application areas.

The other major limitation is that some links may not be viewable if flat model views are used. This occurs when the link order is not a semi-inclusion order. The problem for a user is that a flow may be added between two nodes, but after the add, a different

flow (an ancestor) could appear. This can occur because each node's interface is the maximum of output or input links, and some links (say in a chain) may never be a maximum. Node interfaces could be set manually to overcome this problem, but this extra manual effort will tend to decrease scalability.

### **11.3 Future work for part one**

- The formalism would benefit from a clear classification of structured graphs into groups which satisfy the associative and viewability properties.
- The implementation produced for this thesis is text based. A graphical implementation which followed the visual notation used in Chapter two would allow actual user trials to be conducted to verify the claims of scalability.

### **11.4 The contribution of part two: structured graph components**

In part two the visual notation for structured graph components was given, initially in the context of structured analysis. Components were then defined in two stages: typed order, then components and their composition. Finally formal definitions were given.

Part two addresses the problem of the construction of large structured graphs by a group. It further enhances the scalability of editing. This has been achieved through the introduction of structured graph components and their composition. Each team member constructs only a portion of the overall model, a component. Each component can have its own link name space through the use of flow model scopes. Also, the components can be composed together in a flexible fashion, via meshing, with a single user operation. A flow in a lower component need only be a refinement of a flow in a higher component to mesh. This is far more flexible than the explicit identification by name of which flows from adjoining components should be merged. Also, component composition has been defined to support top-down, bottom-up and layered construction.

The other main contribution has been the design of the typed order and typed order schema. The order schema is a tree schema which has been extended with scope schema. Typed order schema allow arbitrary orders to be generated, so components which include a typed order include an arbitrary ordered set. Typed orders make meshing possible, as meshing was defined in terms of a relationship between typed orders.

The use of link schema becomes necessary for large structured graphs where portions of the link order have a common structure. Having the link schema enhances the scalability of editing by allowing a common structure to be defined only once.

### 11.5 Limitations of structured graph components

One limitation of components is that component merge can introduce non-unique names for non-leaf links. A work around was proposed, an implementation could provide a warning when this situation is detected.

Another limitation is the lack of support for the meshing of components where the link order does not follow the node order. That is, where a lower node may output a parent link and its children, but a higher node outputs only one of the child nodes. This limitation was not directly presented earlier, though it is implicit in the unique leaf name violation case discussed in Chapter six. This limitation is unlikely to be a problem for structured analysis applications, because top-down development makes this situation less likely. However, for applications where development is not top-down and especially where a link may have multiple producers, this could be a significant limitation.

### 11.6 Future work for part two

- Extend both the typed order and meshing to produce a definition of components without the limitations identified in the previous section.
- The design of component libraries. Determine the extent to which component composition is associative, and propose component variations for which composition is associative.
- An implementation is needed to give confidence to further validate the mathematical descriptions and to allow experimentation. An earlier version of components was implemented in Gofer, but this has not been included in this thesis as it was based on an earlier version of structured graphs. In fact, it was the limitations of this earlier component implementation which motivated the final improvement of structured graphs in part one to fully support half flows. Half flows were required in components to support an explicit interface (the link inputs and outputs of a components root node).
- The introduction of component schema. This is likely to require further work on typed order schema. The existing scope based schema has a top down orientation which does not seem suitable for a typed node order. For example, a scope schema in a higher component could distort a lower component (by merging some of its internal nodes) resulting in unintended effects on the model.

### 11.7 A component schema example

Component schema allow the generation of typed models. That is, submodels with the same type should have the same internal structure. Most of the properties required for



typed models have already been met by components. The remaining property is the preservation of typed model substructure, that is, preservation of submodels. To motivate the need for submodels consider the example shown in figure 11.1.

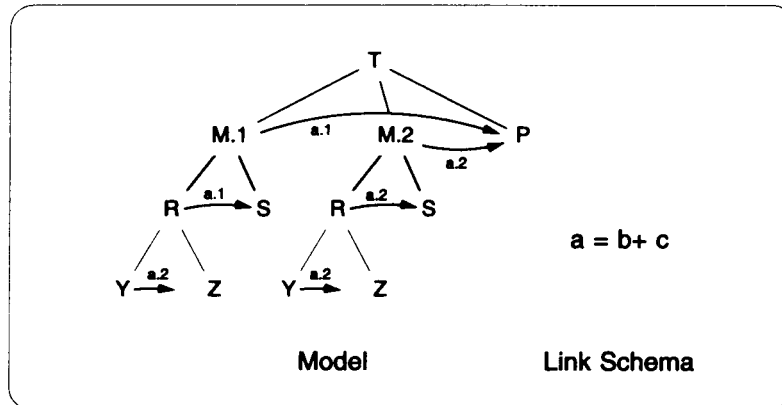


Figure 11.1: An untyped model

Note in addition to flow typing the model in figure 11.1 also uses a typed node order. This model could be built as the composition of three components, with root nodes of *M.1*, *M.2* and *T*. Components *M.1* and *M.2* have a common internal structure; if flows *a.1* and *a.2* were replaced by *a*. A typed model factors out this common structure through the use of schema. The schema used to construct a typed model will be component schema. An example is shown in figure 11.2.

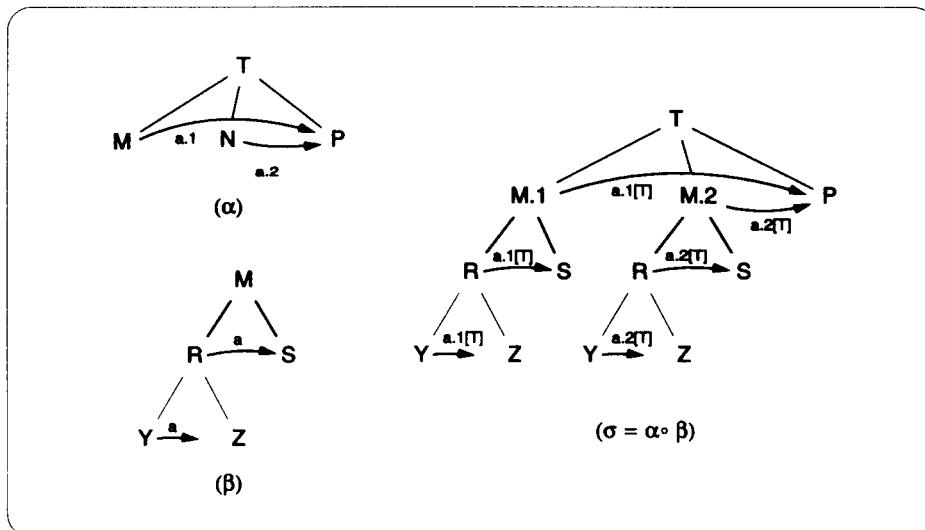


Figure 11.2: A typed model

Two component schema are shown in figure 11.2, for node types *T* and *M*. These are instantiated in the final unfolded model as *T*, *M.1* and *M.2*. Full use has been made of component composition here: flow *a.2[T]* meshes with *a[M.1]*, and *a.2[T]* mesh with *a[M.2]*. The additional restriction for a typed model schema is that component schema share the same flow and node schema, and rather than a library of components there is a library of component schema.

## 11.8 Final Discussion

The approach of this thesis has been, to aim for a formalism which is as general as possible, though many applications may only require a restricted version of the formalism. This approach has been taken because we believe it yields a cleaner, more elegant mathematical treatment, rather than a formalism which is complicated by the internalisation of constraints for some arbitrary domain of application (say structured analysis). Also, abstracting away from any given application domain makes the formalism more widely applicable. Examples of this are, support for links with multiple producers, and the support for bottom-up construction which are not part of traditional structured analysis.

This generality comes at some cost: the formalism becomes further removed from any given application and can initially be more difficult to comprehend. In this thesis, in addition to presenting the formalism we have been at pains to show the formalism in action in an application area (structured analysis).

In summary, this thesis has presented the structured graph visual formalism, which allows the scalable browsing and editing of large graph based models. This has been demonstrated for structured analysis models, while the generic nature of the formalism makes it suitable for possible application to other areas.

---

## Bibliography

---

- Adler, M. (1988): *An Algebra for Data Flow Diagram Process Decomposition*, IEEE Transactions on Software Engineering, Vol. 14, No. 2, Feb. 1988, 169-183.
- Arndt, T. and GUERCIO A. (1992): *Decomposition of Data Flow Diagrams*, Proceedings of the Fourth International Conf. on Software Engineering and Knowledge Engineering, IEEE Computer Society Press, 1992, 560-566.
- Beeck M. von der (1993): *Enhancing Structured Analysis by Timed Statecharts for Real-Time and Concurrency Specification*, Elsevier Science B.V. (North-Holland) 1993, 369-381.
- Berge C. (1962): *The Theory of Graphs and its applications*, translated by Alison Doig, London: Methuen & Co Ltd, New York: John Wiley & Sons Inc, 1962.
- Berge C. (1973): *Graphs and Hypergraphs*, North-Holland, Amsterdam , 1973.
- Boloix G., Sorenson P.G. and Tremblay (1992): *Transformations using a meta-system approach to software development*, IEE and BCS, Software Engineering Journal, Nov. 1992, 425-437.
- Butler G., Grogono P., Shinghal R., and Tjandra I. (1995): *Retrieving information from data flow diagrams*, Proceedings of the 2nd Working Conference on Reverse Engineering, Toronto, Canada, Pub. IEEE, 1995, 22-29.
- Cadre (1990): *Teamwork® User Menus User's Guide*, Release 4.0, Cadre Technologies Inc., 1990.
- Carre, B.(1979): *Graphs and Networks*, Oxford applied mathematics and computer science series, Oxford University Press, 1979.
- Canfora, G. et al (1992): *Data Flow Diagrams: Reverse Engineering Production and Animation*, Conference on Software Maintenance, IEEE Comp. Soc. Press, 1992, 366-375.
- Champeaux D., Constantine L., Jacobson I., Mellor S., Ward P. and Yourdon E. (1990): *PANEL: Structured Analysis and Object Oriented Analysis*, ECOOP/OOPSLA '90 Proceedings, Oct. 21-25, 1990, 135-139.
- Chen, M.J. and Chung, C.C. (1991): *Restructuring operations for data-flow diagrams*, Software Engineering Journal, IEE and BCS, Vol. 6, No. 4, July 1991,

181-195.

- Cimitile, A. and Visaggio, G. (1994): *A formalism for structured planning of a software project*, International Journal of Software Engineering and Knowledge Engineering, World Scientific, Vol. 4 No. 2, 1994, 277-300.
- Conklin J. (1987): *Hypertext: A Survey and Introduction*, IEEE Computer, Vol. 20, No. 9, Sept. 1987, 17-41.
- Consens M., Mendelzon A., and Ryman A. (1991): *Visualizing and Querying Software Structures*, Proceeding of the Fourteenth International Conference on Software Engineering, 1991, 138-156.
- Davey, B.A. and Priestley, H.A. (1990): *Introduction to Lattices and Order*, Cambridge University Press, 1990.
- Davis A.M. (1990): *Software Requirements - Analysis and Specification*, Prentice-Hall, 1990.
- Dean T.R., and Cordy J.R. (1995): *A Syntactic Theory of Software Architecture*, IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995, 302-313.
- Demarco T. (1979): *Structured Analysis and System Specification*, Prentice-Hall, 1979.
- Di Battista G., Eades P., Tamassia R. and Tollis I.G. (1994): *Algorithms for Drawing Graphs: an Annotated Bibliography*, World Wide Web at <http://www.uni-passau.de/agenda/gd95/biblio.html>, June 1994.
- Edwards, H.E. and Munro, M. (1993): *Abstracting the Logical Processing Life Cycle for Entities and the RECAST Method*, Proceeding of the Conference on Software Maintenance, Pub. IEEE, 1993, 162-171.
- Feiner S. (1988): *Seeing the Forest for the Trees: Hierarchical Display of Hypertext Structure*, Proceedings of the Conference on Office Information Systems, IEEE Comp. Soc. Press, March 1988, 205-212.
- Feng Q-W., Cohen R. and Eades P. (1991): *How to draw a planar clustered graph*, Computing and Combinatorics, Lecture Notes in Computer Science, Springer, Vol. 959, 1991, 21-30.
- Forte G. and McCulley K. (1991): *CASE Outlook: Guide to Products and Services*, CASE Consulting Group, Lake Oswego, Ore., 1991.
- France R.B. (1993): *A predicative basis for structured analysis specification tools*, Information and Software Technology, Butterworth-Heinemann, Vol. 35, No. 2,

- Feb. 1993, 67-77.
- Fraser M.D., Kumar K. and Vaishnavi V.K. (1991): *Informal and Formal Requirements Specification Languages: Bridging the Gap*, IEEE transactions on Software Engineering, Vol. 17, No.5, May 1991, 454-466.
- Fuggetta, A. (1993): *A Classification of CASE Technology*, IEEE Computer, Dec. 1993, 25-38.
- Fuggetta, A., Ghezzi C., Mandrioli D. and Morzenti A. (1993): *Executable Specifications with Data-Flow Diagrams*, Software-Practice and Experience, Vol. 23, No. 6, June 1993, 629-653.
- Gane C. and Sarson T. (1979): *Structured Systems Analysis: Tools and Techniques*, Englewood Cliffs, NJ: Prentice-Hall, 1979.
- Gomaa H. (1986): *Software development for Real-Time Systems Specification*, Communications of the ACM, Vol. 29, No. 7, 1986.
- Guindon, R., Krasner H. and Curtis B. (1987): *Breakdowns and processes during the early activities of software design by professionals*. In G. Olson, E. Soloway and S. Sheppard (Eds.), "Empirical Studies of Programmers", Second Workshop, Ablex Publishing.
- Guindon, R. (1990a): *Designing the design process: Exploiting opportunistic thoughts*, Human Computer Interaction, 5, 305-344.
- Guindon, R. (1990b): *Knowledge exploited by experts during software design*, International Journal of Man-Machine Studies, 33, 279-304.
- Guindon, R. (1992): *Requirements and Design of Design Vision, An Object-Oriented Graphical Interface to an Intelligent Software Design Assistant*, ACM Conference on Human Factors in Computing Systems - CHI '92, 499-506.
- Harel, D. (1988): *On Visual Formalisms*, Communications of the ACM, Vol. 31, No. 5, May 1988, 171-187.
- Hashimoto K. (1987) *System Analysis by Extended Data Flow Diagram with Events and Timing*, Proceedings of the International Computer Software & Applications Conference (COMPSAC), October 7-9, Tokyo, Pub. IEEE, 1987, 117-123.
- Hatley D.J. and Pirbhai I.A. (1987): *Strategies for Real-Time System Specification*, Dorset House, 1987.
- Johnson, J.A. et al (1993): *ACE: Building Interactive Graphical Applications*, Communications of the ACM, April 1993, Vol. 36, No. 4, 41-55.

- Kim J. and Lerch F.J. (1992): *Towards a Model of Cognitive Process in Logical Design: Comparing Object-Oriented and Traditional Functional Decomposition Software Methodologies*, Proceeding of CHI 1992, May 3-7, ACM, 489-498.
- Kimelman D., Leban B., Roth T., and Zernik D. (1995): *Dynamic Graph Abstraction for Effective Software Visualisation*, Australian Computer Journal, Vol. 27, No. 4, November 1995, 129-137.
- Khan, J.I. and Miyamoto, I. (1993): *Integrating Abstraction Flexibility with Diverse Program Perspectives*, IEEE International Computer Software and Applications Conference 1993, 186-192.
- Khan, J.I. (1994): Design extraction by adiabatic multi-perspective abstraction, Proceeding of the Conference on Software Maintenance, Victoria, Canada, 1994, IEEE, 191-200.
- Kodosky J., MacCrisken J. and Rymar (1991): *Visual Programming Using Structured Data Flow*, Proceedings of the IEEE Workshop on Visual Languages, 1991, 34-39.
- Kuo F.Y. (1994): *A Methodology for Deriving an Entity-Relationship Model Based on a Data Flow Diagram*, Journal of Systems Software, 24, 1994, 139-154.
- Lee, S. and Carver D. (1990): *The Construction of an Object Oriented Specification Model*, IEEE, proceedings of 1990 Southeastcon, 384-389.
- Levene M. and Loizou G. (1995): *A Graph-Based Model and its Ramifications*, IEEE Transactions on Software Engineering, Vol. 7, No. 5, October 1995, 809-823.
- Levene, M. and Poullovovassilis A. (1990): *The Hypernode Model and its associated Query Language*, Proceeding of the 5th Jerusalem Conference on Information Technology, Jerusalem, October 1990, IEEE Press, 520-530.
- Liu S. (1993): *A formal specification method based on data flow analysis*, Journal of Systems Software, Elsevier Science, Vol. 21, 1993, 141-149.
- Lowe D. and Sifer M. (1996): *Refining the MATILDA Multimedia Authoring Framework with a Visual Formalism*, Proceedings of the IEEE International Conference on Multimedia Computing and Systems, Hiroshima, June 1996, Pub. IEEE Computer Society, 291 - 294.
- Lowe D., Ginige A., Sifer M. and Potter J. (1996): *The MATILDA Data Model and its Implications*, Proceedings of the Third International Conference on Multimedia Modelling, Toulouse, France, Nov. 1996, Pub. Word Scientific Press.
- Nielsen J. (1990): *Through Hypertext*, Communications of the ACM, Vol. 33, No. 3, March 1990, 297-310.

- Nosek J. and Baram G. and Steinberg G. (1992): *Ease of learning and using a CASE software tool: an empirical evaluation*, Proceeding of the ACM/SIGCPR Conference, 1992, 75-80.
- O'Hare, A.B. and Troan, E.W. (1994): *RE-analyzer: from source code to structured analysis*, IBM Systems Journal Vol. 33, No. 1, 1994, 110-130.
- Olive, A. (1983): *Information derivability analysis and consistency checking*, Journal of Systems and Software, Vol. 15, 1983, 185-191.
- Peters L. (1988): *Advanced Structured Analysis and Design*, Englewood Cliffs, NJ: Prentic-Hall, 1988.
- Potter J. and Sifer, M. (1988): *Formal Specification of a Structured Planning System*, (abstract only) Proceedings of the Australian Software Engineering Conference 88, Canberra, May 1988, 77.
- Poulovovassilis, A. and Levene M. (1994): *A Nested-Graph Model for the Representation and Manipulation of Complex Objects*, ACM Trans. Information Systems, Vol. 12, 1994, 35-68.
- Randell G.P. (1990): *Translating Data Flow Diagrams into Z (and visa versa)*, Technical Report 90019, Royal Signals Radar Establishment, London, 1990.
- Read M.C. and Marlin C.D. (1996): *Generating Direct Manipulation Program Editors within the MultiView Programming Environment*, Viewpoints 96: International Workshop on Multiple Perspectives in Software Developments, San Francisco, California, October, 1996, 232-236.
- Richter G. and Maffeo B. (1993): *Towards a Rigorous Interpretation of ESML - Extended Systems Modeling Language*, IEEE trans. on Soft. Eng., Vol. 19, No. 2, Feb. 1993, 165-180.
- Ross D.T. and Brackett J.W. (1976): *An approach to structured analysis*, Computing Decisions, Vol. 8, No. 9, Sept. 1976, 40-44.
- Ross D.T. (1977): *Structured Analysis (SA): A Language for Communicating ideas*, IEEE Transaction on Software Engineering, Vol. SE-3, No. 1, 1977, 16-24.
- Ross D.T., and K.E. Schoman Jr (1977): *Structured Analysis for Requirements Definition*, IEEE Transaction on Software Engineering, Vol. SE-3, No. 1, 1977, 6-15.
- Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W. (1991): *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

- Sarkar M. and Brown M.H. (1994): *Graphical Fisheye Views*, Communications of the ACM, December 1994, Vol. 37, No. 12, 73-84.
- Sifer, M.J (1988): *Structured Planning*, Masters thesis, School of Computing Sciences, University of Technology Sydney.
- Sifer, M. and Potter, J. (1995): *Scalability for Graph Based CASE Tools*, International Journal of Software Engineering and Knowledge Engineering, World Scientific, Vol. 5, No. 3, Sept. 1995, 347-365.
- Sifer, M. and Potter, J. (1996): *Structured Graphs: a visual formalism for scalable CASE tools*, Australian Computer Journal, Pub. Australian Computer Society, Feb. 1996, Vol. 28, No. 1, 13-26. *Errata*, May 1996, Vol. 28, No. 2, 71.
- Shaw M., Deline R., and Zelesnik G. (1996): *Abstractions and Implementations for Architectural Connections*, Proceedings of the third International Workshop on Configurable Distributed Systems, Annapolis, USA, Pub. IEEE, 1996, 2-10.
- Shaw M., Deline R., Klien D., Ross T., Young D., and Zelesnik G. (1995): *Abstractions for Software Architecture and Tools to Support Them*, IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995, 302-313
- Shoval P. (1988): *ADISSA: architectural design of information systems based on structured analysis*, Information Systems, Vol. 13, No. 2, 1988.
- Storey, M. D. and Muller H. A. (1995): *Manipulating and Documenting Software Structures Using SHriMP Views*, Proceeding of the International Conference on Software Maintenance, Nice, France, IEEE Comp. Soc. Press, Oct. 1995, 275-284.
- Sugiyama K. and Misue K. (1991): *Visualization of structural information: Automatic drawing of compound digraphs*, IEEE Transactions on Systems, Man and Cybernetics, Vol. 21, No. 4, 1991, 876-892.
- Tao, Y. and Kung, C. (1991): *Formal definition and verification of data flow diagrams*, Journal of Systems and Software, 1991, Vol. 16, No. 1, 29-36.
- Teorey T., Wei G., Bolton D. and Koenig J.A. (1989): *ER Model Clustering as an Aid for User Communication and Documentation in Database Design*, Communications of the ACM, Vol. 32, No. 8, August 1989, 975-987.
- Teorey T., Yang D. and Fry J. (1986): *A Logical Design Methodology for Relational Databases using the Extended Entity-Relationship Model*, ACM Comp. Surv., Vol. 18, 1986, 197-222.
- Tse T.H. (1994): *The Application of Prolog to Structured Design*, Software-Practice



- and Experience, Pub. John Wiley & Sons, Vol. 24, No. 7, July 1994, 659-676.
- Tse, T.H. (1991): *A Unifying Framework for Structured Analysis and Design Models*, Cambridge Tracts in Theoretical Computer Science 11, Cambridge University Press, 1991.
- Tse, T.H. and Pong L. (1989): *Towards a Formal Foundation for DeMarco Data Flow Diagrams*, Computer Journal, Vol. 32, 1989, 1-11.
- Ward P.T. (1986): *The Transformational Schema: An Extension of the Data Flow Diagram to Represent Control and Timing*, IEEE trans. on Soft. Eng., Vol SE-12, No. 2, Feb. 1986, 198-210.
- Ward P.T. (1989): *How to Intergrate Object\_oriented Orientation with Structured Analysis and Design*, IEEE Software, March 1989, 74-82.
- Wilson, D. and Sifer, M. (1990): *Structured planning: deriving project views*, Software Engineering Journal, IEE and BCS, Vol. 5, No. 2, March 1990, 138-148.
- Wilson, D. and Sifer, M. (1988): *Structured planning: project views*, Software Engineering Journal, IEE and BCS, Vol. 3, No. 4, July 1988, 134 - 140.
- Yeh, R., and P. Zave (1980): *Specifying Software Requirements*, Proceeding of the IEEE, Vol. 68, No. 9, Sept. 1980, 1077-1085.
- Youdon, E., and L.L. Constantine (1979): *Structured Design*, Prentice-Hall, Englewood Cliffs NJ, 1979.

---

## Glossary

---

This thesis introduced a number of new terms, to aid the reader we collated definitions for the most important ones here.

### A.1 Structured Graph Terms

*Structured graph / model:* a hierarchical network structure comprising: a node hierarchy, a link hierarchy, and producer/consumer relations between nodes and links.

*Underlying network:* a normalised representation of a structured graph from which producer/consumer relations can be derived.

*Noncomparable nodes / Outside:* nodes not ordered by the node hierarchy are noncomparable. A node is outside of any other node if they are noncomparable.

*Full flows:* represent a producer/consumer relation formed by a given link between two nodes.

*Half flows:* the subset of producer and consumer relations not participating in full flows.

*Node interface:* the set of input and output links for a node. In a model these sets include all participating leaf links and also their higher level summaries.

*Node view:* a cross-section of the node hierarchy.

*View / Model view:* a restriction of a structured graph to a node view with summarised node interfaces.

*Net interface:* a node interface which satisfies the following two rules. Firstly, the node produces a given link when one of its descendant nodes produces the link and an outside leaf node consumes the link. Secondly, the node consumes a given link when one of its descendant nodes consumes the link and an outside leaf node produces the link.

*Context free:* a property of node interfaces in a model view. Regardless of which model view a node appears in, its interface remains the same when context free.

*Abstract model:* a model with all possible derived flows and with maximum net interfaces.

*Compact model:* a model with no derivable flows and with minimum net interfaces.

## A.2 Structured Graph Component Terms

**Component:** an abstract model with an extended link order. The link order is a typed order. Each link has an associated model qualifier. Further, each link has a unique chain name.

**Model qualifier:** a set of nodes which indicates the portion of the model in which a link can appear. Component link names usually contain only the highest model qualify nodes.

**Chain points:** if we take a subset of a typed order, then the chain points are those elements in the subset which are comparable with all other elements in the subset.

**Chain name:** a typed order element name constructed as a sequence of those ancestor elements which are chainpoints.

**Context:** the context of a link in a component is given by: the suborder formed by the link and its ancestors, and the link's model qualifier.

**Context order:** link a is less than or equal to link b (context order) when there is an embedding from a and its ancestor into b and its ancestors, and a's model qualifier is less than or equal to b's model qualifier (node view order).

**Couple:** two links in a component couple when they have a common producer or consumer node.

**Meshable:** two links in a component are meshable when they couple and are context comparable.

**Component link name:** the name a link has in a component model view. It includes the link's chain name from its link order and its model qualifier.

---

## Structured Graph Proofs

---

The theorems and corollaries given in Chapters four and five are proved in the first two sections. Section three also includes some additional lemmas to assist with the proofs.

### B.1 Chapter four proof

**Theorem 4.1** Let  $P$  be a finite ordered set. Let  $V$  be the set of all views on  $P$ . The function  $span$  is a lattice homomorphism from  $V$  to  $V$ . It partitions the set of all views into equivalence classes (views having the same span), where each equivalence class has a maximum element (view).  $\square$

There is an isomorphism between views (anti-chains) and downsets, as a downset can be represented by its maximal elements (which form a view). So the above theorem will be proved using the downset representation of views. Thus rather than working with the set of all views on  $P$ , we will work with the set of all downsets of  $P$ .

**Proof.** First recall the definition of span, for a subset  $S$  of  $P$ :

$$span S = \underline{\downarrow S} \quad (1)$$

however, if  $S$  is already a downset:

$$span S = \underline{S} \quad (2)$$

To show  $span$  is a lattice homomorphism, we must show it preserves join and meet.

For all  $X, Y \in \text{downsets of } P$  we require:

$$\underline{X \vee Y} = \underline{X \vee Y} \quad (3)$$

$$\underline{X \wedge Y} = \underline{X \wedge Y} \quad (4)$$

For the downsets of  $P$ , join and meet are give by set intersection and union, so we must show:

$$\underline{X \cup Y} = \underline{X \cup Y} \quad (5)$$

$$\underline{X \cap Y} = \underline{X \cap Y} \quad (6)$$

Taking (5) first:

$$\begin{aligned}
 \underline{X \cup Y} &= (X \cup Y) \cap \underline{P} && \text{(by lemma B.1, which appears in Section B.3)} \\
 &= (X \cap \underline{P}) \cup (Y \cap \underline{P}) \\
 &= \underline{X \cup Y} && \text{(by lemma B.1)} \quad (7)
 \end{aligned}$$

Equation (6) is proved in the same way:

$$\begin{aligned}
 \underline{X \cap Y} &= (X \cap Y) \cap \underline{P} && \text{(by lemma B.1)} \\
 &= (X \cap \underline{P}) \cap (Y \cap \underline{P}) \\
 &= \underline{X \cap Y} && \text{(by lemma B.1)} \quad (8) \quad \square
 \end{aligned}$$

## B.2 Chapter five proofs

The abstract and compact operators given in Chapter five use symmetrical definitions for producer and consumer parts. So in these proofs it will be sufficient to show the properties hold for the producer part. First, some properties for structured graphs disregarding the link order are established. Then the Chapter five proofs are given.

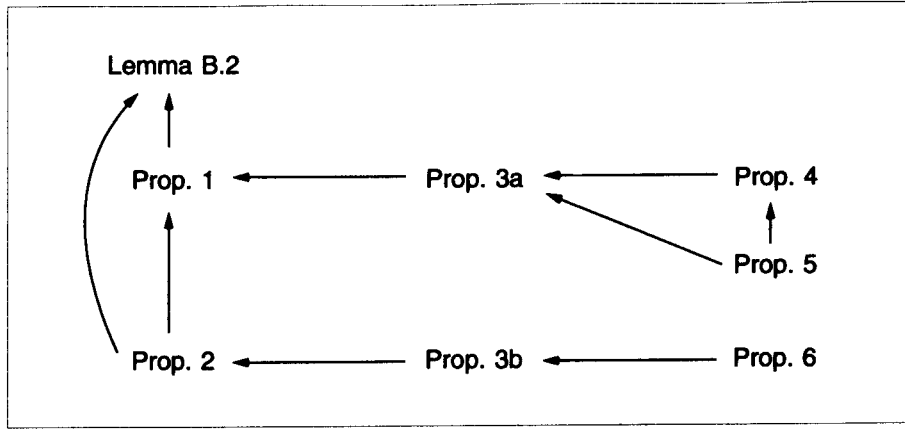
In the following, the properties of leaf link structured graphs are all for arbitrary  $l: \text{Link}$  e.g. (property 2):  $\underline{\text{prods}_C l} = \underline{\text{prods } l}$ . For convenience we omit the  $l$ .

1.  $\underline{\text{prods}_A} = \underline{\text{prods}}$
2.  $\underline{\text{prods}_C} = \underline{\text{prods}}$
- 3(a)  $\text{prods}_{F \circ A} = \text{prods}_F$
- (b)  $\text{prods}_{F \circ C} = \text{prods}_F$
4.  $\text{prods}_{A \circ A} = \text{prods}_A$  ( $A_n \circ A_n = A_n$ )
5.  $\text{prods}_{C \circ A} = \text{prods}_C$  ( $C_n \circ A_n = C_n$ )
6.  $\text{prods}_{A \circ C} = \text{prods}_A$  ( $A_n \circ C_n = A_n$ )

An additional property of structured graphs used is:

$$7. \quad C_l \circ A_l = C_l$$

The dependencies between the proofs of these properties are shown next.



Property 1.

$$\begin{aligned} \text{prods}_F &= \uparrow(\underline{\text{prods}} - \underline{\text{cons}}^{\text{ul}}) - \underline{\text{cons}}^{\text{ul}} \\ &\subseteq \uparrow \underline{\text{prods}} \text{ which equals } \uparrow \text{prods} \end{aligned}$$

So  $\underline{\text{prods}} \cup \text{prods}_F = \underline{\text{prods}}$  (by lemma B.2)

$$\begin{aligned} \text{prods}_A &= \uparrow(\underline{\text{prods}} \cup \text{prods}_F) \\ &= \underline{\text{prods}} \cup \text{prods}_F \\ &= \underline{\text{prods}} \end{aligned}$$

(by B.1 1)

□

Property 2.

$$\overline{\text{prods}_A - \text{prods}_F} \subseteq \overline{\text{prods}_A} \subseteq \uparrow \overline{\text{prods}_A}$$

So  $\underline{\overline{\text{prods}_A} \cup (\overline{\text{prods}_A} - \text{prods}_F)} = \underline{\overline{\text{prods}_A}}$  (by lemma B.2)

$$= \underline{\overline{\text{prods}_A}}$$

That is,  $\underline{\text{prods}_C} = \underline{\overline{\text{prods}_A}}$

$$= \underline{\text{prods}}$$

(by property 1)

□

Property 3a.

$$\begin{aligned} \text{prods}_{F \circ A} &= \uparrow(\underline{\text{prods}_A} - \underline{\text{cons}_A}^{\text{ul}}) - \underline{\text{cons}_A}^{\text{ul}} && \text{(definition of } \text{prods}_F) \\ &= \uparrow(\underline{\text{prods}} - \underline{\text{cons}}^{\text{ul}}) - \underline{\text{cons}}^{\text{ul}} && \text{(by property 1)} \\ &= \text{prods}_F && \text{(definition of } \text{prods}_F) \end{aligned}$$

□

Property 3b.

$$\begin{aligned} \text{prods}_{F \circ C} &= \uparrow(\underline{\text{prods}_C} - \underline{\text{cons}_C}^{\text{ul}}) - \underline{\text{cons}_C}^{\text{ul}} && \text{(definition of } \text{prods}_F) \\ &= \uparrow(\underline{\text{prods}} - \underline{\text{cons}}^{\text{ul}}) - \underline{\text{cons}}^{\text{ul}} && \text{(by property 2)} \\ &= \text{prods}_F && \text{(definition of } \text{prods}_F) \end{aligned}$$

□

Property 4.

$$\begin{aligned}
\text{prods}_{A \circ A} &= \downarrow(\text{prods}_A \cup \text{prods}_{F \circ A}) && \text{(definition of } \text{prods}_A\text{)} \\
&= \downarrow(\text{prods}_A \cup \text{prods}_F) && \text{(property 3a)} \\
&= \downarrow(\downarrow(\text{prods} \cup \text{prods}_F) \cup \text{prods}_F) && \text{(definition of } \text{prods}_A\text{)} \\
&= \downarrow(\text{prods} \cup \text{prods}_F) && \text{since } \text{prods}_F \subseteq \downarrow(\text{prods} \cup \text{prods}_F) \\
&= \text{prods}_A && \text{(definition of } \text{prods}_A\text{)} \quad \square
\end{aligned}$$

Property 5.

$$\begin{aligned}
\text{prods}_{C \circ A} &= \underline{\text{prods}_{A \circ A}} \cup \overline{(\text{prods}_{A \circ A} - \text{prods}_{F \circ A})} && \text{(definition of } \text{prods}_C\text{)} \\
&= \underline{\text{prods}_A} \cup \overline{(\text{prods}_A - \text{prods}_F)} && \text{(properties 3a and 4)} \\
&= \text{prods}_C && \text{(definition of } \text{prods}_C\text{)} \quad \square
\end{aligned}$$

Property 6.

$$\begin{aligned}
\text{prods}_{A \circ C} &= \downarrow(\text{prods}_C \cup \text{prods}_{F \circ C}) && \text{(definition of } \text{prods}_A\text{)} \\
&= \downarrow(\text{prods}_C \cup \text{prods}_F) && \text{(property 3b)} \\
&= \downarrow(\underline{\text{prods}_A} \cup \overline{(\text{prods}_A - \text{prods}_F)} \cup \text{prods}_F) && \text{(defin. of } \text{prods}_C\text{)} \\
&= \downarrow(\underline{\text{prods}_A} \cup \text{prods}_F) && \text{(as } \text{prods}_A \text{ is convex)} \\
&= \text{prods}_A && \text{(since } \text{prods}_F \subseteq \text{prods}_A\text{)} \quad \square
\end{aligned}$$

Property 7.

Theorem 4.1 established a unique maximum element for downsets having the same span. Clearly the leaf view for each span, will be the unique lowest element. There is therefore an equivalence between leaf views and full views (full view: the downset of the maximum view) given by the span function. Functions  $C_l$  and  $C_n$  take the leaf and full views of all interfaces in a model. From this we see that  $C_l \circ A_l = C_l$ .  $\square$

The proofs for the Chapter five theorem and corollaries follow.

Theorem 5.1 Figure 5.2 is a commuting diagram, that is:

$$(i) A \circ C = A \quad \text{and} \quad (ii) C \circ A = C$$

Proof of (i). Recall that,

$$A = A_l \circ A_n \circ C_l \quad \text{and} \quad C = C_n \circ C_l$$

then,

$$\begin{aligned}
 A \circ C &= A_l \circ A_n \circ C_l \circ C_n \circ C_l \\
 &= A_l \circ A_n \circ C_n \circ C_l
 \end{aligned}$$

The leftmost  $C_l$  can be dropped because it has no effect as we now argue. The rightmost  $C_l$  is applied first: it produces a model which only contains leaf link producers and consumers. The following application of  $C_n$  only removes some producer and consumer nodes for those links with producers and consumers. So no non-leaf link producers or consumers can be introduced by  $C_n$ . Therefore the leftmost  $C_l$  is only applied to a leaf interface model, on which  $C_l$  has no effect.

$$\begin{aligned}
 &= A_l \circ A_n \circ C_l && \text{( by property 6 )} \\
 &= A
 \end{aligned}$$

Proof of (ii).

$$\begin{aligned}
 C \circ A &= C_n \circ C_l \circ A_l \circ A_n \circ C_l \\
 &= C_n \circ C_l \circ A_n \circ C_l && \text{( by property 7 )} \\
 &= C_n \circ A_n \circ C_l
 \end{aligned}$$

The leftmost  $C_l$  was dropped because it will have no effect. The rightmost  $C_l$  produces a model which only contains leaf link producers and consumers. The application of  $A_n$  only adds additional producer and consumer nodes for those links with producers and consumers. So no non-leaf link producers or consumers are introduced by  $A_n$ . The input to the leftmost  $C_l$  is then a leaf interface model, on which  $C_l$  has no effect.

$$\begin{aligned}
 &= C_n \circ C_l && \text{( by property 5 )} \\
 &= C
 \end{aligned}$$

□

### Corollary 5.1 The equivalence of compact and abstract models

$$A x = A y \Leftrightarrow C x = C y$$

Proof.

$$\begin{aligned}
 A x = A y &\Rightarrow C(A x) = C(A y) \\
 &\Rightarrow C x = C y && \text{( by theorem 5.1(ii) )}
 \end{aligned}$$

$$\begin{aligned}
 C x = C y &\Rightarrow A(C x) = A(C y) \\
 &\Rightarrow A x = A y && \text{( by theorem 5.1(i) )}
 \end{aligned}$$

□

### Corollary 5.2 Closure

$$(i) A \circ A = A \quad \text{and} \quad (ii) C \circ C = C$$



Proof of (i).

$$\begin{aligned}
 A \circ A &= A \circ C \circ A && \text{( by theorem 5.1(i) )} \\
 &= A \circ C && \text{( by theorem 5.1(ii) )} \\
 &= A && \text{( by theorem 5.1(i) )}
 \end{aligned}$$

Proof of (ii).

$$\begin{aligned}
 C \circ C &= C \circ A \circ C && \text{( by theorem 5.1(ii) )} \\
 &= C \circ A && \text{( by theorem 5.1(i) )} \\
 &= C && \text{( by theorem 5.1(ii) )} \quad \square
 \end{aligned}$$

**Lemma 5.1 Convexity.** An abstract model  $(\leq_l, \leq_n, \text{prods}, \text{cons})$  satisfies:

$$\begin{aligned}
 \forall l : \text{links} \bullet \\
 \quad \text{convex}(\text{prods } l) \text{ and } \text{convex}(\text{cons } l)
 \end{aligned}$$

**Proof.** Follows directly from the definitions of abstract producers and consumers for leaf links. Convexity also holds for non-leaf links as we now argue. Let  $a$  be a non-leaf link with producers  $X$  and  $P$ , where  $X < P$ . Then set of links  $\text{span } a$ , are outputs of  $X$  and  $P$  as the interfaces of an abstract model are down complete. Consider node  $Y$  which satisfies  $X < Y < P$ . Then  $Y$  also outputs the set of links  $\text{span } a$ , by the convexity of leaf links. Further,  $Y$ 's outputs include all links whose  $\text{span}$  is a subset of  $\text{span } a$ , as an abstract models interfaces are maximum downsets. Therefore,  $a$  is produced by  $Y$ .  $\square$

### B.3 Supporting Lemmas

**Lemma B.1** Let  $P$  be a finite ordered set, and  $S$  a downset of  $P$ , then the following holds:

$$\underline{S} = S \cap \underline{P}$$

**Proof** Straightforward.  $\square$

**Lemma B.2** Let  $P$  be a finite ordered set, with  $X$  and  $Y$  subsets of  $P$ , then the following holds:

$$Y \subseteq \uparrow X \Rightarrow (\underline{X \cup Y} = \underline{X})$$

**Proof,**

$$\begin{aligned}
 \underline{X} &= \underline{\uparrow X} \\
 &= \underline{\uparrow X \cup \uparrow Y} && \text{( since } \uparrow Y \subseteq \uparrow X \text{ )} \\
 &= \underline{\uparrow(X \cup Y)} \\
 &= \underline{X \cup Y} \quad \square
 \end{aligned}$$

---

## Gofer Implementation of Structured Graphs

---

STRUCTURED GRAPH's Gofer Implementation, Version 1.0 (Macintosh)

-----  
Copyright Mark J. Sifer 1996.

Gofer is an experimental functional language developed by Mark Jones, which is a variation of Haskell. If you are not familiar with Gofer, please read the documentation that comes with the Gofer package included on the floppy disc.

This collection of files in the folder sgraphs, provides a Gofer implementation of the structured graph visual formalism developed by Mark Sifer and John Potter. The following files should be included:

general, sets, poset, posetV, posetI, posetD, model, modelV, modelI and modelD

General and sets provide generic functions used by the other files. PosetV and ModelV contain the viewing (and editing for modelV) functions. PosetI and ModelI contain the invariant functions; which test what kind of poset or model we have. PosetD and modelD contain test data.

Both poset and model are Gofer project files. If you just wish to run poset functions then change to the sgraphs directory, then load the poset project file ("?" is my Gofer prompt):

```
? :cd sgraphs1.0
? :p poset
```

Example test executions of the poset functions are given in posetD. Similarly, if you wish to run model functions then load the model project file:

```
? :p model
```

then you can run the test executions given in modelD. The major functions demonstrated are compact and abstract model completions, and model composition. When creating new test data note that all lists (of node or link names) should be sorted, as the set and relation functions expect sorted lists (sets).

The viewing and composition functions included in this version support the most general structured graphs, and this accounts for their complexity. In a future version I hope to include functions for restricted structured graphs as well.

### NOTES 1.0

This version was implemented using Gofer 2.21 ported to the Mac by John Reekie. Alpha 4.03, a program text editor is used. This is shareware, if you continue to use this program, please pay the shareware fee.

## General

```

--
-- A collection of general functions to supplement the prelude.
-- Some functions will also facilitate faster execution.
--

-----

--
-- Ordering definitions and functions. Additional class instances to
-- allow comparison of (a,b,c) triples.

instance (Eq a, Eq b, Eq c) => Eq (a,b,c) where
  (x,y,z) == (u,v,w) = x==u && y==v && z==w

instance (Ord a, Ord b, Ord c) => Ord (a,b,c) where
  (x,y,z) <= (u,v,w) = x<u || (x==u && y<v) || (x==u && y==v && z<=w)

-- A function for ordering "names", and a faster maximum function.

matchL      :: Eq a => [a] -> [a] -> Bool
matchL _ [] = True
matchL [] _  = False
matchL (x:xs) (y:ys)
  | x == y    = matchL xs ys
  | otherwise = False

maximum     :: Ord a => [a] -> a
maximum (a:as) = fst (max_ (a,as))
              where max_ (m, []) = (m, [])
                    max_ (m, (x:xs)) = (max m x, xs)

-----

--
-- List functions. Position return the position of an item in a list.
-- Front returns a list minus its last item. End returns the last item.

position    :: (Eq a) => [a] -> a -> Int
position [] _ = 0
position (m:ms) x
  | m == x    = 1
  | otherwise = (position ms x) + 1

front      :: [a] -> [a]
front []   = []
front [x]  = []
front (x:xs) = x:front xs

end        :: [a] -> [a]
end []     = []
end (_:xs) = xs

exists     :: [a] -> Bool
exists x   = not (null x)

headEq    :: Eq a => a -> [a] -> Bool
headEq _ [] = False
headEq v (x:xs) = v == x

```

```

unique          :: [a] -> Bool
unique []       = True
unique [_]      = True
unique _       = False

-----

--
-- Functions to build and lookup a table. LookupI, LookupC and LookupS
-- are optimised for tables with a sorted index of their respective
-- integer, character and string types.

mhtable         :: [a] -> (a -> b) -> [(a, b)]
mhtable dom f   = zip dom (map f dom)

mhtableR        :: [a] -> (a -> b) -> [(b, a)]
mhtableR dom f  = zip (map f dom) dom

domainT         :: Eq a => [(a,b)] -> [a]
domainT t       = map fst t

lookup          :: Eq a => [(a, b)] -> a -> b
lookup (t@(a,b):tx) i
  | a == i      = b
  | otherwise   = lookup tx i

lookupZ         :: [(Int, a)] -> Int -> a
lookupZ array i = snd (array !! (i))

lookupI        :: [(Int, a)] -> Int -> a
lookupI array i = snd (array !! (i-1))

lookupN        :: [(Int, a)] -> Int -> a
lookupN tab@(fi,q):tx i = snd (tab !! (i - fi))

lookupC        :: [(Char, a)] -> Char -> a
lookupC tab@(fc,q):tx i = snd (tab !! (ord i - ord fc))

lookupS        :: [(String, a)] -> String -> a
lookupS (t@(s,a):tx) i
  | i `matchL` s = a
  | otherwise   = lookup tx i

lookup2        :: Eq a => [(a,b,c)] -> a -> b
lookup2 (t@(a,b,c):tx) i
  | a == i      = b
  | otherwise   = lookup2 tx i

lookup3        :: Eq a => [(a,b,c)] -> a -> c
lookup3 (t@(a,b,c):tx) i
  | a == i      = c
  | otherwise   = lookup3 tx i

-----

--
-- Functions for manipulating tuples and functions

swap           :: [(a,b)] -> [(b,a)]
swap []        = []
swap ((a,b):r) = (b,a) : swap r

```

```

stripFst      :: [(a,b)] -> [a]
stripFst ls   = [ a | (a,b) <- ls ]

stripSnd      :: [(a,b)] -> [b]
stripSnd ls   = [ b | (a,b) <- ls ]

inv           :: (Eq a, Eq b) => [a] -> (a ->[b]) -> (b -> [a])
inv d f       = nub.inv_d f
              where inv_ [] f x      = []
                    inv_ (a:as) f x =
                      | x `elem` f a = a : inv_ as f x
                      | otherwise    = inv_ as f x

```

---

## Sets

-- Functions for standard set operations. Sets are implemented as  
-- sorted lists without duplicates. Bags are implemented as sorted lists  
-- with possible duplicates. Rdup converts a bag into a set.

```

type Member = Int
type Set    = [Member]

mkset      :: Ord a => [a] -> [a]
mkset      = rdup.mkbag

set        :: Ord a => [a] -> Bool
set []     = True
set [a]    = True
set (a:r@(b:s))
  | a < b  = set r
  | otherwise = False

mkbag     :: Ord a => [a] -> [a]
mkbag     = sort

bag       :: Ord a => [a] -> Bool
bag []    = True
bag [a]   = True
bag (a:r@(b:s))
  | a <= b = bag r
  | otherwise = False

rdup     :: Ord a => [a] -> [a]
rdup []  = []
rdup [a] = [a]
rdup (a : r@(b : _))
  | a < b      = a : rdup r
  | otherwise  = rdup r

union    :: Ord a => [a] -> [a] -> [a]
union []  ys = ys
union xs [] = xs
union (x:xs) (y:ys)
  | x == y  = x : union xs ys
  | x < y   = x : union xs (y:ys)
  | x > y   = y : union (x:xs) ys

isect   :: Ord a => [a] -> [a] -> [a]
isect []  ys = []
isect xs [] = []
isect (x:xs) (y:ys)

```

```

| x == y      = x : isect xs ys
| x < y       = isect xs (y:ys)
| x > y       = isect (x:xs) ys

diff          :: Ord a => [a] -> [a] -> [a]
diff []       ys      = []
diff xs      []       = xs
diff (x:xs) (y:ys)
| x == y      = diff xs ys
| x < y       = x : diff xs (y:ys)
| x > y       = diff (x:xs) ys

disjoint      :: (Ord a, Eq [a]) => [a] -> [a] -> Bool
disjoint x y  = isect x y == []

unions        :: Ord a => [[a]] -> [a]
unions xs
| null xs     = []
| otherwise   = foldr1 union xs

isects        :: Ord a => [[a]] -> [a]
isects xs
| null xs     = []
| otherwise   = foldr1 isect xs

subset        :: (Ord a, Eq [a]) => [a] -> [a] -> Bool
subset x y    = x `isect` y == x

subsetS       :: (Ord a, Eq [a]) => [a] -> [a] -> Bool
subsetS x y   = (x `isect` y == x) && (x /= y)

--
--
-- Relation Table Functions
--
-- mkRT converts a list of pairs into a relation table. padRT pads the
-- relation table with empty list values so it is defined for a domain.
-- TotalR converts a partial function to a total function. ZipR combines
-- two relations together using the supplied function to combine values.

mkRT          :: Ord (a,b) => [(a,b)] -> [(a,[b])]
mkRT []       = []
mkRT pT      = mkRT_ (fst (head pT')) [] pT'
               where pT' = mkset pT
                     mkRT_ i r [] = [(i,r)]
                     mkRT_ i r ((a,b):ps)
                       | i == a    = mkRT_ i (r++[b]) ps
                       | otherwise = (i,r) : mkRT_ a [b] ps

mkRT2         :: Ord (a,[b]) => [(a,[b])] -> [(a,[b])]
mkRT2 []      = []
mkRT2 pT      = mkRT2_ (fst (head pT')) [] pT'
               where pT' = mkset pT
                     mkRT2_ i r [] = [(i,r)]
                     mkRT2_ i r ((a,bs):ps)
                       | i == a    = mkRT2_ i (r++bs) ps
                       | otherwise = (i,r) : mkRT2_ a bs ps

partition     :: Ord (a,b) => [(a,b)] -> [(a,[b])]
partition     = mkRT

padRT         :: Ord (a,b) => [a] -> [(a,[b])] -> [(a,[b])]
padRT []      [] = []
padRT (d:ds) [] = (d,[]) : padRT ds []
padRT (d:ds) tT@(t@(i,v):ts)
| d == i      = t : padRT ds ts

```

```

    | otherwise = (d,[]) : padRT ds tT

domainRT      :: (Ord a, Ord b) => [(a, [b])] -> [a]
domainRT rT   = map fst rT

rangeRT       :: (Ord a, Ord b) => [(a, [b])] -> [b]
rangeRT rT    = unions [ bs | (a, bs) <- rT ]

lookupRT      :: Ord (a,b) => [(a,[b])] -> a -> [b]
lookupRT rT x
  | x `elem` (map fst rT) = lookup rT x
  | otherwise             = []

totalR        :: Ord (a,b) => [a] -> (a -> [b]) -> a -> [b]
totalR dom f x
  | x `elem` dom = f x
  | otherwise    = []

zipR          :: Ord (a,b) => [a] -> (a -> [b]) -> [a] -> (a -> [b]) -> ([b] -> [b]
-> [b]) -> a -> [b]
zipR dA rA dB rB f = lookupRT newRT
                    where newRT = [(a,val) | a <- dA `union` dB,
                                             val = totalR dA rA a `f` totalR dB rB a]

--          List Comprehension Optimisation
--
-- Two functions to [ | x <- set, y <- set, ... ] where (x,y) and
-- (y,x) are equivalent.

after         :: Ord a => [a] -> a -> [a]
after set x   = [ a | a <- set, a > x ]

afterEq      :: Ord a => [a] -> a -> [a]
afterEq set x = [ a | a <- set, a >= x ]

--          Inverse Function
--
-- Unlike the "inv" function, "inV" assumes the first two arguments: the domain
-- and range of the function, are sets. Also this function requires a finite range.
-- "inVI" is used when the domain and range are optimised integer sets.

inV          :: (Ord a, Ord b) => [a] -> [b] -> (a ->[b]) -> (b -> [a])
inV d r f    = let
                invT = zip r (map inv' r)
                invR x = lookup invT x
                inv' x = [ a | a <- d, x `elem` f a ]
            in
                invR

inVI         :: Set -> Set -> (Member -> Set) -> (Member -> Set)
inVI d r f   = let
                invT = zip r (map inv' r)
                invR x = lookupI invT x
                inv' x = [ a | a <- d, x `elem` f a ]
            in
                invR

```

**PosetV**

```

--
--                               Labeled Relations
--
-- ConvertR, separates a labeled relation into an un-named relation
-- implemented with integers, and functions which convert between
-- named items and un-named items.

type Domain      = Set
type DomainN    = [Name]

type Name       = String
type NameF     = Member -> Name
type IndxF     = Name -> Member

type Relation   = Member -> Set
type RelationN = Name -> [Name]
type RelationT = (NameF, IndxF, Domain, Relation)

convertR      :: DomainN -> RelationN -> RelationT
convertR d r  = let
    relatnT   = zip domain (map relatn' domain)
    relatn' i = map indx (r (name i))
    name i    = d !! (i-1)
    indx n    = position d n
    domain    = [1..length d]
    relatn i  = lookupI relatnT i
  in
    (name, indx, domain, relatn)

printR       :: DomainN -> RelationN -> [(Name,Member)], [(Member,Set)]
printR d r   = let
    nameT     = mktable d indx
    relatnT   = mktable domain relatn'
    relatn' i = map indx (r (name i))
    name i    = d !! (i-1)
    indx n    = position d n
    domain    = [1..length d]
  in
    (nameT, relatnT)

--
--                               POSET Table
--
-- A table to capture the major poset functions for a given poset.
-- For each member of the poset; children, parents, down set, up set,
-- leaf set, and roots set are captured (for efficiency). For the whole
-- poset; the domain, maximals, minimals, and labeled item conversion
-- functions are captured.
--
-- "lookupP" pre: m is in domain of p, and hence table is not empty.

type Children  = Member -> Set
type Parents   = Member -> Set

type Ford     = (Member,Set,Set,Set,Set,Set,Set)

member_ (m, _, _, _, _, _, _) = m
children_ (_, c, _, _, _, _, _) = c

```



```

parents_ (_ , _ , p, _ , _ , _ , _) = p
down_    (_ , _ , _ , d, _ , _ , _) = d
up_      (_ , _ , _ , _ , u, _ , _) = u
leaf_    (_ , _ , _ , _ , _ , l, _) = l
root_    (_ , _ , _ , _ , _ , _ , r) = r

data Poset = Poset NameF IndxF Set Set Set [Pord]

name      (Poset x _ _ _ _ _) = x
indx      (Poset _ x _ _ _ _) = x
domain    (Poset _ _ x _ _ _) = x
maximals  (Poset _ _ _ x _ _) = x
minimals  (Poset _ _ _ _ x _) = x
pordT     (Poset _ _ _ _ _ x) = x

mkposet   :: RelationT -> Poset
mkposet (n,i,dom,ch) = Poset n i dom mx mn t
              where mx = mkmaximals dom ch
                    mn = mkminimals dom ch
                    t  = mkpordT dom ch

mkmaximals :: Domain -> Children -> FView
mkmaximals d ch = [ m | m <- d, p m == [] ]
              where p m = sort (nub [ c | c <- d, m `elem` ch c])

mkminimals  :: Domain -> Children -> FView
mkminimals d ch = [ m | m <- d, ch m == [] ]

mkpordT     :: Domain -> Children -> [Pord]
mkpordT dom ch = zip7 (dom)
                    (map ch dom)
                    (map parents' dom)
                    (map down' dom)
                    (map up' dom)
                    (map leaf' dom)
                    (map root' dom)
              where
                parents' m = sort (nub [ c | c <- dom, m `elem` ch c])
                down' m   = unions ([m] : map down' (ch m))
                up' m     = unions ([m] : map up' (parents' m))
                leaf' m   = (down' m) `isect` mn
                root' m   = (up' m) `isect` mx
                mx        = mkmaximals dom ch
                mn        = mkminimals dom ch

lookupP     :: Poset -> Member -> Pord
lookupP p m = (pordT p) !! (m - 1)

prposet     :: Poset -> (String, [(Member, Name)], String, [(Member, [Member])])
prposet p   = let
                dom      = domain p
                domT     = mhtable dom (name p)
                chT      = [ (i, chI) | i <- dom, chI = children p i ]
              in
                (" ## Domain ## ", domT,
                 " ## Children ## ", chT )

```

```

--                               POSET Standard Functions
--
-- Standard functions on a poset. Most are just a lookup of the POSET
-- table, or a small variation on one of these. DownS is read 'strict
-- down set', and DownE is read 'exclusive down set'.

type View    = Set      -- Arbitrary View
type LView   = View     -- Leaf View
type RView   = View     -- Root View
type FView   = View     -- Flat View

children     :: Poset -> Member -> FView
children p m = children_ (lookupP p m)

parents      :: Poset -> Member -> FView
parents p m  = parents_ (lookupP p m)

down         :: Poset -> Member -> View
down p m    = down_ (lookupP p m)

up           :: Poset -> Member -> View
up p m     = up_ (lookupP p m)

leaf         :: Poset -> Member -> LView
leaf p m   = leaf_ (lookupP p m)

root         :: Poset -> Member -> RView
root p m   = root_ (lookupP p m)
downS       :: Poset -> Member -> View
downS p m  = (down p m) `diff` [m]

downE       :: Poset -> Member -> View
downE p m  = [ x | x <- down p m, up p x `subset` mUpDown ]
             where mUpDown = up p m `union` down p m

downs       :: Poset -> View -> View
downs p ms = unions (map (down p) ms)

downsS     :: Poset -> View -> View
downsS p ms = unions (map (downS p) ms)

upS        :: Poset -> Member -> View
upS p m    = (up p m) `diff` [m]

ups        :: Poset -> View -> View
ups p ms   = unions (map (up p) ms)

uppers     :: Poset -> View -> View
uppers p ms = [ m | m <- domain p, ms `subset` down p m ]

lowers     :: Poset -> View -> View
lowers p ms = [ m | m <- domain p, ms `subset` up p m ]

uplows     :: Poset -> View -> View
uplows p ms = uppers p ms `union` lowers p ms

convex     :: Poset -> View -> View
convex p ms = downs p ms `isect` ups p ms

```

```

--
--                               Poset VIEWING functions
--
-- Functions for comparing and transforming 'views'. Less is the ordering
-- relation between items in the poset. Vless is the ordering relation
-- between views (subsets) in the poset. Flat checks a view is flat.
-- FlattenU returns the local maximals of a view which is a flat view.
-- DomainsD returns the lowest covering flat view of a view.
--
-- Leaf, Roots, MaxV, Level, and Context are the standard viewing
-- functions. Note that if the poset is not a tree, Context returns multiple
-- flat context views, one for each 'path'. Mdepth returns the maximum
-- depth of a poset.
--
-- dlevel          precondition: i >= 0
-- dcontext         : m element downset d
-- dpaths, dcontexts : b <= t

type Path    = [Member]

less        :: Poset -> Member -> Member -> Bool
less p x y  = x `elem` down p y

lessS       :: Poset -> Member -> Member -> Bool
lessS p x y = (x `elem` down p y) && (x /= y)

vless       :: Poset -> View -> View -> Bool
vless p x y = (downs p x) `subset` (downs p y)
vlessS      :: Poset -> View -> View -> Bool
vlessS p x y = ((downs p x) `subset` (downs p y)) && (x /= y)

leafV       :: Poset -> View -> Bool
leafV p v   = and [ children p x == [] | x <- v ]

flat        :: Poset -> View -> Bool
flat p []   = True
flat p v    = not (foldr (||) False [x `elem` (down p y) | x <- v, y <- v, x /= y])

flattenU    :: Poset -> View -> FView
flattenU p [] = []
flattenU p v  = v `diff` rdup [x | x <- v, y <- v, lessS p x y]

flattenI    :: Poset -> View -> FView
flattenI p [] = []
flattenI p v  = v `diff` rdup [x | x <- v, y <- v, (leaf p x) `subsetS` (leaf p y)]

flattenD    :: Poset -> View -> FView
flattenD p [] = []
flattenD p v  = v `diff` rdup [x | x <- v, y <- v, lessS p y x]

domainsD    :: Poset -> View -> FView
domainsD p ms = flattenD p (isects (map (up p) ms))

leaves     :: Poset -> View -> LView
leaves p ms = unions (map (leaf p) ms)

roots      :: Poset -> View -> RView
roots p ms  = unions (map (root p) ms)

maxV1     :: Poset -> LView -> FView
maxV1 p ms = flattenU p (unions (map maxV1' (roots p ms)))
  where top    = ups p ms
        maxV1' m
          | m `notElem` top    = []
          | (leaf p m) `subset` ms = [m]

```

```

| otherwise
    = unions (map maxV1' (children p m))

maxV2      :: Poset -> LView -> FView
maxV2 p ms = flattenU p [ m | m <- top, leaf p m `subset` ms]
            where top = ups p ms

maxD       :: Poset -> LView -> FView
maxD p ms  = [ m | m <- top, leaf p m `subset` ms]
            where top = ups p ms

maximumI   :: Poset -> LView -> FView
maximumI p ms = flattenI p [ m | m <- top, leaf p m `subset` ms]
            where top = ups p ms

maximumV   :: Poset -> LView -> FView
maximumV p ms = maxV1 p ms

minV1      :: Poset -> RView -> FView
minV1 p ms = flattenD p (unions (map minV1' (leaves p ms)))
            where bot = downs p ms
                  minV1' m
                    | m `notElem` bot = []
                    | (root p m) `subset` ms = [m]
                    | otherwise = unions (map minV1' (parents p m))

minV2      :: Poset -> RView -> FView
minV2 p ms = flattenD p [ m | m <- bot, root p m `subset` ms]
            where bot = downs p ms

minimumV   :: Poset -> RView -> FView
minimumV p ms = minV1 p ms

dlevel     :: Poset -> Int -> Member -> FView
dlevel p i d
  | children p d == [] = [d]
  | i == 0             = [d]
  | otherwise          = unions [dlevel p (i-1) c | c <- children p d]

dlevels    :: Poset -> Int -> View -> View
dlevels p i ms = unions (map (dlevel p i) ms)

mdepth     :: Poset -> Int
mdepth p = mdepth' (minimals p) (maximals p)
            where mdepth' ns ms
                  | ns `subset` ms = 0
                  | otherwise = (mdepth' ns (dlevels p 1 ms)) + 1

dpaths     :: Poset -> Member -> Member -> [Path]
dpaths p t b
  | t == b = [[t]]
  | otherwise = map (b:) (concat (map (dpaths p t) (parents p b)))

dcontext   :: Poset -> Path -> FView
dcontext p (b:np) = [b] `union` ((unions (map (children p) np)) `diff` np)

dcontexts  :: Poset -> Member -> Member -> [FView]
dcontexts p t b = map (dcontext p) (dpaths p t b)

chains     :: Poset -> Set -> [Path]
chains p xs = concat [ trim (dpaths p t b) | t <- tops,
                    b <- down p t `isect` bots ]

```

```

where tops = flattenU p xs
      bots = flattenD p xs
      trim = nub.(map (filter (\n -> n `elem` xs)))

--
--          Graphic Interface POSET Viewing functions
--
-- WidthD returns the width if boxes used to represent each item of the
-- poset in the graphical poset viewing display. WidthD returns the width
-- of an item's box w.r.t. the items descendents. WidthU returns the
-- width of an item's box w.r.t. the items ancestors. Leaf and root box's
-- are length 10 respectively.
--
-- Spacing is the space between sibling boxes. WidthsD and WidthsU return
-- box widths for a whole interface when the selected member is the
-- viewing 'domain'. The 'domain' member box will have the 'max' width.
-- In practice 'max' is the number of pixels the widest box can be.
spacing = 0.2

convIF = primIntToFloat

widthD    :: Poset -> Member -> Float
widthD p m
  | children p m == []    = 1.0
  | otherwise             = (1.0 + (chC - 1.0) * spacing) * chW
                        where ch  = children p m
                              chW = foldr1 (+) (map (widthD p) ch)
                              chC = convIF (length ch)

widthU    :: Poset -> Member -> Float
widthU p m
  | parents p m == []    = 1.0
  | otherwise            = (1.0 + (paC - 1.0) * spacing) * paW
                        where pa  = parents p m
                              paW = foldr1 (+) (map (widthU p) pa)
                              paC = convIF (length pa)

widthsD   :: Poset -> Member -> [(Member, Float)]
widthsD p m = let
      max = 100.0
      top = (widthD p m)
      dom = down p m
      f x = ((widthD p x) * max) / top
  in
      mktable dom f

widthsU   :: Poset -> Member -> [(Member, Float)]
widthsU p m = let
      max = 100.0
      bot = (widthU p m)
      dom = up p m
      f x = ((widthU p x) * max) / bot
  in
      mktable dom f

--
--          POSET Name Filter Functions
--
-- Name filter functions allow names to be used rather than poset member
-- indexes which are integers, in both function inputs and outputs. Two
-- varieties of filter functions are provided. An example shows the
-- difference.
--

```

```

-- Specific filter function use: ? nN down posetC "a"
--
-- General filter function use: ? nX down posetC "a" :: [Name]
--
-- Note the general name filter function use requires the "nX" call to be
-- explicitly qualified with the return type. If this is not done
-- unresolved function overloading will occur.

type ChildrenN = RelationN

mkposetN      :: DomainN -> ChildrenN -> Poset
mkposetN dN cN = mkposet (convertR dN cN)
-- Specific name filter functions for each return type -----

class NameFilterBool a b where
  nB :: (Poset -> a) -> Poset -> b -> Bool

instance NameFilterBool (View -> Bool) [Name] where
  nB f p ns = f p (map (indx p) ns)

class NameFilterBool2 a b c where
  nB2 :: (Poset -> a) -> Poset -> b -> c -> Bool

instance NameFilterBool2 (Member -> Member -> Bool) Name Name where
  nB2 f p a b = f p ((indx p) a) ((indx p) b)

instance NameFilterBool2 (View -> View -> Bool) [Name] [Name] where
  nB2 f p x y = f p (map (indx p) x) (map (indx p) y)

class NameFilterFloat a b where
  nF :: (Poset -> a) -> Poset -> b -> Float

instance NameFilterFloat (Member -> Float) Name where
  nF f p m = f p ((indx p) m)

class NameFilterFloatTable a b where
  nFT :: (Poset -> a) -> Poset -> b -> [(Name, Float)]

instance NameFilterFloatTable (Member -> [(Member, Float)]) Name where
  nFT f p m = let
      nameT [] = []
      nameT ((m,s):ms) = ((name p) m, s):nameT ms
    in
      nameT (f p ((indx p) m))

class NameFilterView a b where
  nV :: (Poset -> a) -> Poset -> b -> View

instance NameFilterView (Member -> View) Name where
  nV f p m = f p ((indx p) m)

class NameFilterNames a b where
  nN :: (Poset -> a) -> Poset -> b -> [Name]

instance NameFilterNames (Member -> View) Name where
  nN f p m = map (name p) (f p ((indx p) m))

```

```

instance NameFilterNames (View -> View) [Name] where
  nN f p ns = map (name p) (f p (map (indx p) ns))

class NameFilterNames1 a where
  nN1 :: (Poset -> a) -> Poset -> [Name]

instance NameFilterNames1 View where
  nN1 f p = map (name p) (f p)
class NameFilterNames2 a b c where
  nN2 :: (Poset -> a) -> Poset -> b -> c -> [Name]

instance NameFilterNames2 (Int -> Member -> View) Int Name where
  nN2 f p i m = map (name p) (f p i ((indx p) m))

instance NameFilterNames2 (Int -> View -> View) Int [Name] where
  nN2 f p i ns = map (name p) (f p i (map (indx p) ns))

instance NameFilterNames2 (Member -> Member -> View) Name Name where
  nN2 f p a b = map (name p) (f p ((indx p) a) ((indx p) b))

class NameFilterPaths2 a b c where
  nP2 :: (Poset -> a) -> Poset -> b -> c -> [[Name]]

instance NameFilterPaths2 (Member -> Member -> [Path]) Name Name where
  nP2 f p a b = map (map (name p)) (f p ((indx p) a) ((indx p) b))

-- General filter function "nX" for all return types -----

class NameFilter a b where
  nX :: (Poset -> a) -> Poset -> b

instance NameFilter (View -> Bool) ([Name] -> Bool) where
  nX f p ns = f p (map (indx p) ns)

instance NameFilter (Member -> Member -> Bool) (Name -> Name -> Bool) where
  nX f p a b = f p ((indx p) a) ((indx p) b)

instance NameFilter (View -> View -> Bool) ([Name] -> [Name] -> Bool) where
  nX f p x y = f p (map (indx p) x) (map (indx p) y)

instance NameFilter (Member -> Float) (Name -> Float) where
  nX f p m = f p ((indx p) m)

instance NameFilter (Member -> [(Member, Float)]) (Name -> [(Name, Float)]) where
  nX f p m = let
      nameT [] = []
      nameT ((m,s):ms) = ((name p) m, s):nameT ms
    in
      nameT (f p ((indx p) m))

instance NameFilter (Member -> View) (Name -> View) where
  nX f p m = f p ((indx p) m)

instance NameFilter (Member -> View) (Name -> [Name]) where
  nX f p m = map (name p) (f p ((indx p) m))

```





```

poset      :: Poset -> Bool
poset po   = and [noCycle po, minTran po]

-- Constraints for the covering relation of an Inclusion Order -----

uniqueSp   :: Poset -> Bool
uniqueSp po = and [ leaf po x /= leaf po y | x <- domain po, y <- domain po, x /= y ]

subsetOrd  :: Poset -> Bool
subsetOrd po = and [ lessS po x y | x <- domain po,
                               y <- domain po, leaf po x `subsetS` leaf po y ]

incOrd     :: Poset -> Bool
incOrd po   = and [noCycle po, minTran po, uniqueSp po, subsetOrd po]

-- Constraints for the covering relation of a Forest* -----

onePa      :: Poset -> Bool
onePa po   = and [ length ((parents po) x) <= 1 | x <- domain po ]

oneCh      :: Poset -> Bool
oneCh po   = and [ length ((children po) x) <= 1 | x <- domain po ]

forest     :: Poset -> Bool
forest po  = and [noCycle po, minTran po, onePa po, not (oneCh po)]

-- Constraints for the covering relation of a Tree* -----

oneMax     :: Poset -> Bool
oneMax po  = length (maximals po) == 1

tree       :: Poset -> Bool
tree po    = and [noCycle po, minTran po, onePa po, not (oneCh po), oneMax po]

-- Order Classifier -----

orderF     :: Poset -> [Bool]
orderF po  = [ not (noCycle po) = [False]
              | otherwise       = [True, minTran po, uniqueSp po, subsetOrd po, onePa po,
                                  oneMax po] ]

order      :: Poset -> String
order po   = [ not (noCycle po)      = "Directed Graph"
              | not (minTran po)     = "Directed Acyclic Graph"
              | not (uniqueSp po)    = "Poset"
              | not (subsetOrd po)   = "Semi-Inclusion Order"
              | not (onePa po)       = "Inclusion Order"
              | not (oneMax po)      = "Forest*"
              | otherwise            = "Tree*" ]

```

**PosetD**

```

-----
--
--
--                                POSET DATA
--
-----

-- 1 : A directed graph with a cycle -----

domain1      :: DomainN
domain1      = ["a", "b", "c", "d"]

children1    :: ChildrenN
children1 "a" = ["b"]
children1 "b" = ["c", "d"]
children1 "c" = ["a"]
children1 _   = []

relatn1      :: Poset
relatn1      = mkposetN domain1 children1

-- 2 : A DAG with a redundant transitive link -----

domain2      :: DomainN
domain2      = ["a", "b", "c", "d", "e", "f"]

children2    :: ChildrenN
children2 "a" = ["b", "f"]
children2 "b" = ["c", "d"]
children2 "d" = ["e", "f"]
children2 _   = []

relatn2      :: Poset
relatn2      = mkposetN domain2 children2

-- 3 : A poset with non unique span -----

domain3      :: DomainN
domain3      = ["a", "b", "c", "d"]

children3    :: ChildrenN
children3 "a" = ["c", "d"]
children3 "b" = ["c", "d"]
children3 _   = []

relatn3      :: Poset
relatn3      = mkposetN domain3 children3

-- 4 : A semi-inclusion order with non subset ordering -----

domain4      :: DomainN
domain4      = ["a", "b", "c", "d", "e", "f", "g"]

children4    :: ChildrenN
children4 "a" = ["b", "d"]
children4 "b" = ["e", "f"]
children4 "c" = ["e", "g"]
children4 "d" = ["f", "g"]
children4 _   = []

```

```

relatn4      :: Poset
relatn4      = mkposetN domain4 children4

-- 5 : An inclusion order with multiple parents -----

domain5      :: DomainN
domain5      = ["a","b","c","d","e","f"]

children5    :: ChildrenN
children5 "a" = ["b","c"]
children5 "b" = ["d","e"]
children5 "c" = ["e","f"]
children5 _   = []

relatn5      :: Poset
relatn5      = mkposetN domain5 children5

-- 6 : An forest with multiple maximals -----

domain6      :: DomainN
domain6      = ["a","b","c","d","e","f","g","h","i"]

children6    :: ChildrenN
children6 "a" = ["b","c"]
children6 "b" = ["d","e"]
children6 "f" = ["g","h"]
children6 _   = []

relatn6      :: Poset
relatn6      = mkposetN domain6 children6

-- 7 : A Tree -----

domain7      :: DomainN
domain7      = ["a","b","c","d","e","f"]

children7    :: ChildrenN
children7 "a" = ["b","c"]
children7 "b" = ["d","e","f"]
children7 _   = []

relatn7      :: Poset
relatn7      = mkposetN domain7 children7

-----

-- 8 : Another poset, an almost tree with with a chain -----

domain8      :: DomainN
domain8      = ["a","b","c","d","e","f"]

children8    :: ChildrenN
children8 "a" = ["b","c"]
children8 "b" = ["d"]
children8 "d" = ["e","f"]
children8 _   = []

relatn8      :: Poset
relatn8      = mkposetN domain8 children8

```

```

-- 9 : Another poset, two chains -----
domain9      :: DomainN
domain9      = ["a","b","c","d","e","f"]

children9    :: ChildrenN
children9 "a" = ["b"]
children9 "b" = ["c"]
children9 "d" = ["e"]
children9 "e" = ["f"]
children9 _   = []

relatn9      :: Poset
relatn9      = mkposetN domain9 children9

-- 10 : Another poset, a three way multi-chain -----
domain10     :: DomainN
domain10     = ["a","b","c","d","e","f","g","h","i"]

children10   :: ChildrenN
children10 "a" = ["d","e"]
children10 "b" = ["d","f"]
children10 "c" = ["e","f"]
children10 "d" = ["g","h"]
children10 "e" = ["g","i"]
children10 "f" = ["h","i"]
children10 _   = []

relatn10     :: Poset
relatn10     = mkposetN domain10 children10

-- 11 : Another semi-inclusion order, overlapping trees -----
domain11     :: DomainN
domain11     = ["a","b","c","d","e","f","g","h"]

children11   :: ChildrenN
children11 "a" = ["c","d"]
children11 "c" = ["e","f"]
children11 "d" = ["g","h"]
children11 "b" = ["f","g"]
children11 _   = []

relatn11     :: Poset
relatn11     = mkposetN domain11 children11

```

---

## ModelV

```

--
--           MODEL Classifier Functions
--
-- Boolean functions which indicate if an input is a: model, compact model,
-- abstract model and a complete model. All these functions require an input
-- output representation of models.

model      :: IOmodel -> Bool
model (n0,l0,o,i) = and [poset n0, poset l0]

```

```

compactModel    :: IOmodel -> Bool
compactModel m  = (compactIO m) `equalIO` m

abstractModel   :: IOmodel -> Bool
abstractModel m = (abstractIO m) `equalIO` m

equalIO         :: IOmodel -> IOmodel -> Bool
equalIO x y     = let
    (n0,l0,o1,i1) = x
    ( _,  _,o2,i2) = y
    equal_outs    = and [ o1 n == o2 n | n <- domain n0 ]
    equal_ins     = and [ i1 n == i2 n | n <- domain n0 ]
  in
    and [equal_outs, equal_ins]

completeModel   :: IOmodel -> Bool
completeModel m = and [ complete l | l <- domain l0 ]
  where (n0,l0,p,c) = makePC (lC m)
        complete l = and [leafV n0 lowP, leafV n0 lowC, null (lowP
`isect` lowC)]
                                where lowP = flattenD n0 (p l)
                                        lowC = flattenD n0 (c l)

```

## ModelD

```

-----
--
--                                MODEL DATA
--
-----

-- Thesis figure 3.8 (i) -----

ndomain1      :: DomainN
ndomain1      = ["P","Q","U","V","X","Y"]

nchildren1    :: ChildrenN
nchildren1 "P" = ["U","V"]
nchildren1 "Q" = ["X","Y"]
nchildren1 _   = []

ldomain1      :: DomainN
ldomain1      = ["a"]

lchildren1    :: ChildrenN
lchildren1 "a" = []
lchildren1 _   = []

sOut1         :: NodeN -> [LinkN]
sOut1 "V"     = ["a"]
sOut1 _       = []

sIn1          :: NodeN -> [LinkN]
sIn1 "X"      = ["a"]
sIn1 _        = []

modell = mkIOmodel (ndomain1,nchildren1,ldomain1,lchildren1,sOut1,sIn1)

-- ? printIOmodel modell
-- ("Out-In",[(("P",[],[ ]), ("Q",[],[ ]), ("U",[],[ ]), ("V",["a"],[ ]), ("X",[],[ "a"])),

```

```

-- ("Y", [], []) :: (String, [(NodeN, [LinkN], [LinkN])])

-- ? compactModel model1
-- True :: Bool

-- ? printIOmodel (abstractIO model1)
-- ("Out-In", [{"P", ["a"], []}, {"Q", [], ["a"]}, {"U", [], []}, {"V", ["a"], []},
{"X", [], ["a"]},
-- ("Y", [], [])]) :: (String, [(NodeN, [LinkN], [LinkN])])

-- Thesis figure 3.8 (ii) -----

ndomain2      :: DomainN
ndomain2      = ["P", "Q", "V", "X", "Y"]

nchildren2    :: ChildrenN
nchildren2 "P" = ["V", "X"]
nchildren2 "Q" = ["X", "Y"]
nchildren2 _  = []

ldomain2      :: DomainN
ldomain2      = ["a"]

lchildren2    :: ChildrenN
lchildren2 "a" = []
lchildren2 _  = []

sOut2        :: NodeN -> [LinkN]
sOut2 "V"     = ["a"]
sOut2 _      = []

sIn2         :: NodeN -> [LinkN]
sIn2 "Y"      = ["a"]
sIn2 _       = []

model2 = mkIOmodel (ndomain2, nchildren2, ldomain2, lchildren2, sOut2, sIn2)

-- ? printIOmodel model2
-- ("Out-In", [{"P", [], []}, {"Q", [], []}, {"V", ["a"], []}, {"X", [], []}, {"Y", [], ["a"]}))
-- :: (String, [(NodeN, [LinkN], [LinkN])])

-- ? compactModel model2
-- True :: Bool

-- ? printIOmodel (abstractIO model2)
-- ("Out-In", [{"P", ["a"], []}, {"Q", [], ["a"]}, {"V", ["a"], []}, {"X", [], []},
{"Y", [], ["a"]}))
-- :: (String, [(NodeN, [LinkN], [LinkN])])

-- Thesis figure 3.8 (iii) -----

ndomain3      :: DomainN
ndomain3      = ["P", "Q", "V", "X", "Y"]

nchildren3    :: ChildrenN
nchildren3 "P" = ["V", "X"]
nchildren3 "Q" = ["X", "Y"]
nchildren3 _  = []

ldomain3      :: DomainN
ldomain3      = ["a"]

```

```

lchildren3      :: ChildrenN
lchildren3 "a"  = []
lchildren3 _    = []

sOut3           :: NodeN -> [LinkN]
sOut3 "X"       = ["a"]
sOut3 _         = []

sIn3            :: NodeN -> [LinkN]
sIn3 "Y"        = ["a"]
sIn3 _          = []

model3 = mkIOmodel (ndomain3,nchildren3,lchildren3,sOut3,sIn3)

-- ? printIOmodel model3
-- ("Out-In",[(("P",[[]],[[]]), ("Q",[[]],[[]]), ("V",[[]],[[]]), ("X",["a"],[[]]), ("Y",[[]],[[]]))])
-- :: (String,[(NodeN,[LinkN],[LinkN])])

-- ? compactModel model3
-- True :: Bool

-- ? printIOmodel (abstractIO model3)
-- ("Out-In",[(("P",["a"],[[]]), ("Q",[[]],[[]]), ("V",[[]],[[]]), ("X",["a"],[[]]), ("Y",[[]],[[]]))])
-- :: (String,[(NodeN,[LinkN],[LinkN])])

-- Thesis figure 3.8 (iv) -----

ndomain4        :: DomainN
ndomain4        = ["P","Q","U","V","X","Y"]

nchildren4      :: ChildrenN
nchildren4 "P"  = ["U","V","X"]
nchildren4 "Q"  = ["V","X","Y"]
nchildren4 _    = []

ldomain4        :: DomainN
ldomain4        = ["a"]

lchildren4      :: ChildrenN
lchildren4 "a"  = []
lchildren4 _    = []

sOut4           :: NodeN -> [LinkN]
sOut4 "V"       = ["a"]
sOut4 _         = []

sIn4            :: NodeN -> [LinkN]
sIn4 "X"        = ["a"]
sIn4 _          = []

model4 = mkIOmodel (ndomain4,nchildren4,ldomain4,lchildren4,sOut4,sIn4)

-- ? printIOmodel model4
-- ("Out-In",[(("P",[[]],[[]]), ("Q",[[]],[[]]), ("U",[[]],[[]]), ("V",["a"],[[]]), ("X",[[]],[[]]), ("Y",[[]],[[]]))])
-- :: (String,[(NodeN,[LinkN],[LinkN])])

-- ? compactModel model4
-- True :: Bool

-- ? printIOmodel (abstractIO model4)
-- ("Out-In",[(("P",[[]],[[]]), ("Q",[[]],[[]]), ("U",[[]],[[]]), ("V",["a"],[[]]), ("X",[[]],[[]]), ("Y",[[]],[[]]))])
-- :: (String,[(NodeN,[LinkN],[LinkN])])

```

```

-- Test figures for half flow variations -----

ndomain5      :: DomainN
ndomain5      = ["R", "S", "T", "U", "V", "W"]

nchildren5    :: ChildrenN
nchildren5 "S" = ["R", "T"]
nchildren5 "T" = ["U"]
nchildren5 "U" = ["V"]
nchildren5 "W" = ["V"]
nchildren5 _   = []

ldomain5      :: DomainN
ldomain5      = ["a"]

lchildren5    :: ChildrenN
lchildren5 "a" = []
lchildren5 _   = []

sOut5         :: NodeN -> [LinkN]
sOut5 "R"     = ["a"]
sOut5 "T"     = ["a"]
sOut5 _       = []

sIn5          :: NodeN -> [LinkN]
sIn5 "W"      = ["a"]
sIn5 _        = []

model5 = mkIOmodel (ndomain5, nchildren5, ldomain5, lchildren5, sOut5, sIn5)

-- ? printIOmodel model5
-- ("Out-In", [{"R", ["a"], []}, {"S", [], []}, {"T", ["a"], []}, {"U", [], []}, {"V", [], []}, {"W", [], ["a"]}]) :: (String, [(NodeN, [LinkN], [LinkN])])

-- ? compactModel model5
-- True :: Bool

-- ? printIOmodel (abstractIO model5)
-- ("Out-In", [{"R", ["a"], []}, {"S", ["a"], []}, {"T", ["a"], []}, {"U", [], []}, {"V", [], []}, {"W", [], ["a"]}]) :: (String, [(NodeN, [LinkN], [LinkN])])

-- Test figures for diamond node order variations I -----

ndomain6a     :: DomainN
ndomain6a     = ["P", "Q", "R", "S", "T", "U", "V", "W", "X"]

nchildren6a   :: ChildrenN
nchildren6a "P" = ["Q"]
nchildren6a "Q" = ["R", "S"]
nchildren6a "R" = ["T"]
nchildren6a "S" = ["U"]
nchildren6a "T" = ["V", "W"]
nchildren6a "U" = ["W"]
nchildren6a "W" = ["X"]
nchildren6a _   = []

ldomain6a     :: DomainN
ldomain6a     = ["a"]

lchildren6a   :: ChildrenN
lchildren6a "a" = []
lchildren6a _   = []

```



```

sOut6a      :: NodeN -> [LinkN]
sOut6a "T"  = ["a"]
sOut6a "V"  = ["a"]
sOut6a "X"  = ["a"]
sOut6a _    = []

sIn6a       :: NodeN -> [LinkN]
sIn6a "U"   = ["a"]
sIn6a _     = []

model6a = mkIOmodel (ndomain6a,nchildren6a,ldomain6a,lchildren6a,sOut6a,sIn6a)

-- ? printIOmodel model6a
-- ("Out-In",[(("P",[],[]), ("Q",[],[]), ("R",[],[]), ("S",[],[]), ("T",["a"],[]),
("U",[],["a"])),
-- ("V",["a"],[]), ("W",[],[]), ("X",["a"],[])]) :: (String,[(NodeN,[LinkN],[LinkN])])

-- ? compactModel model6a
-- False :: Bool

-- ? printIOmodel (abstractIO model6a)
-- ("Out-In",[(("P",[],[]), ("Q",[],[]), ("R",["a"],[]), ("S",[],["a"]),
("T",["a"],[]),
-- ("U",[],["a"]), ("V",["a"],[]), ("W",["a"],[]), ("X",["a"],[])]) :: (String,
-- [(NodeN,[LinkN],[LinkN])])
-- Test figures for diamond node order variations II -----

ndomain6b   :: DomainN
ndomain6b   = ["P","Q","R","S","T","U","V","W","X"]

nchildren6b :: ChildrenN
nchildren6b "P" = ["Q"]
nchildren6b "Q" = ["R","S"]
nchildren6b "R" = ["T"]
nchildren6b "S" = ["U"]
nchildren6b "T" = ["V","W"]
nchildren6b "U" = ["W"]
nchildren6b "W" = ["X"]
nchildren6b _  = []

ldomain6b   :: DomainN
ldomain6b   = ["a"]

lchildren6b :: ChildrenN
lchildren6b "a" = []
lchildren6b _  = []

sOut6b      :: NodeN -> [LinkN]
sOut6b "T"  = ["a"]
sOut6b _    = []

sIn6b       :: NodeN -> [LinkN]
sIn6b "U"   = ["a"]
sIn6b _     = []

model6b = mkIOmodel (ndomain6b,nchildren6b,ldomain6b,lchildren6b,sOut6b,sIn6b)

-- ? printIOmodel model6b
-- ("Out-In",[(("P",[],[]), ("Q",[],[]), ("R",[],[]), ("S",[],[]), ("T",["a"],[]),
("U",[],["a"])),
-- ("V",[],[]), ("W",[],[]), ("X",[],[])])

-- ? compactModel model6b
-- True :: Bool

```

```

-- ? printIOmodel (abstractIO model6b)
-- ("Out-In", [{"P", [], []}, {"Q", [], []}, {"R", ["a"], []}, {"S", [], ["a"]},
{"T", ["a"], []},
-- {"U", [], ["a"]}, {"V", [], []}, {"W", [], []}, {"X", [], []}])

-----
-- Checking closure property for compact mapping -----
--
-- ? compactModel (compactIO model6a)
-- True :: Bool
-- (41409 reductions, 63648 cells)

-----
-- Checking closure property for abstract mapping -----
--
-- ? abstractModel (abstractIO model1)
-- True :: Bool
-- (7510 reductions, 11934 cells)
-- ? abstractModel (abstractIO model2)
-- True :: Bool
-- (6678 reductions, 10823 cells)
-- ? abstractModel (abstractIO model3)
-- True :: Bool
-- (6502 reductions, 10572 cells)
-- ? abstractModel (abstractIO model4)
-- True :: Bool
-- (7665 reductions, 12185 cells)
-- ? abstractModel (abstractIO model5)
-- True :: Bool
-- (9058 reductions, 14302 cells)
-- ? abstractModel (abstractIO model6a)
-- True :: Bool
-- (24536 reductions, 36466 cells, 1 garbage collection)

-- Model composition I -----

ndomain7      :: DomainN
ndomain7      = ["P", "Q", "R", "S", "T", "U"]

nchildren7    :: ChildrenN
nchildren7 "P" = ["Q", "T"]
nchildren7 "Q" = ["R"]
nchildren7 "R" = ["S"]
nchildren7 "T" = ["U"]
nchildren7 _  = []

ldomain7      :: DomainN
ldomain7      = ["a", "b"]

lchildren7    :: ChildrenN
lchildren7 "a" = ["b"]
lchildren7 _  = []

sOut7         :: NodeN -> [LinkN]
sOut7 "Q"     = ["b"]
sOut7 "S"     = ["b"]
sOut7 _      = []

sIn7          :: NodeN -> [LinkN]
sIn7 "X"      = []
sIn7 _       = []

```

```

model7 = makePC (mkIOmodel (ndomain7,nchildren7,ldomain7,lchildren7,sOut7,sIn7))

ldomain8      :: DomainN
ldomain8      = ["a","c"]

lchildren8    :: ChildrenN
lchildren8 "a" = ["c"]
lchildren8 _  = []

sOut8         :: NodeN -> [LinkN]
sOut8 "R"     = ["c"]
sOut8 _      = []

sIn8          :: NodeN -> [LinkN]
sIn8 "U"     = ["a"]
sIn8 _      = []

model8 = makePC (mkIOmodel (ndomain7,nchildren7,ldomain8,lchildren8,sOut8,sIn8))
model9 = unionModelC model7 model8

-- ? printPCmodel model7
-- ("Prod-Con",[(("a",[],[ ]), ("b",["Q", "S"],[ ]))])
-- ? printPCmodel model8
-- ("Prod-Con",[(("a",[,"U"]), ("c",["R"],[ ]))])

-- ? printOrders model9
-- ("NODE Order",[(("P",["Q", "T"]), ("Q",["R"]), ("R",["S"]), ("S",[ ]), ("T",["U"]),
-- ("U",[ ]), "LINK Order",[(("a",["b", "c"]), ("b",[ ]), ("c",[ ]))])

-- ? printPCmodel model9
-- ("Prod-Con",[(("a",[],[ ]), ("b",["S"],["U"]), ("c",["R"],["U"])]))

-- ? printPCmodel (abstractPC model9)
-- ("Prod-Con",[(("a",["Q", "R"],["T", "U"]), ("b",["Q", "R", "S"],["T", "U"]),
-- ("c",["Q", "R"],["T", "U"])]))
-- (31547 reductions, 50111 cells, 1 garbage collection)

-- Model composition II -----

ndomain10     :: DomainN
ndomain10     = ["R","S","T","V","W"]

nchildren10   :: ChildrenN
nchildren10 "S" = ["R","T"]
nchildren10 "T" = ["V"]
nchildren10 "W" = ["V"]
nchildren10 _  = []

ldomain10     :: DomainN
ldomain10     = ["a"]

lchildren10   :: ChildrenN
lchildren10 _ = []

sOut10        :: NodeN -> [LinkN]
sOut10 "R"    = ["a"]
sOut10 _     = []

sIn10         :: NodeN -> [LinkN]
sIn10 _      = []

model10 = makePC (mkIOmodel
(ndomain10,nchildren10,ldomain10,lchildren10,sOut10,sIn10))

```

```

modell10a = abstractPC modell10

sOut11      :: NodeN -> [LinkN]
sOut11 _    = []

sIn11       :: NodeN -> [LinkN]
sIn11 "V"   = ["a"]
sIn11 _    = []
modell11 = makePC (mkIOmodel
  (ndomain10, nchildren10, ldomain10, lchildren10, sOut11, sIn11))
modell11a = abstractPC modell11

sOut12      :: NodeN -> [LinkN]
sOut12 _    = []

sIn12       :: NodeN -> [LinkN]
sIn12 "W"   = ["a"]
sIn12 _    = []

modell12 = makePC (mkIOmodel
  (ndomain10, nchildren10, ldomain10, lchildren10, sOut12, sIn12))
modell12a = abstractPC modell12

modell13 = (modell11 `unionModelC` modell10) `unionModelC` modell12
modell14 = modell11 `unionModelC` (modell10 `unionModelC` modell12)
modell13a = (modell11a `unionModelA` modell10a) `unionModelA` modell12a
modell14a = modell11a `unionModelA` (modell10a `unionModelA` modell12a)
modell13_ = (modell11a `unionModel_` modell10a) `unionModel_` modell12a
modell14_ = modell11a `unionModel_` (modell10a `unionModel_` modell12a)

-- ? printPCmodel modell10
-- ("Prod-Con", [{"a", ["R"], []}])
-- ? printPCmodel modell11
-- ("Prod-Con", [{"a", [], ["V"]}])
-- ? printPCmodel modell12
-- ("Prod-Con", [{"a", [], ["W"]}])

-- ? printOrders modell10
-- ("NODE Order", [{"R", []}, {"S", ["R", "T"]}, {"T", ["V"]}, {"V", []}, {"W", ["V"]}])
-- "LINK Order", [{"a", []}])

-- ? printPCmodel modell13
-- ("Prod-Con", [{"a", ["R"], ["V"]}])
-- (15971 reductions, 25952 cells)
-- ? printPCmodel modell14
-- ("Prod-Con", [{"a", ["R"], ["V"]}])
-- (15776 reductions, 25278 cells, 1 garbage collection)

-- ? printPCmodel modell13a
-- ("Prod-Con", [{"a", ["R"], ["T", "V", "W"]}])
-- (11658 reductions, 18548 cells)
-- ? printPCmodel modell14a
-- ("Prod-Con", [{"a", ["R", "S"], ["T", "V", "W"]}])
-- (11560 reductions, 18350 cells)

-- This example demonstrates that abstract composition is not associative.

-- ? printPCmodel (modell11a `unionModelA` modell10a)
-- ("Prod-Con", [{"a", ["R"], ["T", "V", "W"]}])
-- ? printPCmodel (modell10a `unionModelA` modell12a)
-- ("Prod-Con", [{"a", ["R", "S"], ["W"]}])

```

```

-- ? printPCmodel model13_
-- ("Prod-Con",[(("a",["R"],["V", "W"])]))
-- ? printPCmodel model14_
-- ("Prod-Con",[(("a",["R"],["V", "W"])]))

-- ? printPCmodel (abstractPC model13_)
-- ("Prod-Con",[(("a",["R"],["T", "V", "W"])]))

-- Model composition III -----

ndomain15      :: DomainN
ndomain15      = ["R", "S", "T", "U", "V", "W", "X"]

nchildren15    :: ChildrenN
nchildren15 "R" = ["T"]
nchildren15 "S" = ["U"]
nchildren15 "T" = ["V", "W"]
nchildren15 "U" = ["W"]
nchildren15 "W" = ["X"]
nchildren15 _  = []

ldomain15      :: DomainN
ldomain15      = ["a"]

lchildren15    :: ChildrenN
lchildren15 "a" = []
lchildren15 _  = []

sOut15         :: NodeN -> [LinkN]
sOut15 "X"     = ["a"]
sOut15 _      = []

sIn15          :: NodeN -> [LinkN]
sIn15 _       = []

model15 = makePC (mkIOmodel
  (ndomain15,nchildren15,ldomain15,lchildren15,sOut15,sIn15))
model15a = abstractPC model15

sOut16         :: NodeN -> [LinkN]
sOut16 "T"     = ["a"]
sOut16 _      = []

sIn16          :: NodeN -> [LinkN]
sIn16 "U"     = ["a"]
sIn16 _       = []

model16 = makePC (mkIOmodel
  (ndomain15,nchildren15,ldomain15,lchildren15,sOut16,sIn16))
model16a = abstractPC model16

sOut17         :: NodeN -> [LinkN]
sOut17 "R"     = ["a"]
sOut17 _      = []

sIn17          :: NodeN -> [LinkN]
sIn17 "S"     = ["a"]
sIn17 _       = []

model17 = makePC (mkIOmodel
  (ndomain15,nchildren15,ldomain15,lchildren15,sOut17,sIn17))

```

```

modell17a = abstractPC modell17

modell18 = (modell15 `unionModelC` modell16) `unionModelC` modell17
modell19 = modell15 `unionModelC` (modell16 `unionModelC` modell17)
modell18a = (modell15a `unionModelA` modell16a) `unionModelA` modell17a
modell19a = modell15a `unionModelA` (modell16a `unionModelA` modell17a)
modell18_ = (modell15a `unionModel_` modell16a) `unionModel_` modell17a
modell19_ = modell15a `unionModel_` (modell16a `unionModel_` modell17a)

-- ? printPCmodel modell15
-- ("Prod-Con", [{"a", ["X"], []}])
-- ? printPCmodel modell16
-- ("Prod-Con", [{"a", ["T"], ["U"]}])
-- ? printPCmodel modell17
-- ("Prod-Con", [{"a", ["R"], ["S"]}])

-- ? printOrders modell15
-- ("NODE Order", [{"R", ["T"]}, {"S", ["U"]}, {"T", ["V", "W"]}, {"U", ["W"]}, {"V", []},
-- {"W", ["X"]}, {"X", []}], "LINK Order", [{"a", []}])

-- ? printPCmodel (modell15 `unionModelC` modell16)
-- ("Prod-Con", [{"a", ["T", "X"], ["U"]}])
-- ? printPCmodel (modell16 `unionModelC` modell17)
-- ("Prod-Con", [{"a", ["T"], ["U"]}])

-- ? printPCmodel modell18
-- ("Prod-Con", [{"a", ["R", "X"], ["S", "U"]}])
-- ? printPCmodel modell19
-- ("Prod-Con", [{"a", ["T", "X"], ["U"]}])

-- This example demonstrates that compact composition is not associative.

-- ? printPCmodel modell18a
-- ("Prod-Con", [{"a", ["R", "T", "W", "X"], ["S", "U"]}])
-- (35020 reductions, 53326 cells, 1 garbage collection)
-- ? printPCmodel modell19a
-- ("Prod-Con", [{"a", ["R", "T", "W", "X"], ["S", "U"]}])
-- (18575 reductions, 28042 cells)

-- ? printPCmodel modell18_
-- ("Prod-Con", [{"a", ["R", "T", "X"], ["S", "U"]}])
-- ? printPCmodel modell19_
-- ("Prod-Con", [{"a", ["R", "T", "X"], ["S", "U"]}])

-- ? printPCmodel (abstractPC modell18_)
-- ("Prod-Con", [{"a", ["R", "T", "W", "X"], ["S", "U"]}])

-- This example demonstrates that raw composition is associative here.

```

---

## A Console Based Structured Graph Tool

---

This appendix demonstrates a structured graph tool, implemented in C++ which is included on the floppy disc as SG\_demo. This program runs on a Macintosh computer with at least a 68020 processor.

A summary of the program commands is followed by a screen shot of a small example. A larger example is then given. We take a model containing eight DFDs and we move a process to a new location. Diagrammatic representations of the before and after models are given, followed by a log of the move-operation performed with the C++ tool.

### D.1 Summary of commands

#### STRUCTURED GRAPH DEMO COMMANDS

Adding and removing nodes/links from the model:

```

mn {text label}      - add a new node
nl {text label}      - add a new link
fn {node label}      - free a node
fl {link label}      - free a link

```

Updating the node/link orders:

```

an nodeA { nodeB }  - make each nodeB a child of nodeA
al linkA { linkB }  - make each linkB a child of linkA
dn nodeA { nodeB }  - delete each nodeB from nodeA's children
dl linkA { linkB }  - delete each linkB from linkA's children

cn { nodeA }        - cut each nodeA from its parents
cl { linkA }        - cut each linkA from its parents
mN nodeA nodeB      - move nodeA under nodeB"
cN linkA nodeB      - collapse nodeA into one of its parents nodeB"

pn                  - print the node order
pl                  - print the link order

```

Updating the Input/Output relations:

```

ao nodeA { linkA }  - make each linkA an output of nodeA
ai nodeA { linkA }  - make each linkA an input of nodeA
do nodeA { linkA }  - remove each linkA from nodeA's outputs
di nodeA { linkA }  - remove each linkA from nodeA's inputs
dO nodeA { linkA }  - remove each linkA from nodeA's descendants' outputs
dI nodeA { linkA }  - remove each linkA from nodeA's descendants' inputs

po                  - print the output relation
pi                  - print the input relation

```

Selecting node/links for later viewing of the full model,  
the (default is all nodes and links):

```

sn { nodeA }        - select each nodeA for viewing

```

```

sl { linkA }           - select each linkA for viewing
sN { nodeA }          - select all the nodes below each nodeA for viewing
sL { linkA }          - select all the links below each linkA for viewing

v                     - view the original input and output relations"
vF                    - view the FULL input and output relations"
vf                    - view the FULL input and output relations (with Flat I/O)"

```

Preserving a model in a file:

```

s filename            - save the current model into the nominated file
l filename            - load the current model from the nominated file (unreliable)

```

General information:

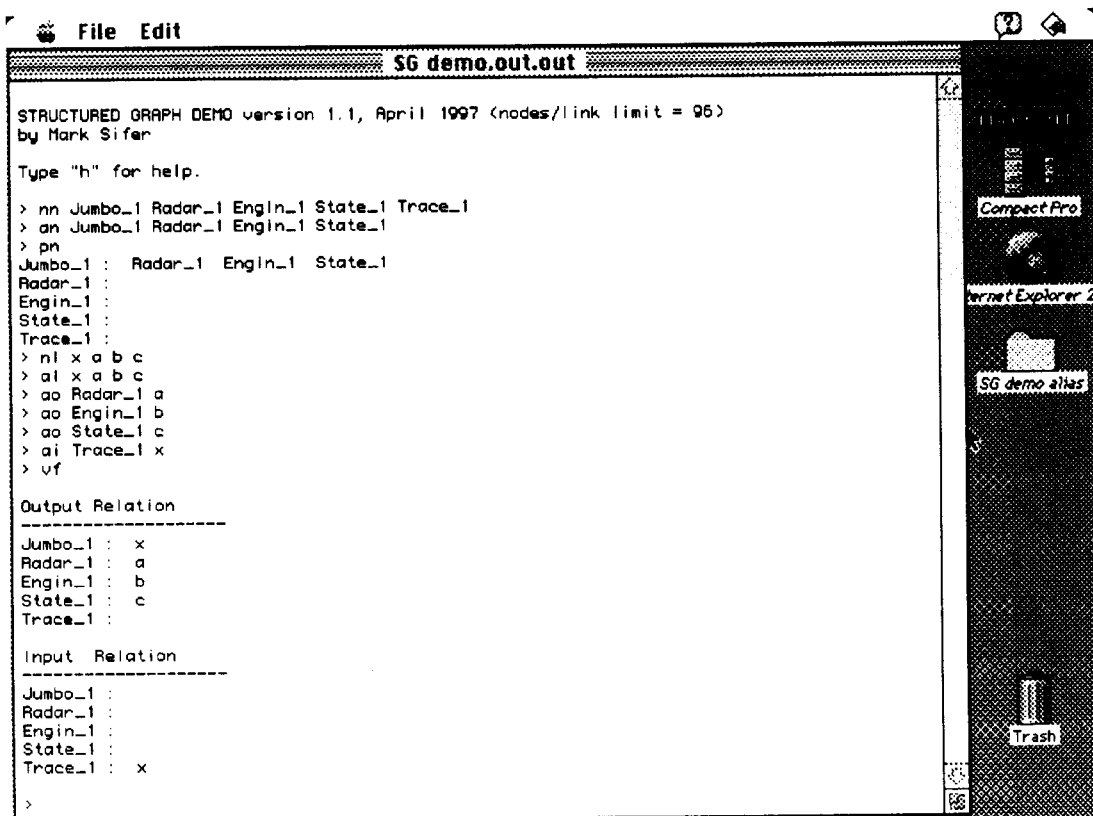
```

h                    - displays a summary of all commands

```

## D.2 A mini browsing example

The following screen shot shows the construction of a model. The nodes `Jumbo_1`, `Radar_1`, `Engin_1`, `State_1` and `Trace_1` are added. Node `Jumbo_1` is then made the parent of `Radar_1`, `Engin_1` and `State_1`. Links `x`, `a`, `b` and `c` are added. Link `x` is made the parent of `a`, `b` and `c`. `Radar_1` is made a producer of `a`, `Engin_1` is made a producer of `b`, and `State_1` is made a producer of `c`. `Trace_1` is made a consumer of `x`. The final command, `vf`, shows the net interfaces of the built model. This model is in fact, the `Jumb1` submodel shown in figure D.1.



```

File Edit
SG demo.out.out
STRUCTURED GRAPH DEMO version 1.1, April 1997 (nodes/link limit = 96)
by Mark Sifer

Type "h" for help.

> nn Jumbo_1 Radar_1 Engin_1 State_1 Trace_1
> an Jumbo_1 Radar_1 Engin_1 State_1
> pn
Jumbo_1 : Radar_1 Engin_1 State_1
Radar_1 :
Engin_1 :
State_1 :
Trace_1 :
> nl x a b c
> al x a b c
> ao Radar_1 a
> ao Engin_1 b
> ao State_1 c
> ai Trace_1 x
> vf

Output Relation
-----
Jumbo_1 : x
Radar_1 : a
Engin_1 : b
State_1 : c
Trace_1 :

Input Relation
-----
Jumbo_1 :
Radar_1 :
Engin_1 :
State_1 :
Trace_1 : x

>

```



### D.3 An editing example

Figure D.1 shows a model before and after a move node. Figures D.2 and D.3 show the same before-and-after models respectively, but as a collection of DFDs and data dictionary. Finally a log of the move-operation performed with the C++ tool is presented.

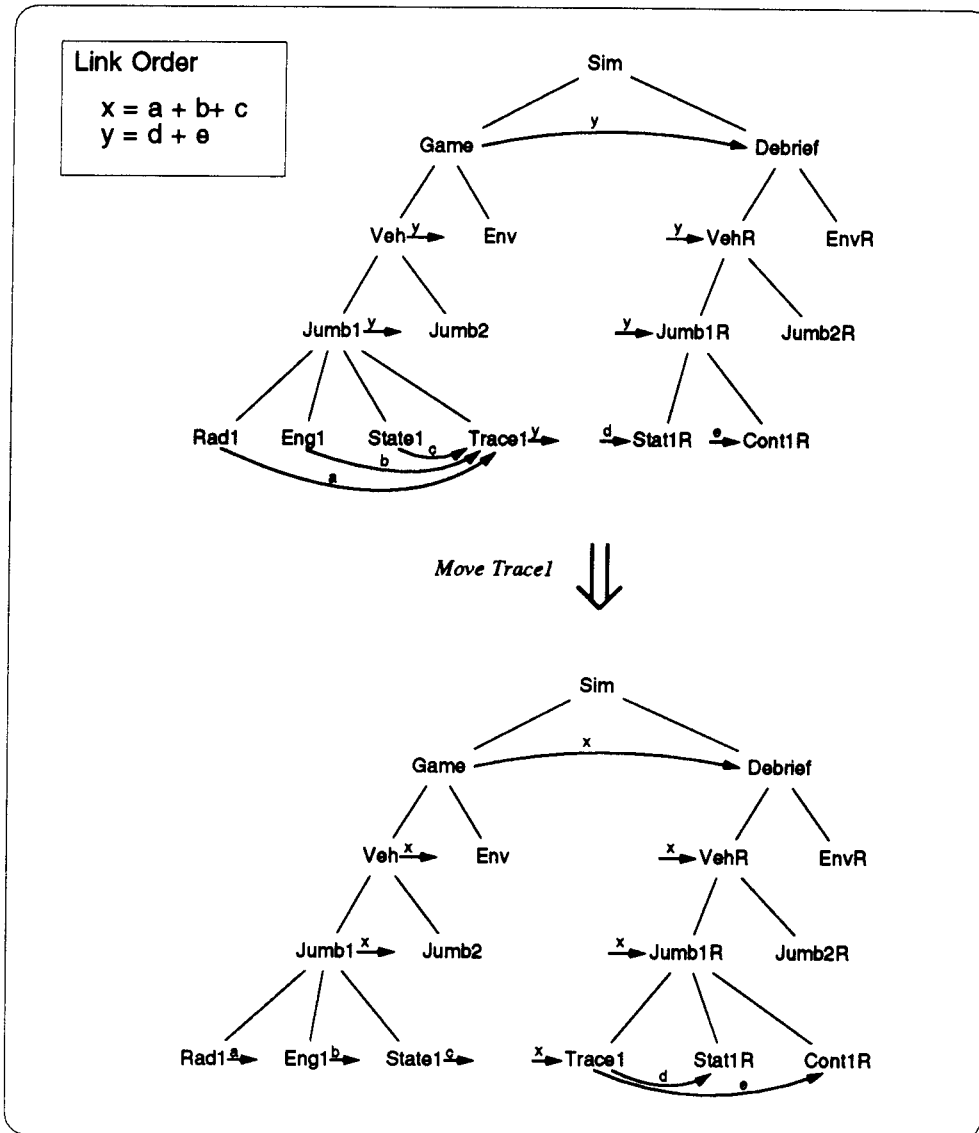


Figure D.1 A model before and after moving a node

In figure D.1 the node Trace1 which appears under Jumb1, is moved under a new parent, node Jumb1R. The nodes Game, Veh and Jumb1 which produced link y, produce link x after the move. The nodes Debrief, VehR and Jumb1R which consumed link y, consume link x after the move. However, from the user's perspective, as we shall see in the log presented later, this move requires only one operation whilst keeping the model balanced.

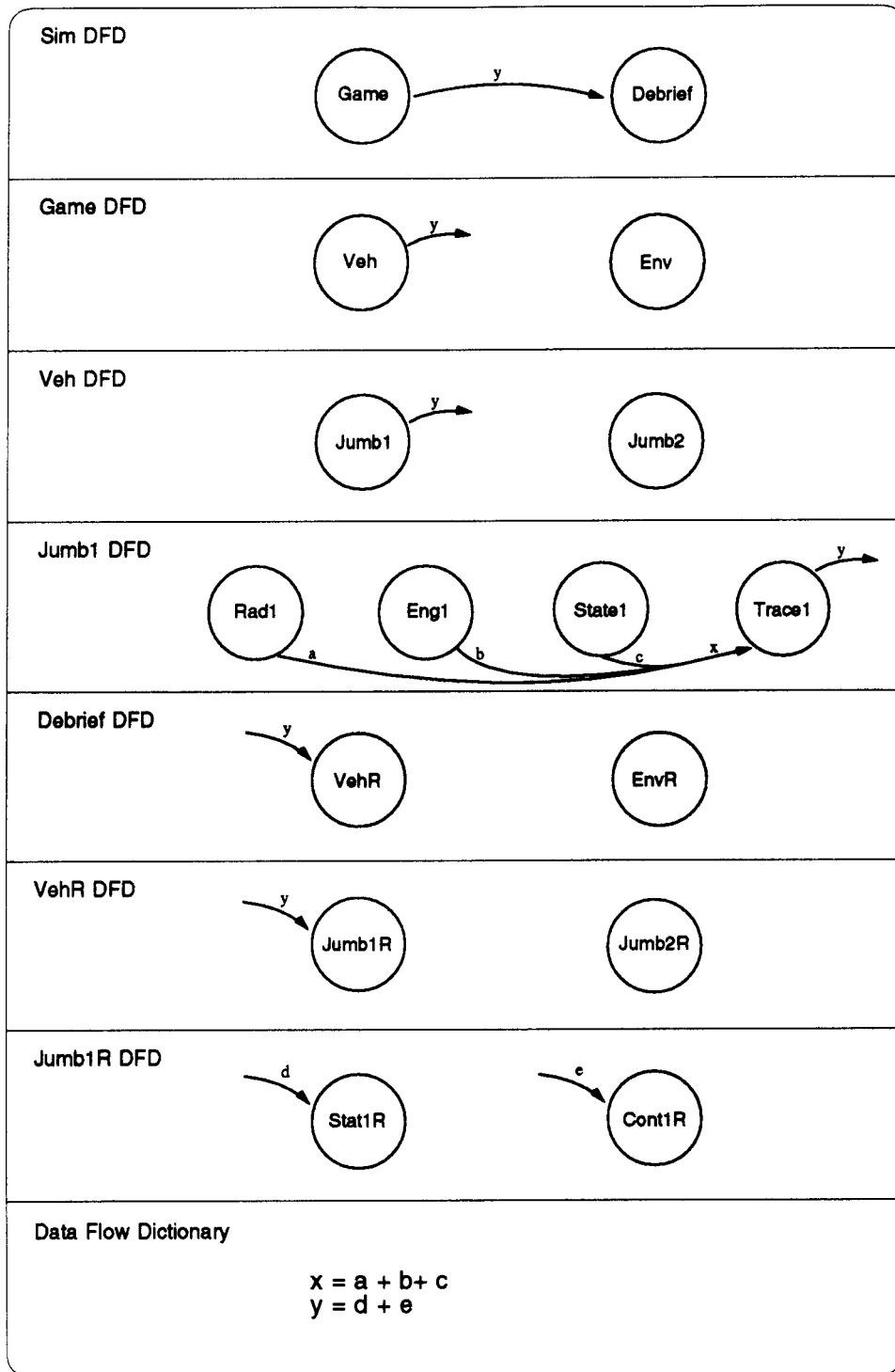


Figure D.2 Model DFDs before moving a node

Figure D.2 shows the same model presented in the upper part of figure D.1. Figure D.2 uses the same style as Chapter Two, a collection of DFDs and a data dictionary, whilst figure D.1 uses the style of Chapter Three.

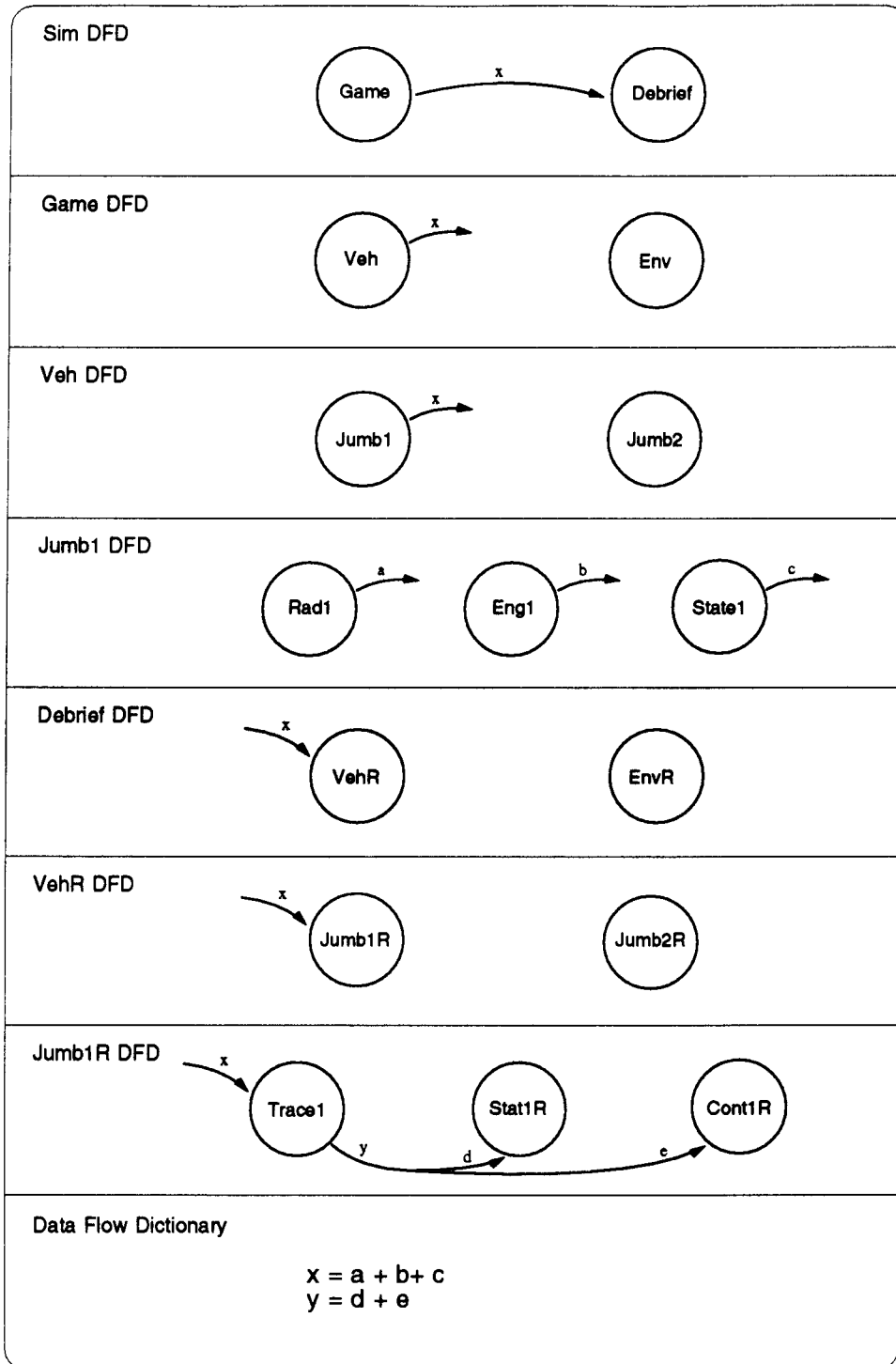


Figure D.3 Model DFDs after moving a node

Figure D.3 shows the new DFDs after the move of process Trace1 to Jumb1R was performed. Though only one operation was performed (the move operation), all seven DFDs changed. If the user wished to perform this move using a structured analysis tool such as Teamwork, seven DFDs would need to be updated, to keep the model balanced. This further demonstrates the scalability of a tool based on structured graphs.

Using the C++ tool, the before-model shown in figure D.1 was built incrementally, followed by the move of Trace1. The log for this follows, with the move command highlighted:

```
STRUCTURED GRAPH DEMO version 1.1, April 1997 (nodes/link limit = 96)
by Mark Sifer

Type "h" for help.

> nn Simultr Game_pl Debrief Vehicle Environ VehiclR EnviroR Jumbo_1 Jumbo_2
> nn Jumbo1R Jumbo2R Radar_1 Engin_1 State_1 Trace_1 State1R Contr1R
> an Simultr Game_pl Debrief
> an Game_pl Vehicle Environ
> an Vehicle Jumbo_1 Jumbo_2
> an Jumbo_1 Radar_1 Engin_1 State_1 Trace_1
> an Debrief VehiclR EnviroR
> an VehiclR Jumbo1R Jumbo2R
> an Jumbo1R State1R Contr1R
> pn
Simultr : Game_pl Debrief
Game_pl : Vehicle Environ
Debrief : VehiclR EnviroR
Vehicle : Jumbo_1 Jumbo_2
Environ :
VehiclR : Jumbo1R Jumbo2R
EnviroR :
Jumbo_1 : Radar_1 Engin_1 State_1 Trace_1
Jumbo_2 :
Jumbo1R : State1R Contr1R
Jumbo2R :
Radar_1 :
Engin_1 :
State_1 :
Trace_1 :
State1R :
Contr1R :
> nl x a b c y d e
> al x a b c
> al y d e
> pl
x      : a b c
a      :
b      :
c      :
y      : d e
d      :
e      :
> ao Radar_1 a
> ao Engin_1 b
> ao State_1 c
> ao Trace_1 y
> ai Trace_1 x
> ai State1R d
> ai Contr1R e
> v
```

Output Relation

```
-----
Simultr :
Game_pl :
Debrief :
```

```

Vehicle :
Environ :
VehiclR :
EnviroR :
Jumbo_1 :
Jumbo_2 :
Jumbo1R :
Jumbo2R :
Radar_1 : a
Engin_1 : b
State_1 : c
Trace_1 : d e
State1R :
Contr1R :

```

```
Input Relation
```

```

-----
Simultr :
Game_pl :
Debrief :
Vehicle :
Environ :
VehiclR :
EnviroR :
Jumbo_1 :
Jumbo_2 :
Jumbo1R :
Jumbo2R :
Radar_1 :
Engin_1 :
State_1 :
Trace_1 : a b c
State1R : d
Contr1R : e

```

```
> vf
```

```
Output Relation
```

```

-----
Simultr :
Game_pl : y
Debrief :
Vehicle : y
Environ :
VehiclR :
EnviroR :
Jumbo_1 : y
Jumbo_2 :
Jumbo1R :
Jumbo2R :
Radar_1 : a
Engin_1 : b
State_1 : c
Trace_1 : y
State1R :
Contr1R :

```

```
Input Relation
```

```

-----
Simultr :
Game_pl :
Debrief : y
Vehicle :
Environ :
VehiclR : y

```

```

EnviroR :
Jumbo_1 :
Jumbo_2 :
Jumbo1R : y
Jumbo2R :
Radar_1 :
Engin_1 :
State_1 :
Trace_1 : x
State1R : d
Contr1R : e

```

```
> s example1
```

```
> mN Trace_1 Jumbo1R
```

```
> pn
```

```

Simultr : Game_pl Debrief
Game_pl : Vehicle Environ
Debrief : VehiclR EnviroR
Vehicle : Jumbo_1 Jumbo_2
Environ :
VehiclR : Jumbo1R Jumbo2R
EnviroR :
Jumbo_1 : Radar_1 Engin_1 State_1
Jumbo_2 :
Jumbo1R : Trace_1 State1R Contr1R
Jumbo2R :
Radar_1 :
Engin_1 :
State_1 :
Trace_1 :
State1R :
Contr1R :
> v

```

```
Output Relation
```

```

-----
Simultr :
Game_pl :
Debrief :
Vehicle :
Environ :
VehiclR :
EnviroR :
Jumbo_1 :
Jumbo_2 :
Jumbo1R :
Jumbo2R :
Radar_1 : a
Engin_1 : b
State_1 : c
Trace_1 : d e
State1R :
Contr1R :

```

```
Input Relation
```

```

-----
Simultr :
Game_pl :
Debrief :
Vehicle :
Environ :
VehiclR :
EnviroR :
Jumbo_1 :
Jumbo_2 :

```

```

Jumbo1R :
Jumbo2R :
Radar_1 :
Engin_1 :
State_1 :
Trace_1 : a b c
State1R : d
Contr1R : e

```

```
> vf
```

#### Output Relation

```

-----
Simultr :
Game_pl : x
Debrief :
Vehicle : x
Environ :
VehiclR :
EnviroR :
Jumbo_1 : x
Jumbo_2 :
Jumbo1R :
Jumbo2R :
Radar_1 : a
Engin_1 : b
State_1 : c
Trace_1 : y
State1R :
Contr1R :

```

#### Input Relation

```

-----
Simultr :
Game_pl :
Debrief : x
Vehicle :
Environ :
VehiclR : x
EnviroR :
Jumbo_1 :
Jumbo_2 :
Jumbo1R : x
Jumbo2R :
Radar_1 :
Engin_1 :
State_1 :
Trace_1 : x
State1R : d
Contr1R : e

```

```
> s example2
```

```
> ?
```

```

nn {text} : new node          an parent {child} : add node children
nl {text} : new link          al parent {child} : add link children
fn {node} : free node         cn/cl {node/link} : cut nodes/links
fl {link} : free link         mN/cN node node   : move/collapse node

ao node {link} : add node outputs      pn : print node heirarchy
ai node {link} : add node inputs        pl : print link heirarchy
do/dO node {link} : delete node/s outputs po : print output relation
di/dI node {link} : delete node/s inputs pi : print input relation

sn/sN {node} : select nodes      v : view model          ?,h : help
sl/sL {link} : select links      vF : view FULL model   cE : compact edit

```

```
sR : selection reset          vf : view FLAT model    rE : raw edit
q  : quit                    s/l file : save/load file  ?E : edit mode
> sn Vehicle Trace_1 State1R Control1R
Argument [Control1R] is not a known node label
> sn Vehicle Trace_1 State1R Contr1R
> vf
```

Output Relation

```
-----
Vehicle : x
Trace_1  : y
State1R  :
Contr1R  :
```

Input Relation

```
-----
Vehicle :
Trace_1  : x
State1R  : d
Contr1R  : e
```

```
> s example1
> q
```

Program has completed