

AD-A197 658

DTIC FILE COPY



(4)

RADC-TR-88-71
Final Technical Report
May 1988

TARA: TOOL ASSISTED REQUIREMENTS ANALYSIS

University of London

J. Kramer, K. Whitehead, et al



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

88 8 23 032

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

ADA197658

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS N/A		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A			5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-71		
4. PERFORMING ORGANIZATION REPORT NUMBER(S) Imperial College Research Report Document 87/18			7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COEE)		
6a. NAME OF PERFORMING ORGANIZATION University of London Imperial College of Science		6b. OFFICE SYMBOL (If applicable)		7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	
6c. ADDRESS (City, State, and ZIP Code) and Technology 180 Queen's Gate London SW7 2BZ United Kingdom		8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F49620-85-C-0132	
8b. OFFICE SYMBOL (If applicable) COEE		10. SOURCE OF FUNDING NUMBERS			
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		PROGRAM ELEMENT NO. 63728F		TASK NO. 01	
		PROJECT NO. 2527		WORK UNIT ACCESSION NO. 03	
11. TITLE (Include Security Classification) TARA: TOOL ASSISTED REQUIREMENTS ANALYSIS					
12. PERSONAL AUTHOR(S) J. Kramer, K. Whitehead, et al					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM Sep 85 TO Sep 87		14. DATE OF REPORT (Year, Month, Day) May 1988	
15. PAGE COUNT 154					
16. SUPPLEMENTARY NOTATION N/A					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP			
15	07		requirements analysis active guidance		
12	05		requirements specification reuse		
			animation CORE		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Requirements analysis has been recognized as one of the most critical and difficult tasks in software engineering. The need for tool support is therefore easily justified. In this paper we describe an approach to the provision of such support for three particular aspects: method support by active guidance, validation by transaction animation, and reuse of specification fragments. Method guidance is supported by a method model used to describe the sequence of method steps that should be followed. This model is directly interpreted by the tools to provide advice and reasoning. It is used in conjunction with rules used for consistency checking to provide remedial advice. The animator provides facilities for the selection and execution of a transaction to reflect the specified behavior given a particular scenario. Actions are described in terms of input-out relations. Simple rules can be specified to control the execution of actions. Facilities are provided to replay and interact with transactions. Reuse is supported by facilities for identifying candidate transactions from a reuse (over)					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL William E. Rzepka			22b. TELEPHONE (Include Area Code) (315) 330-2762		22c. OFFICE SYMBOL RADC (COEE)

DD Form 1473, JUN 86

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

UNCLASSIFIED

database. The search strategies provided include browsing in an inheritance structure, different levels of pattern matching, casual chain matching (matching of the underlying control structures), and purpose matching. Support is then provided for the allocation of the selected fragment to the target environment.

The approach has been tested by implementing a prototype set of tools for the CORE method and the Analyst workstation. A major case study, the ASE (Advanced Sensor Exploitation) test environment, has been analyzed and specified using CORE, the Analyst, and the tools described above. The results of that work are described and evaluated.

Accession For	
NTIS ODA&I	☒
DTIC T/S	☐
Unannounced	☐
Justification	
By	
Distribution	
Availability Codes	
Dist	Availability or Special
A-1	



UNCLASSIFIED

Contents

	<u>Page</u>
Chapter 1: <u>Introduction</u>	1.1
1. Background	1.1
2. Objectives	1.1
2.1 Method Guidance	1.1
2.2 Animation	1.2
2.3 Re-use	1.2
2.4 General Observations	1.3
3. Methodology	1.3
3.1 CORE	1.3
3.2 The Analyst	1.4
4. Structure of this Document	1.6
Chapter 2: <u>Method Guidance</u>	2.1
1 The Approach	2.1
1.1 Method Model	2.1
1.2 Generating Advice	2.1
1.3 Analysis Status	2.2
2 Current Status	2.2
2.1 An ERA Model for Active Guidance	2.2
2.2 Method Model	2.4
2.3 Formal Description of CORE	2.6
2.4 Note Passing	2.8
2.5 Remediation Strategies	2.14
2.6 Advice-Giving Heuristics	2.15
2.7 Advice Attributes	2.15
2.8 Priority Factors	2.16
2.9 Presentation of advice	2.17
2.10 The Status of the Specification	2.19
3 Conclusions	2.22

Chapter 3: <u>Animation of Specifications</u>	3.1
1 The Approach	3.1
1.1 Transactions	3.1
1.2 Transaction Selection Strategies	3.2
1.3 Action Definitions	3.3
1.4 Data and Action (De)composition	3.4
2 Current Status of the Animator	3.5
2.1 Selection	3.5
2.2 Action Definitions	3.6
2.3 Executable Conditions of Actions	3.7
2.4 Channels and Pools	3.7
2.5 Transaction Replay	3.8
2.6 Graphical Interface	3.8
2.7 Timing Analysis	3.10
3 Conclusions	3.11
Chapter 4: <u>Re-use</u>	4.1
1 The Approach	4.1
1.1 Introduction	4.1
1.2 Reuse in Core	4.1
1.3 Analogy and Reuse	4.1
1.4 Model of Reuse	4.2
2 TRUE : The Reuse Tool	4.3
2.1 Application Concepts	4.3
2.2 Structure: Base and Target	4.3
2.3 Views	4.4
2.4 Selection Strategies	4.5
2.5 Allocation	4.11
2.6 Method	4.12
2.7 TRUE: A Tool for Transaction Reuse	4.12
3 Conclusions	4.13
Chapter 5: <u>Enhancements to the Core Analyst</u>	5.1
1 Analyst Enhancements	5.1
1.1 Release 2A Enhancements	5.1
1.2 Enhancements implemented after Release 2A	5.2
2 Prototype Designs and Tools	5.3
3 Superstructure Developments	5.4

Chapter 6: <u>The ASET Case Study</u>	6.1
1 ASET Objectives	6.1
2 Form of the Case Study	6.1
3 CORE Requirements Analysis - Limitations of the ASET Specification	6.2
4 The Analysis	6.2
5 Discussion of the use of Core	6.3
5.1 Viewpoint Structuring	6.3
5.2 Tabular Collection	6.3
5.3 Data Structuring	6.3
6 Discussion of the ASET Specification	6.3
6.1 Timing and Control	6.3
6.2 User Interface	6.4
7 Gains from the use of CORE	6.4
Chapter 7: <u>Conclusions and Further Work</u>	7.1
1 Method Guidance	7.1
1.1 Evaluation	7.1
1.2 Conclusions and further work	7.2
2 Animation	7.3
2.1 Evaluation	7.3
2.2 Conclusions and further work	7.5
3 Reuse	7.6
3.1 Evaluation	7.6
3.2 Conclusions	7.6

References

Appendix A: Formal Specification of CORE

Appendix B: Performance Animation

Appendix C: Demonstration Notes

Animator Demonstration Script

Method Guidance Demonstration

Chapter 1

Introduction

This document is the final report of the TARA Project on "Tool Assisted Requirements Analysis" conducted by Imperial College of Science and Technology and Systems Designers p.l.c. It contains a description of the objectives and achievements during this two year project and a discussion of future work.

In this chapter, the background to the project are explained and the objectives summarized.

1. Background

The project follows naturally from two previous efforts involving each of the collaborators separately: the RADC-funded "Executable Metric Model Applications" (EMMA) project at Imperial College; and the development of Systems Designers' p.l.c. product "Analyst", a CORE workstation based on the Apple Macintosh.

EMMA was concerned with the use of executable models of systems for the purpose of pre-implementation validation. One of the techniques examined in detail was the use of direct 'animation' of data flow specifications in Prolog [Bartlett, Cherrie, Lehman, MacLean and Potts, 1984].

Analyst is a CORE support tool, implemented in Prolog, which permits the CORE practitioner to enter CORE diagrams directly and analyze them individually and together for consistency, completeness and style [Stephens and Whitehead, 1985].

While we wished to retain general applicability, we made a commitment to use CORE and the Analyst as research vehicles in order to provide a practical focus for our work. That is, we have used both as starting points with the intention of extending them in a variety of ways to support our work.

2. Objectives

Requirements analysis is one of the most critical tasks in software engineering. Unrecognized errors made early in the development process may have widespread repercussions in the later phases. As a consequence, the cost of correcting such errors is high [Boehm, 1982]. Support for requirements analysis is therefore crucial.

We are interested in the role of automatically provided **method guidance** to support the use of requirements analysis methods, the ability to use computer tools to help clients and analysts visualize the behaviour of the specified system by **animation** of the specification — including some performance analysis, and the possibility of **reusing** specification fragments or parts of existing specifications in the composition of a new specification.

2.1 Method Guidance

Many requirements methods are in use in industry, and most practising systems analysts are familiar with one or two. The range of expertise, though, is vast. There are few real experts in a

given requirements analysis method in the same sense as a Pascal programmer with five years solid experience is a Pascal expert. Automated tool support for requirements analysis may not, therefore, benefit from the 'power tools' paradigm [Sheil, 1984]. Instead, a requirements analysis tool must be seen as an intelligent assistant that caters for users of widely varying degrees of expertise in the requirements method.

Requirements methods are systems of *recommended* procedures and are intended to supplement rather than replace an analyst's skill. Advice should be provided to support normal use of the method. However, a support tool that could not deal with deviations from the recommended method and treated them as 'errors' from which it could not recover, would be unacceptable. A crucial part of any active guidance system for a requirements method is the *remediation* mechanism whereby possible repair procedures are deduced and recommended to the practitioner. In addition, the guidance system should include some ordering or prioritization of advice between alternative actions, such as corrective actions before method steps.

2.2 Animation

Most approaches to requirements analysis are strong in their representation of structure, but weak when specifying processes. They usually produce a specification in terms of a composition of actions and data flows, but cannot reflect the intended behaviour in a dynamic, process form. This imbalance needs to be redressed by augmenting representations to provide further process information to support facilities such as specification animation.

Animation of a specification is the process of providing an indication of the *dynamic behaviour* specified by walking through a specification fragment in order to follow some scenario. Further process information for the actions can be added by specifying the mapping from inputs to outputs. Animation then involves (dynamically) stepping through some specification examining the resulting output behaviour for given inputs. This can be contrasted with analysis of static, structural properties on the one hand (such as given by data flow diagrams) and detailed prototyping on the other (such as the basis for an implementation). Animation can be used to determine causal relationships embedded in the specification, or simply as a means of browsing through the specification to ensure adequacy and accuracy by reflection of the specified behaviour back at the user. In particular, there is a need to permit reflection of specified behaviour under different circumstances (ie. animation replay with different data values). This is sometimes referred to as "what if ..." or consequence testing.

Animation is deliberately less exact and detailed than current work on either executable specifications, such as PAISLey [Zave 1982] or rapid prototyping, such as PDS [Klausner 1982]. Both can provide a more exact execution of the specification but require far more information and expertise. We feel that this is inappropriate to this level of specification, where such formality and detail may actually obscure the more general requirements that are being specified. That is not to say that we do not believe in those approaches, but rather that they should be used in later phases of system specification. Animation seems to provide the right balance for this level of requirements specification and for obtaining a reflection of its intended behaviour.

2.3 Reuse

As an ideal, the reuse of fragments of specifications is clearly desirable. The benefits in terms of cost and convenience are obvious. In addition, there is a reliability benefit in terms of the inclusion of previously 'validated' specifications. It is, however, acknowledged to be a very difficult problem requiring advanced technology.

There is a need to identify, characterise and retain specification fragments which are good candidates for future reuse. These form the base cases from which an analyst selects. Sophisticated

and versatile search strategies are necessary to select matching fragments for the target environment, even where the base and target application domains may be very different. Finally there is a requirement for tailoring reused specifications to suit the new environment.

Although reuse was not previously considered in the CORE method, the potential benefits are sufficient justification for an investigation of the support tools required for realistic specification reuse.

2.4 General Observations

The common theme that is apparent here is that errors that occur during requirements analysis are due to failures of **understanding**: requirements may be misunderstood because they are so complex that the client and practitioner have difficulty focusing attention on one aspect at a time and perceiving interactions between requirements, or because the specified system is impossible to visualize from the resulting specification. The concepts and prototype tools provided in the TARA project are specifically aimed at supporting the analyst in the analysis process and in visualizing the specification.

3. Methodology

As vehicles for this research we are using the requirements analysis method CORE [Mullery, 1979] and the CORE support tool the Analyst [Stephens and Whitehead, 1985]. Although it is our objective to provide tools and techniques that are tightly coupled to CORE and the Analyst, as it is only by making such a commitment that they can be properly evaluated, we intend to do so in a way that could be generalized to any informal diagrammatic requirements or early design method and its supporting analysis tools.

3.1 CORE

CORE is a widely used requirements analysis method in the UK. First documented in [Mullery, 1979], a comprehensive account is given in the manual [Systems Designers, 1986]. It is assumed that the reader is familiar with the spirit of formatted requirements analysis methods and has access to the CORE manual. The following brief summary should be all that is needed to understand the remainder of the report.

CORE is one of the few truly prescriptive methods available. It consists of a sequence of steps which elucidate the user view of the functional architecture of the proposed system and its operational environment, together with a limited amount of performance and reliability analysis. It provides techniques and notations for all phases of *elicitation*, *specification* and *analysis* of requirements and results in a structured, action/dataflow form of specification.

In CORE the constituent steps should be performed in a well-defined order. Figure 2.1 illustrates the grammar of a perfect CORE project in the form of a structure diagram [Jackson 1975]. (Two steps of CORE not directly supported by the Analyst have been omitted from the figure).

In the first step, the domain of discourse is partitioned into **viewpoints**. These entities are organizational, human, software or hardware components of the system *and its environment*. Some, **direct viewpoints**, are to be subject to further analysis. Others, **indirect viewpoints**, are of interest only as sources or destinations of data. To aid the understanding of complex systems, direct viewpoints are decomposed into sub-viewpoints recursively. The remaining steps of the method are repeated at each level of the viewpoint hierarchy.

In the remaining steps, the viewpoints at the current level are analysed further. In **tabular collection**, a viewpoint's responsibilities are represented as a set of actions. These are tabulated in a **tabular collection form** which lists the actions, their input and output data and the sources and destinations of the data. In **data structuring**, the output of each viewpoint is analyzed. A diagram is produced resembling a Jackson structure diagram (cf. Figure 2.1) which shows the legal sequencing of the outputs. Using the actions and their interfaces from the tabular collection form and the order of production of the outputs from the data structuring, the practitioner can now draw a **single viewpoint model (SVM)**, a data flow diagram, for each viewpoint. An SVM contains additional information, such as internal data flows, repetitive or optional actions and control flows. Information from several SVMs can be merged into a **combined viewpoint model (CVM)**. An arbitrary number of CVMs could be prepared for any complex system, so only actions that are pertinent for a particular transaction are selected for a given CVM.

Thus while CORE uses a reasonably rich set of representations, it is far more than a collection of representational techniques. The interrelationships between the representations are not simple, and the redundancy that is encouraged obliges the practitioner to perform a large number of consistency checks. At any point in a project it may be possible to proceed by performing a variety of method steps. CORE includes many heuristics at the strategic and tactical levels to help the practitioner decide which is best.

3.2 The Analyst

The Analyst Workstation, under development by Systems Designers p.l.c. [Stephens and Whitehead, 1985], is an interactive software tool which supports the CORE method. It provides a basic set of clerical tools for storing and presenting graphically CORE specifications. It includes rule-based syntactic and semantic checking implemented in the logic programming language, Prolog. For instance, syntactic checks for ill-formed diagrams are made before storage, and semantic checks (eg. data flows with no specified destination) can either be done continuously or on demand. Many of the pragmatic aspects described above are, or will be, supported by the Analyst.

Figure 2.2 shows an example of the interface and functionality of the Analyst. The Analyst runs on the Apple® Macintosh™ range of machines and, as can be seen, follows the Macintosh user interface paradigm. On Figure 2.2, the practitioner is creating a tabular collection form and has requested, using the 'Check' menu, that semantic checking is to be performed continuously. A new destination for data has been entered - 'Emergency Services' - and one of the analysis rules has discovered that this viewpoint has not been entered in the viewpoint hierarchy. This causes the practitioner to be informed as shown. The use of the two buttons, Ignore and Cancel will be explained later.

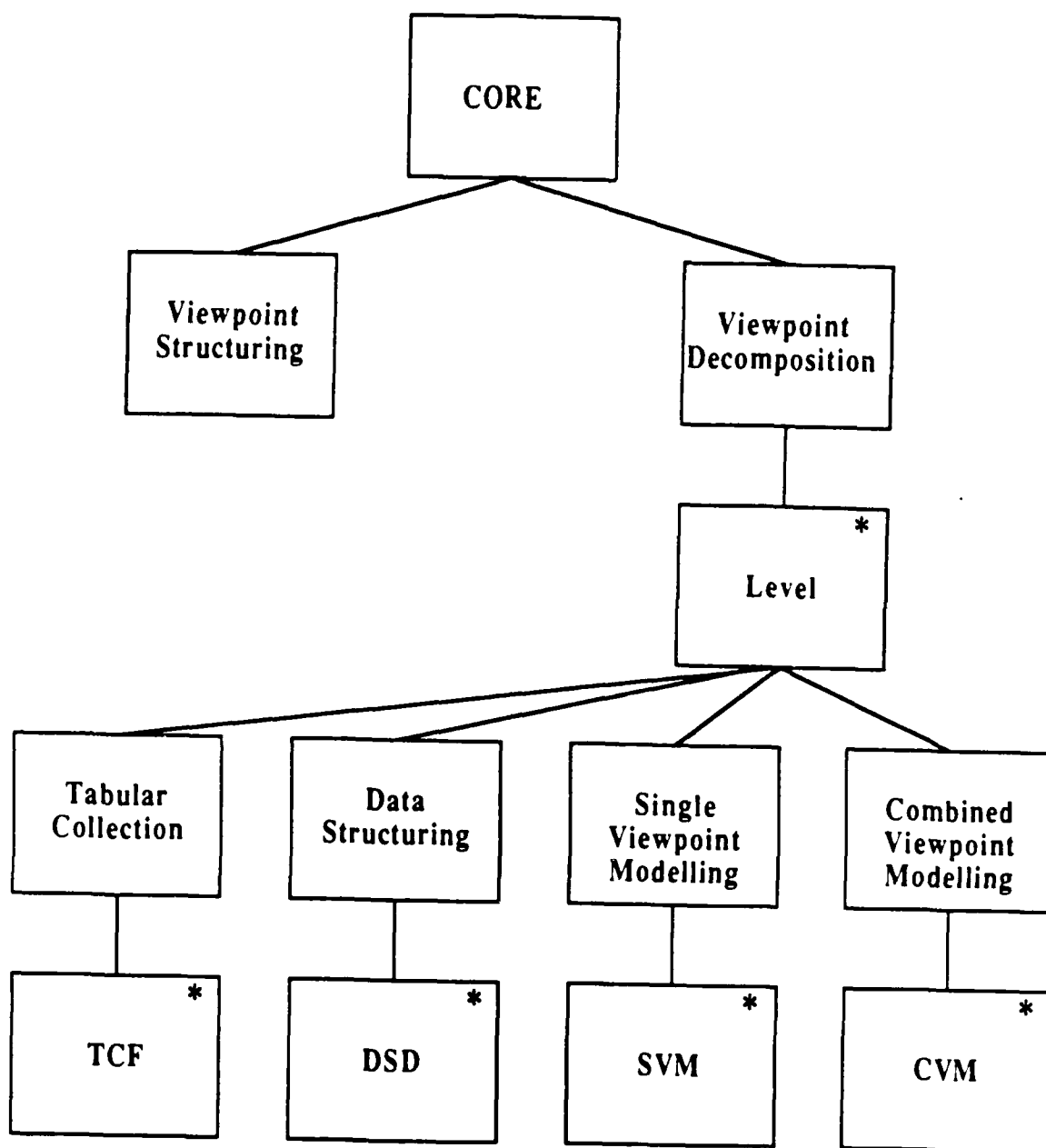


Figure 2.1

DSD depicting the steps of CORE

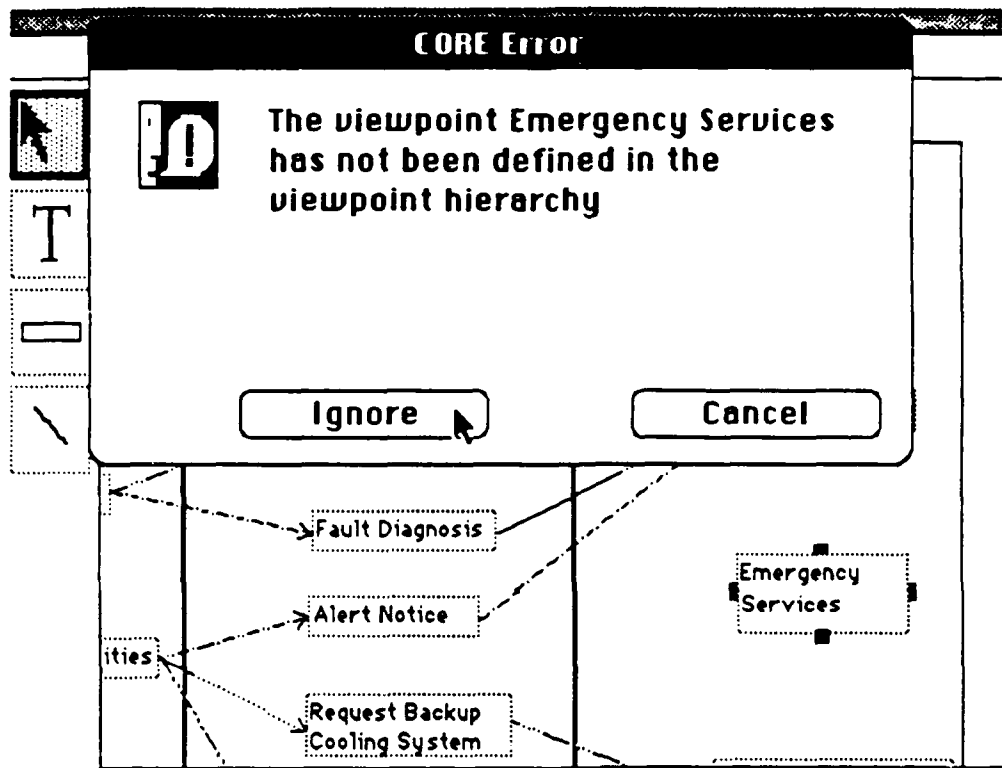


Figure 2.2

4. Structure of this Report

The remainder of this document describes the work done in each of the areas described in the objectives. Chapter 2 deals with method guidance, Chapter 3 with animation and Chapter 4 with the reuse of CORE specification fragments. Chapter 5 deals with more general Analyst enhancements provided during the project and to aid tool integration. Chapter 6 provides a brief discussion of the experience of specifying the ASET case study in CORE using the Analyst. Chapter 7 provides some conclusions from an evaluation of the toolset, and gives a brief indication of future work.

Appendix A contains a formal specification of a large part of CORE written in modal action logic which serves as a method model. Appendix B describes and illustrates the use of performance animation for timing analysis. Appendix C provides a set of notes describing a demonstration.

The User guides for each of the tools in the TARA toolset, and the full ASET Requirements Specification produced from the Analyst are in separate documents.

Chapter 2

METHOD GUIDANCE

1 The Approach

1.1 Method Model

Method advice should provide support for both the normal, prescribed use of the method, and remedial advice when errors or inconsistencies are detected. To provide **normative** and **remedial** advice, an active guidance system must maintain an internal model of the method. Rather than being hard-coded, this **method model** should be explicit and directly examinable. There are several advantages in representing the method directly. A hard-coded method model could give rise to sensible advice, but it would necessarily be less flexible and context-sensitive, and it would be difficult to provide any justification of the advice beyond displaying canned text messages. As most methods do not exist in canonical or standardized versions, but instead have varying house and individual styles, it is important that the method model be accessible for modification without requiring re-coding. Not only do methods exist in different versions, but even in a single organization they may change over time as a result of experience or the demands of new applications. Finally, constructing an explicit method model is an essential part of engineering any method support tool. It is seldom the case that a method is sufficiently fully documented to permit implementation without recourse to a method expert. A method model encourages an incremental development approach, and furthermore, brings to light ambiguities and gaps in the method that may have previously been ignored.

Any method model involves a *normative component*, and a *descriptive component*. The normative component contains rules about what to do in different situations and the descriptive component encodes knowledge about the concepts underlying the method.

1.2 Generating Advice

If the generation of guidance is to be seen by the user as a central part of the tool's behaviour with no external difference between the guidance and analysis components of the tool, there must be some mechanism for the guidance component to inspect the checks performed by the analysis component. A well-designed method associates a small set of representations with each step. For example, CORE produces a single diagram in a step. A natural means of linking the analysis and guidance components, therefore, would appear to be at that level, the analysis component attaching notes to the diagrams that may need revision and the guidance component inspecting the position, and in some cases the content, of such notes.

Using the normative model together with the notes produced during analysis, the tool should be able to explain the current state of the requirements analysis and what actions are required to complete it. In short, active guidance should answer the questions :

- "How am I doing ? " and
- "What should I do next ?

Finally, if active guidance is to prove useful in requirements analysis, it must be possible to experiment with different advice giving principles. Useful active guidance must therefore be under the control of easily changed advice-giving heuristics. In most cases these will be method-specific.

1.3 Analysis Status

The above system would be able, using the rules of the method model, to deduce the state of the user's analysis and from this, again using the method model, generate advice about what needs to be done. We have also considered how to represent the state of the analysis in a graphical form, without any explicit guidance to the user about what to do next, with the aim that it should be 'obvious' from the diagram what task should be considered next.

2 Current Status

An active guidance system for CORE has been implemented in Prolog within the CORE Analyst. This is tightly coupled to the Analyst so the user sees the combination as one system. This section describes the architecture of the guidance system, in terms of an Entity, Relationship, Attribute model. The attributes of the entities are not described but are shown in later sections. This is followed by a description of various models which may be used to describe methods and a particular model is then described in the following section. The remaining sections within this chapter then describe the implementation of the active guidance system and show how the information is presented to the user.

2.1 An ERA Model for Active Guidance

This section presents an ERA Model for the active guidance system. The ERA Model is presented in Figure 2.0. This will be used as the basis for the discussion of active guidance.

ERA Model for Active Guidance

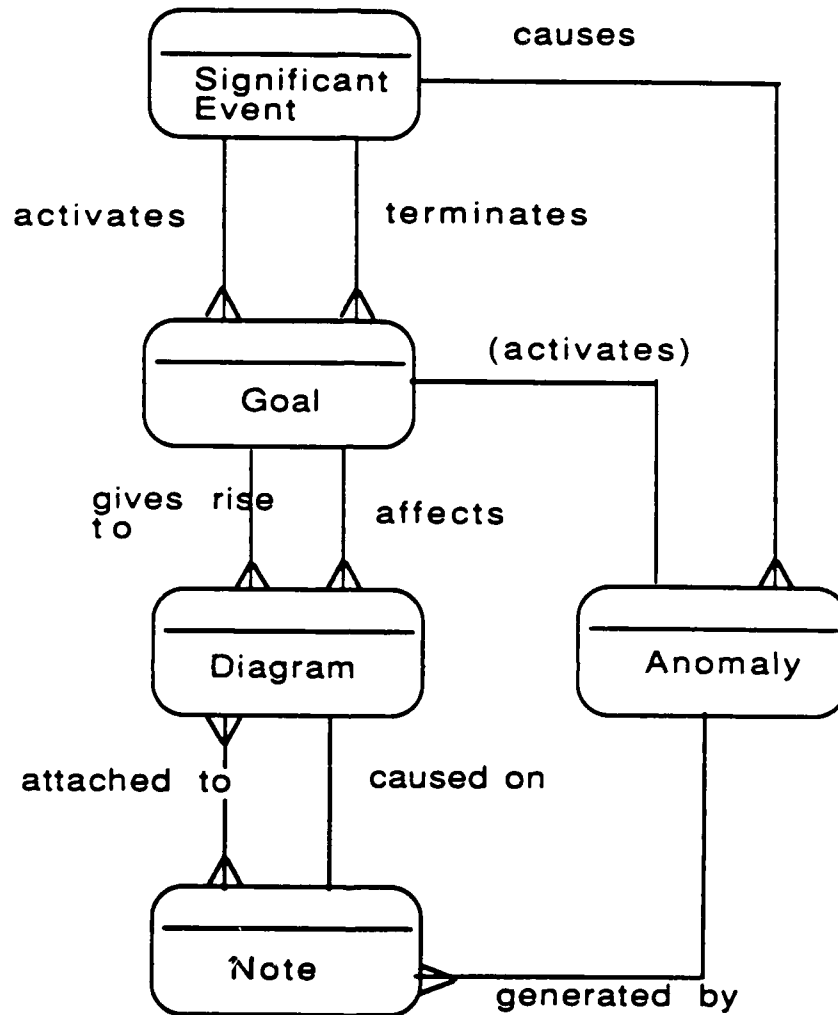


Figure 2.0

Entity Descriptions

Significant Event

A significant event is an action or sequence of actions by the user which invokes the method rules within the Analyst System. For example, the user actions of selecting an area of text, modifying the text and deselecting the area. The significant events are :-

- create an object

- delete an object
- update text associated with an object
- change the type of an object
- close an object . This only applies to diagram type objects.
- check an object.

Note that

- (i) A Significant Event **causes** Anomalies to be detected when the event would result in an inconsistent state of the specification.
- (ii) A Significant Event **activates** a Goal when the event results in the specification satisfying the pre-conditions of the Goal.
- (iii) A Significant Event **terminates** a Goal when the event results in the specification satisfying the completion conditions for the Goal.

Goal

A Goal is a desired state of the specification. We have considered two *styles* of goal - goals in which the desired state is to correct an anomaly, e.g. update all of the diagrams which are affected by the new dataflow I have discovered and goals where the desired state is the completion and validation of one or more diagrams e.g. the completion of all the Tabular Collections at a given level in the Viewpoint Hierarchy. These different styles are reflected in two of the relationships associated with Goals.

Goals are created (activated) by events which transform the specification database into a certain state. The characteristics of this state we shall term *pre-conditions*. A Goal is deleted (terminated) again when an event causes the database to achieve a certain state. The characteristics of this state we shall term *completion-conditions*. The aspect of automatic deletion of goals given these completion-conditions has not been addressed in the project. It should be noted that a particular event can cause both the posting and termination of multiple active goals.

A Goal **affects** a Diagram if the completion-condition of the Goal includes a desired state for the Diagram e.g. validation to a particular level.

Diagram

Any Diagram in the specification.

Note

A note generated by the Active Guidance system.

Anomaly

Some anomaly detected by the system as a result of a user action.

2.2 Method Model

Several approaches to representing the normative model have been investigated. The simplest approach is to model CORE as a context-free grammar, the alphabet of which denotes the types of method events or actions performed by the practitioner. These events are described above.

Figure 2.1 depicts the context-free skeleton of CORE as a structure diagram [Jackson 1975], in which the leaf nodes represent the major types of method event in CORE. A prototype using such a model was constructed as part of a student project by Nigel Kerr. Unfortunately, the rules of CORE are too context-dependent for this approach to support more than the simplest form of guidance.

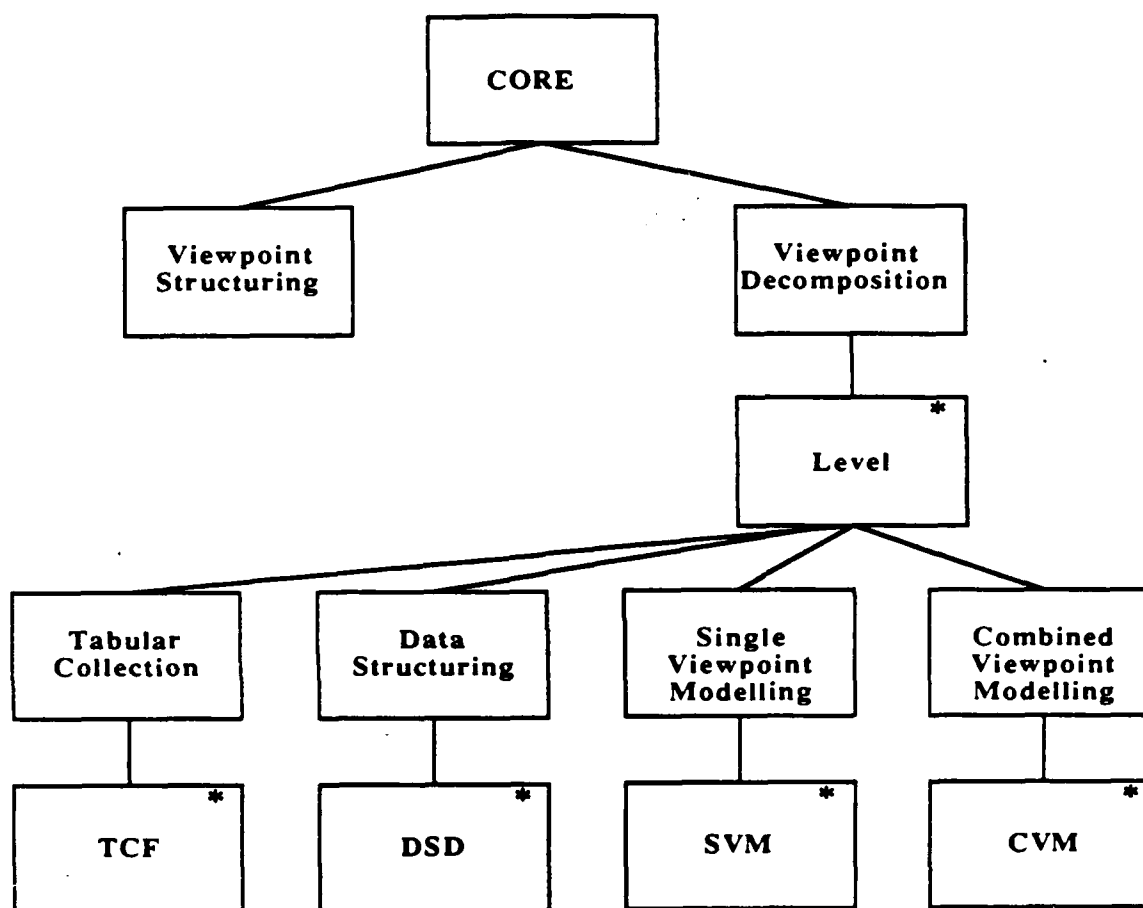


Figure 2.1

DSD depicting the steps of CORE

A second approach was to develop a formal definition of a large body of CORE in the modal action logic of Maibaum, Khosla and Jeremaes [1986]. This appears a promising basis for continued work and is described separately in the next section.

In the prototype, a more pragmatic approach has been adopted: the normative model is encoded as a set of Prolog clauses. This includes the context-free grammar rules depicted in Figure 2.1, integrity constraints such as those defining the binding of parameters, and the termination conditions for iterative steps. At present they have been written in an optimized CORE-specific form. For example, we have a rule defining under what conditions it is permissible to create a tabular collection diagram for a particular viewpoint. In English we would say *Tabular collection for a direct viewpoint at the current level is allowed, provided it has not already been done, after*

viewpoint structuring.' This would be expressed in Prolog as:

```
is-permissible-for ('tabular collection' _viewpoint) if
    'post viewpoint structuring' &
    direct-viewpoint (_viewpoint) &
    the-current-level-is (_level) &
    is-at-the-level (_viewpoint _level) &
    not ('post tabular collection'(_viewpoint)).
```

Descriptive rules and definitions have also been encoded in Prolog from higher level descriptions. CORE actions are defined solely in terms of their consequences given different preconditions - cf. the situation calculus [McCarthy 1969]. Static concept definitions also take the form of rules, which ultimately are defined in terms of specification-level predicates. A specification is held by the Analyst in a disk-based database that stores each diagram as a collection of objects and relations between them. The objects are, in general, user-selectable diagrammatic elements. As an example of the relationship between the descriptive component of the method model and the specification database, consider the useful concept 'analyzed-level' which is true of a level of the viewpoint hierarchy when data structuring has been completed for *all* the viewpoints at the level and tabular collection, isolated viewpoint modelling have been completed for all the *direct* viewpoints at the level. This is defined in rules in terms of the existence in the specification of diagrams corresponding to each of the analyses of each viewpoint.

```
analyzed-level (_level) if
    (forall is-at-the-level (_viewpoint _level)
    then
        done-data-structuring (_viewpoint)) &
    (forall is-at-the-level (_viewpoint _level) &
        direct-viewpoint (_viewpoint)
    then
        done-tabular-collection (_viewpoint) &
        done-isolated-viewpoint-modelling (_viewpoint)).
```

where:

```
done-tabular-collection (_viewpoint) if
    diagram-details(_viewpoint 'tabular collection' _id) &
    completed (_id).
```

In this example the "diagram-details" and "completed" clauses are true if the facts exist in the specification database.

The method model can be used in conjunction with the specification to generate a set of acceptable next steps at the diagram level. That is, it is known whether a diagram level step has occurred (referred to as a *method event*) depending on the presence of a complete version of the diagram it is known to produce. Recommendations beneath the level of diagrams are mediated by notes attached to diagrams (see below). At present, the only information used by the active guidance system is that a note exists. No automatic analysis of note contents occurs.

2.3 Formal Description of CORE

We have chosen a formal approach based on the modal action logic (MAL) of Maibaum, Khosla and Jeremaes (1986). This was developed originally for specifying database management semantics (for example, the preservation of integrity constraints over transactions), and is being used with some temporal extensions in the formal specification of real-time systems [Finkelstein

and Potts, 1986]. Because of our practical motivation any theoretical issues concerning the generalized decidability and tractability of MAL can happily be ignored. It is our contention that it is preferable to possess a detailed formal model of CORE, the precise semantics of which can be weakened by principled implementation decisions, than to start from an ad hoc model, the meaning of which is embedded in its implementation.

MAL is a non-classical formal system in which the notions of actions and agency are central constructs. It treats separately the definition of states, the definition of actions, action composition and the conditions under which actions can or must occur. It is thus a powerful and intuitively expressive way of formally describing a method such as CORE.

This is not the place for a full exposition of MAL, for which the reader is directed to Maibaum et. al. [1986]. We shall skate over the formal semantics and instead appeal to the readers intuitions about the notions of state, agency, permission, and so forth. The notation is explained as it is introduced.

MAL is based on a layered approach. At the heart is a standard many-sorted (i.e. typed) first-order predicate logic. Formulae constructed using only this layer can be used to assert facts and rules about states. For example, to say that Tabular Collection is complete at a given level of the Viewpoint Hierarchy if Tabular Collection has been performed on all Direct Viewpoints at that level, one would say (inventing the appropriate vocabulary):

```
(~exists v)(  ~indirect(v) &
              ~post-tabular-collection(v) ->
              post-tc-level(level-of(v))).
```

Here, 'v' is assumed to have been declared to be a variable of type 'viewpoint', so the quantification is already restricted over viewpoints. The symbol '~' represents negation and '->' implication.

The first non-classical extension is introduced by the notion of *actions*. These are state-changing (i.e. modal) transformations. The performance of actions is represented by enclosing the action name in brackets. The formula to the right of the action describes its post-condition. For example, to say that after Viewpoint Structuring there is a unique Viewpoint Hierarchy (vh) and that the initial Viewpoint Hierarchy Level is the first, one would specify:

```
(exists! vh)(  [viewpoint-structuring]
               post-viewpoint-structuring &
               vpt-hierarchy = vh &
               curr-level = 1)
```

Actions are performed by 'agents'. In the CORE normative model, there are several agent types such as Analyst, Client authority and Viewpoint authority. For completeness, one should always indicate which agent is performing the action, so the term [viewpoint-structuring] should be replaced in the above example by [analyst, viewpoint-structuring], but we shall abbreviate the notation where it is obvious that the Analyst is the agent.

The last axiom does not, of course say anything about *when* Viewpoint Structuring should happen: it just specifies its consequences when it does. To describe the conditions under which actions can occur, or their causes, one used a further extension to the logic: the deontic operators *PER* (permitted to) and *OBL* (obliged to). In CORE, for example, Tabular Collection for a Direct Viewpoint on the current level of the viewpoint hierarchy is permissible (provided that it has not already been done) after Viewpoint Structuring. This would be formalized as:

```
(post-viewpoint-structuring &
```

~post-tabular-collection(v) &
~indirect(v) &
level-of(v) = curr-level) ->
PER(tabular-collection).

When an action is obligatory and several others are permitted, the obligation overrides the permissions until it is discharged. The question of conflict resolution among obligatory actions does not arise as two conflicting obligations constitutes a logical inconsistency and should never happen in a sound model. Indicating preferences among permitted actions. For now we shall pretend that all permitted actions are equally sensible.

Appendix A contains a portion of the MAL specification of CORE. Given the preceding discussion, the notation should be self-explanatory.

The great benefit of a formally based normative model is that the same formalism can also be used for describing the remediation mechanisms and advice-giving heuristics.

2.4 Note Passing

When the Analyst detects an inconsistency or incompleteness in the specification it does so as the result of a check on the current diagram. The set of checks that are performed is detailed in Appendix A of the latest User Guide for the CORE Analyst, issued by Systems Designers Plc. The error may have resulted from an error in the current diagram or earlier related diagrams. If the user decides that the current diagram needs revision it can be changed there and then. If, however, the other diagrams need changing or the current diagram needs changing but the user decides to defer the revision until later, some note must be left attached to the diagrams in question explaining the kind of change required and why it is necessary. This facility is under the control of the user in the sense that the Active Guidance may be switched off. In this case the user is warned when anomalies are detected but if they decide to continue with the change no attempt will be made to assess the impact of the anomaly.

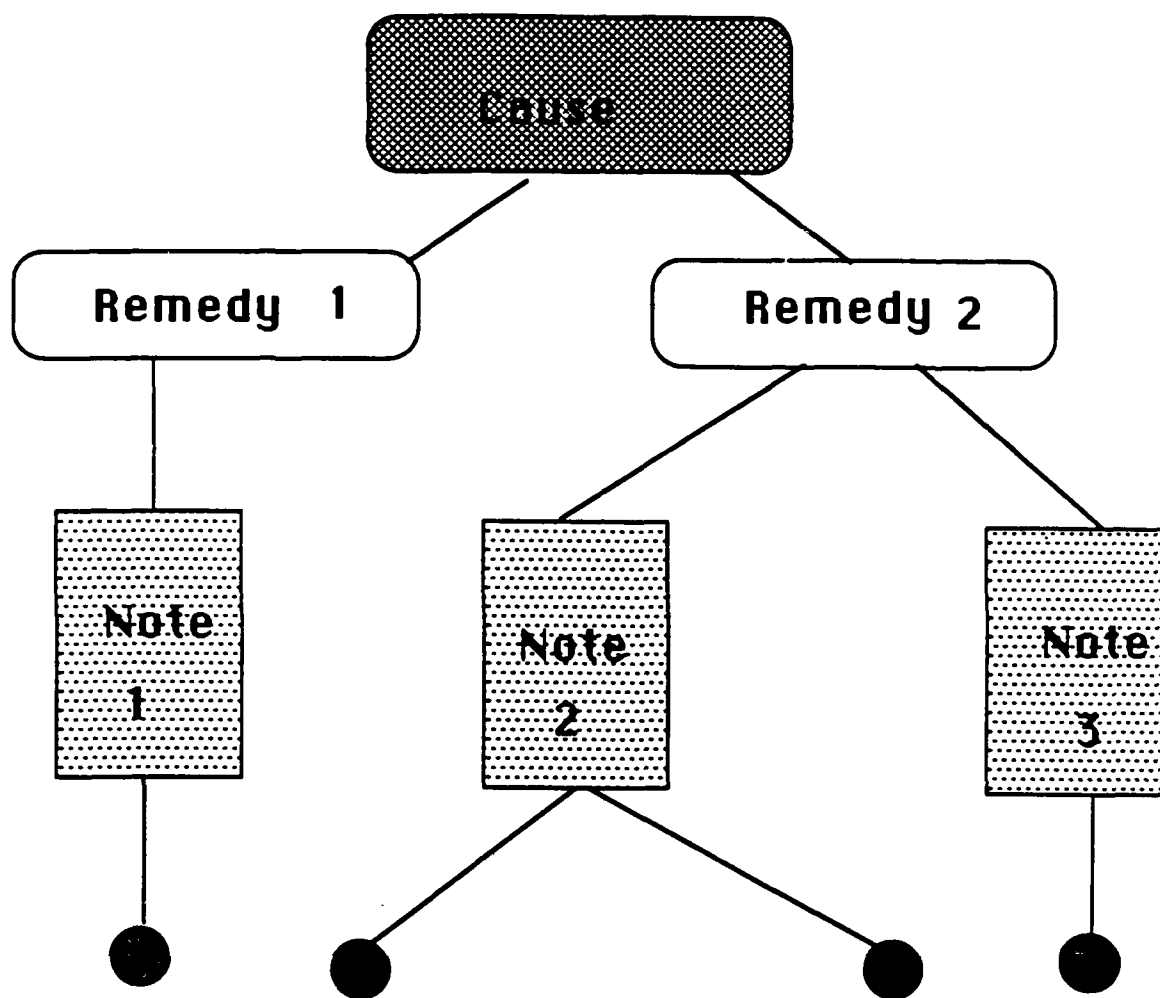
Often only one note is required although its contents may apply to more than one diagram, thus a *note-reference* is attached to each of these diagrams which indicates the common note.

During the generation of notes, the Analyst associates both a *cause* and a possible *remedy* with the note. A *cause* is defined as a *generic* type of anomaly together with a *specific* object-type and name. A *remedy* is a generic action that may be applied to the named object in order to solve the problem. Several generic kinds of cause and remedial action have been identified and are presented in table 2.1.

Anomaly		Remedy	
Duplicate	An object has been defined twice within the same diagram or duplicate diagrams exist within the same project.	Rename	- one of the objects
		Abandon	- one of the diagrams
Syntax	A syntax error has occurred	Amend	- the object in order to comply with recognised syntax
Too many	One feature of the diagram is becoming too complex e.g. too many data flows on one action.	Simplify	- reduce the complexity
		Decompose	- split the object into components
Inconsistent	An object on diagram "d1" is missing from a corresponding diagram "d2".	Add	- add the object from "d1" to diagram "d2"
		Delete	- the offending object on diagram "d1"
		Rename	- the offending object on diagram "d1" or an existing object on diagram "d2"
Illegal Decomp.	A decomposition action is being performed on an object which cannot be decomposed	Abandon	- give up on this action
Premature	A stage of the method is being performed on a diagram before completion of steps at a previous level	Abandon	- go back to a previous step

Table 2.1 Generic Anomalies and Remediation Mechanisms

Each of the remedies associated with a single cause are mutually exclusive, that is compliance with one problem-solution will render others redundant. However more than one note may be connected to a single remedy and each of these must be acted upon in order to correct the original anomaly. Figure 2.5 depicts the relation of causes and remedies to notes and note-references.



note - references

Figure 2.5

In the prototype, the presence of notes is detected and used to suggest future actions to the practitioner. Several kinds of notes can be attached to diagrams, the types reflecting how they were put there. One type, the 'active guidance note', may necessitate some modification to the diagram it refers to. The active guidance system uses this information to update its model of the completion status of the current specification. Such a diagram cannot be complete, even if the user previously claimed it to be, and any diagram based upon it may also need subsequent revisions. The method model can be used to identify such candidates for future revision. It is not always possible for the Active Guidance System to deduce which diagram needs revision. In particular, it is always the case that the user has an option of modifying the diagram on which the anomaly was detected or the affected diagrams. In addition many anomalies will necessitate a change to one of a set of diagrams. Thus diagrams do not totally lose the completion status they had acquired before the anomaly was detected and if no action is done on the diagram and the notes are deleted the diagram reverts to its previous state.

Consider the following example. The practitioner has completed the viewpoint structuring stage and is constructing the tabular collections for the first level viewpoints. During discussions with the user about a particular tabular collection it is discovered that a new viewpoint is needed in the viewpoint hierarchy and as a destination for data flows. The analysis component of the tool detects this anomaly and asks the practitioner to either **Cancel** the new name or to deduce the remedial notes using the **Ignore** option as in Figure 2.2. When the practitioner chooses Ignore, notes are attached to the diagrams which are potentially affected by this anomaly. In particular a note is attached to the viewpoint structuring diagram. In Figure 2.3 we see what happens when the practitioner opens the viewpoint structuring diagram and asks for any notes attached. A list of all of the abstracts is presented, the abstract in this case being "New Viewpoint Introduced". Figure 2.4 shows the result of opening the note.

The underlying cause-remedy-note structure is shown in Figure 2.6 with the active notes highlighted.

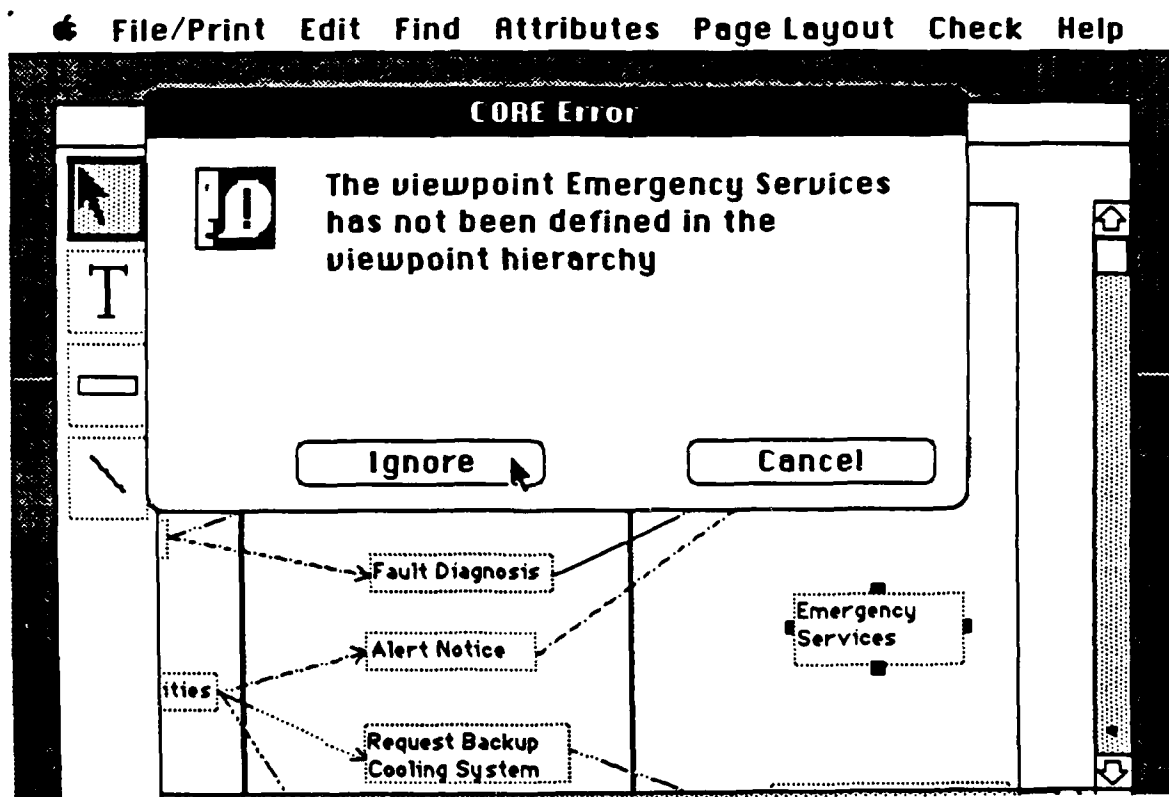


Figure 2.2

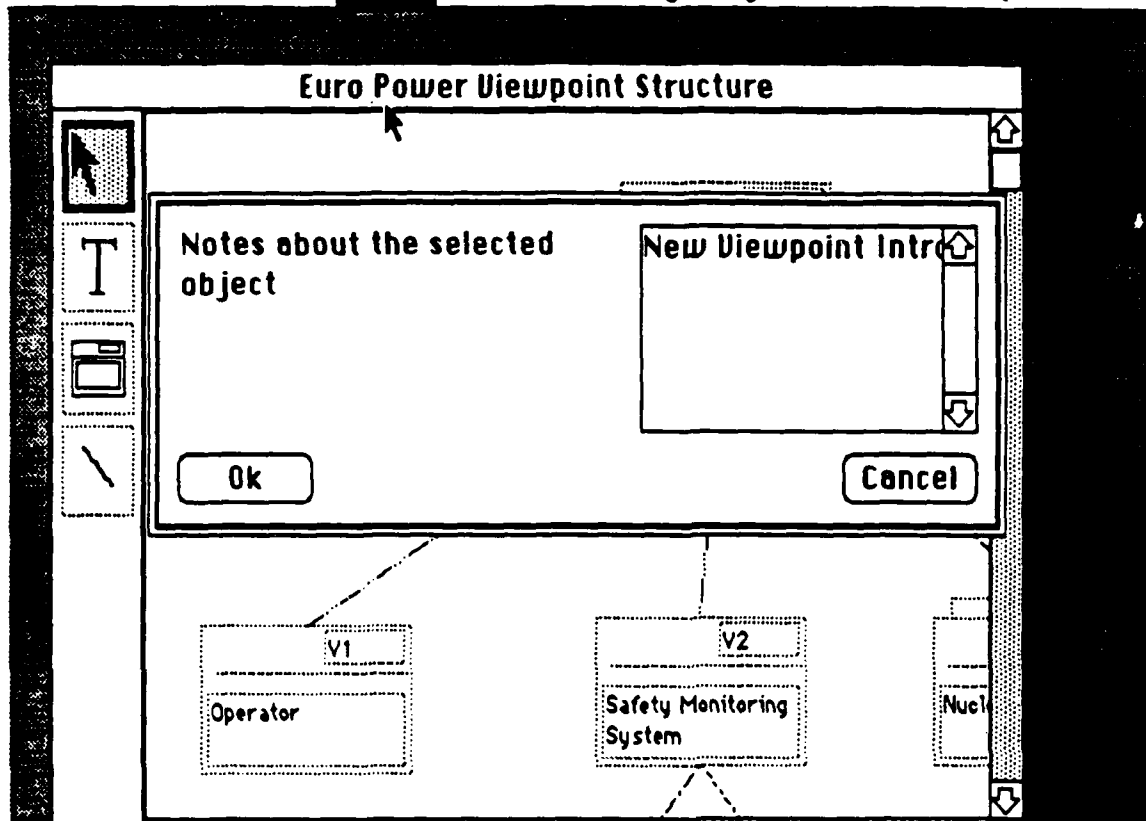


Figure 2.3

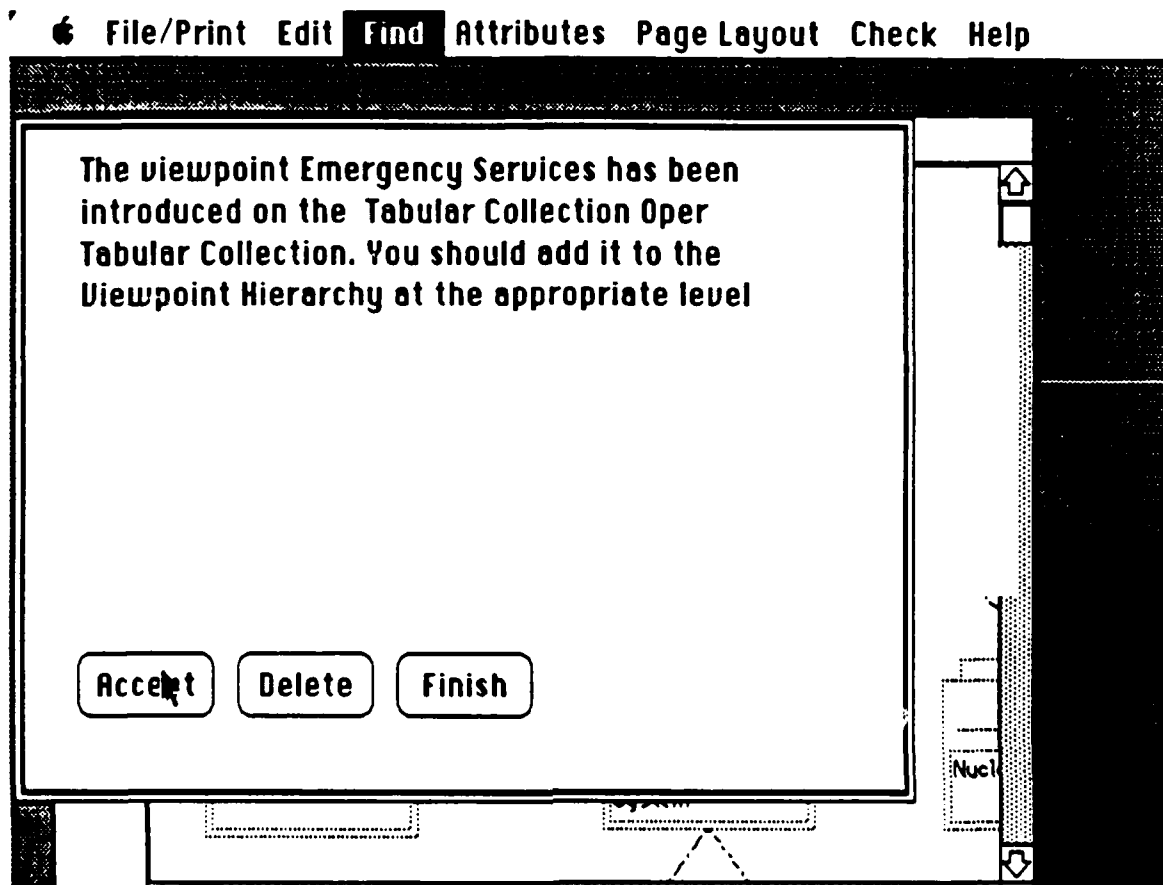
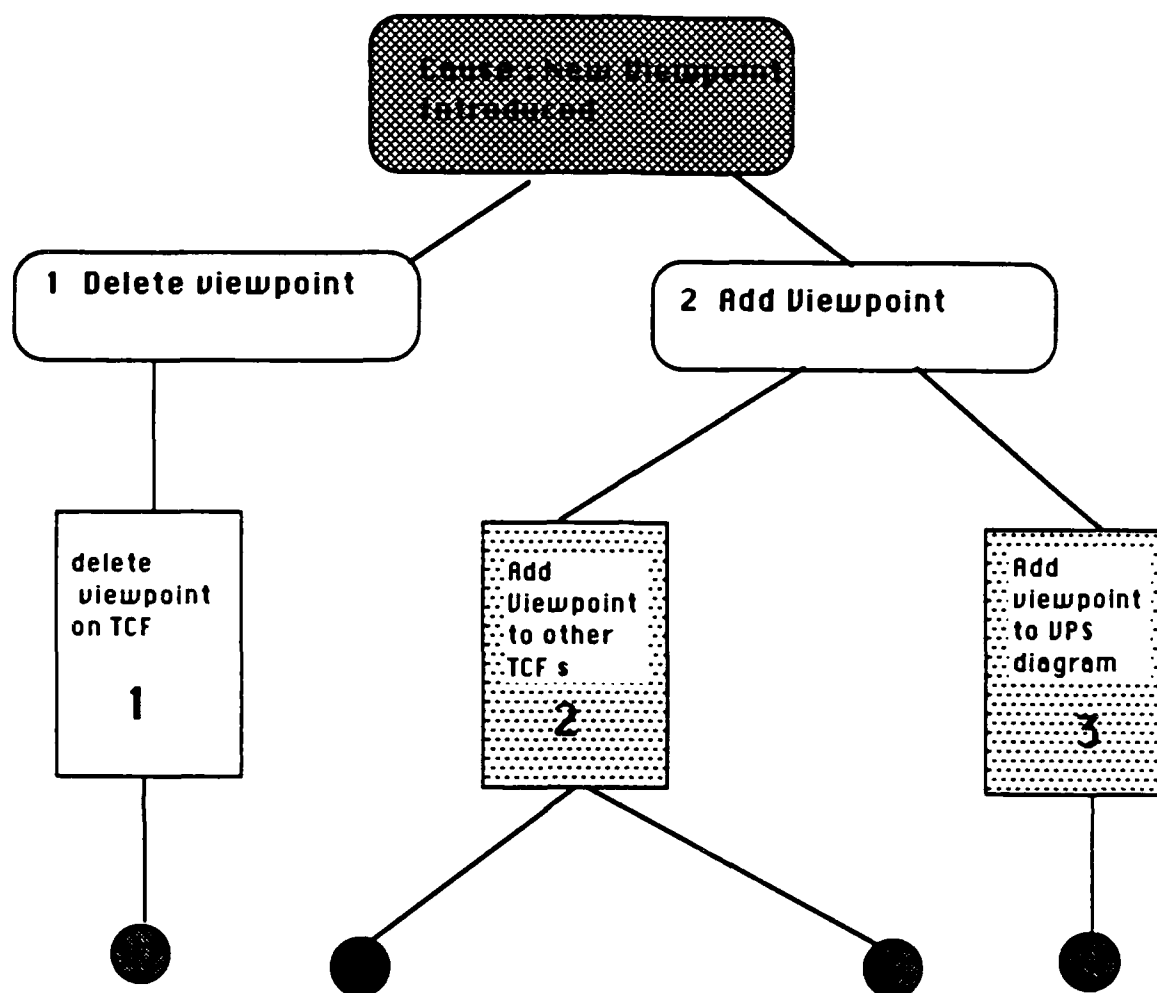


Figure 2.4



note - references

Figure 2.6

2.5 Remediation Strategies

Remediation is necessary whenever the current specification exhibits a method anomaly. Because of the nature of requirements analysis methods and practitioners' preferred ways of working, not all anomalies should be regarded as outright 'errors', although some undoubtedly are. Several general classes of anomaly have been identified (see Table 2.1). These have been kept as generic as possible so that they could apply equally to methods other than CORE, although the rules for detecting an anomaly in a specification database are, of course, method-specific. Among the general classes of anomalies are **missing precursor**, where a diagram has been created before one on which it depends. In CORE, a single viewpoint model depends upon a corresponding tabular collection form and data structure diagram. If either of these is missing in the presence of the single viewpoint model, then it is missing a precursor. Another is the **premature analysis** where a step has been performed before it should be, even though all its precursors exist. An

example of this in CORE is where analysis is started at a level of the viewpoint hierarchy, when analysis at the previous level is still incomplete.

It is not possible to anticipate all possible anomalies and devise specific remediation strategies for them. General mechanisms are quite feasible, however. For example, the remediation strategy for coping with a missing precursor is to recommend the creation of a precursor, followed by re-analysis of all its dependents. The remediation strategy for coping with premature analysis is to give higher *priority* to all actions still necessary at the previous level than to those now possible at the lower level.

2.6 Advice-Giving Heuristics

Usually, in any situation the practitioner *could* perform a large number of actions. Some follow from the normative component of the method model. In a perfect CORE project only these actions ever need be performed. Others are **remedial actions** to correct inconsistencies and incompletenesses that have arisen as the specification evolves. Finally, there are simple clerical **actions**, such as the completion of diagrams that exist but which are known not to be complete yet, or the analysis of diagrams that have not been analyzed since a prior change or the receipt of a note. Given the range of possible courses of action, the active guidance system must restrict its advice by filtering the candidate actions through a set of heuristics.

Active Guidance is generated in three stages :

- (i) Collection of all plausible actions generated by the normative model and the notes generated by CORE Analyst according to the remedial model.
- (ii) Ordering and possibly filtering these actions by prioritisation heuristics.
- (iii) Presentation of advice, status information and explanations to the tool-user on demand.

The aggregation of actions suggested by the normative model is achieved by simply invoking PROLOG rules of the form described in Section 2.2 of this chapter. Remedial actions are derived from notes that have been generated by the CORE Analyst during a diagram check.

2.7 Advice Attributes

In order to derive suitable prioritisation heuristics several attributes have been identified on which to judge each piece of advice and some of these are described below.

Type	- source of the advice, whether normative or derived from notes. In the case of the latter, four further categories are required corresponding to the four types of note generated by the Analyst, that is, error , warning , guidance and active guidance .
Action	- the action proposed by the advice. In the normative model suggested actions are to start or finish a particular diagram. Remedial actions are those listed in table x.y.
Object	- the diagram or object <i>type</i> to which the advice applies. For example, normative advice refers to the diagram types used in CORE (Viewpoint Structuring , Tabular Collection and so on) while remedial advice may also pertain to objects within diagrams such as actions and data .

diagram or viewpoint.

- Level** - broadly equivalent to the level of abstraction of the advice. In CORE this may be related to level in the viewpoint hierarchy at which the advice applies.
- Dependency** - whether or not advice already exists pertaining to earlier related diagrams which may affect other alternative actions .
- History** - which types of advice have been folloed by the user on previous occasions.
- Effort** - the amount of effort required to comply with the advice.

The last three heuristics have been considered but not implemented during this project.

2.8 Priority Factors

A priority is assigned to each piece of advice to be presented to the User of CORE Analyst and which is designed to indicate its importance. This *final* rating is derived from a *combination* of factors which are associated with the different attributes of the advice. For example, consider the *action* that is associated with each piece of advice. For the normative model these may be one of "start" or "finish". If it is desired to stress the importance of completing a diagram before starting a new one then that action is given a higher priority than the other.

Priorities may be associated directly with attribute values (as in the example above) or calculated by means of a formula. The priority factor assigned to the level attribute of advice (the advice-level) is calculated according to the following equation :

$$\text{Factor} = \frac{(\text{current-level}) - (\text{advice-level})}{\text{MAXIMUM}(|\text{current-level}| |\text{advice-level}|)}$$

The current-level is the layer in the viewpoint hierarchy at which the analysis is currently concentrated. It may be observed that the sign of the calculated factor will be negative for levels below the current one and positive for higher levels. The magnitude of the factor increases with increasing 'distance' from the current level. The behaviour of this function is shown graphically in figure f.g. in general terms, it is recommended that higher levels are completed before moving onto lower ones.

It is a fairly straight-forward task to assign appropriate priority factors for the different values of a given advice-attribute. For example, consider the *type* attribute - a suitable order of priorities might be :

- | | | |
|----|-----------------|-------------|
| 1) | error | |
| 2) | warning | |
| | active guidance |) equal |
| 3) | normative |) weighting |
| 4) | guideline | |

Similarly completing diagrams may be a more advisable action than starting new ones ; higher levels must be completed before lower ones (see above) and so on.

What is more difficult to assess is the relative importance of the different attributes of the advice, for example, is the advice model-type of more significance than the level of the diagram to which it refers ? The answers to questions such as these are only likely to be discovered after a process of trial and error, to facilitate this process, an additional scaling factor associated with each attribute is introduced into the calculation

process of trial and error, to facilitate this process, an additional scaling factor associated with each attribute is introduced into the calculation

By amending these scaling factors it is possible to experiment with the relative importance of different 'kinds' of advice. For each piece of advice that is found, the prioritising algorithm must first find the base priority factor associated with a given attribute. Each of these factors must then be multiplied by the scaling factor associated with the appropriate advice-attribute. Finally, the resulting number is combined with the corresponding values for other attributes of the advice to obtain a final weighting.

For simplicity of implementation, the range and combination of priorities that are employed by EMYCIN are adopted. Factors range from -1.0 (strongly discourage) to 1.0 (strongly recommend). Two factors F1 and F2 may be combined to yield an overall factor F using the following formulae:

$$\begin{aligned} F &= F1 + F2 - F1 * F2 && \text{if } F1, F2 > 0 \\ F &= F1 + F2 + F1 * F2 && \text{if } F1, F2 < 0 \\ F &= \frac{F1 + F2}{1 - \text{MIN}(|F1|, |F2|)} && \text{otherwise} \end{aligned}$$

2.9 Presentation of Advice

Advice is generated on user-demand and results in the Analyst exercising the normative model and collecting notes associated with previous diagram checks. A simple analysis of the normative advice reveals the stage and level which the user has reached, while a count of the number of notes attached to the requirements analysis gives a rough indication of its correctness (although this is naturally dependent upon the amount of checking that has been performed). The dialog shown below should thus be able to answer the first question posed in section 1.2 ... "How am I doing?"

You appear to be at the TCF stage on level 1 .
There are 6 notes outstanding .

View Advice By :-

Rank

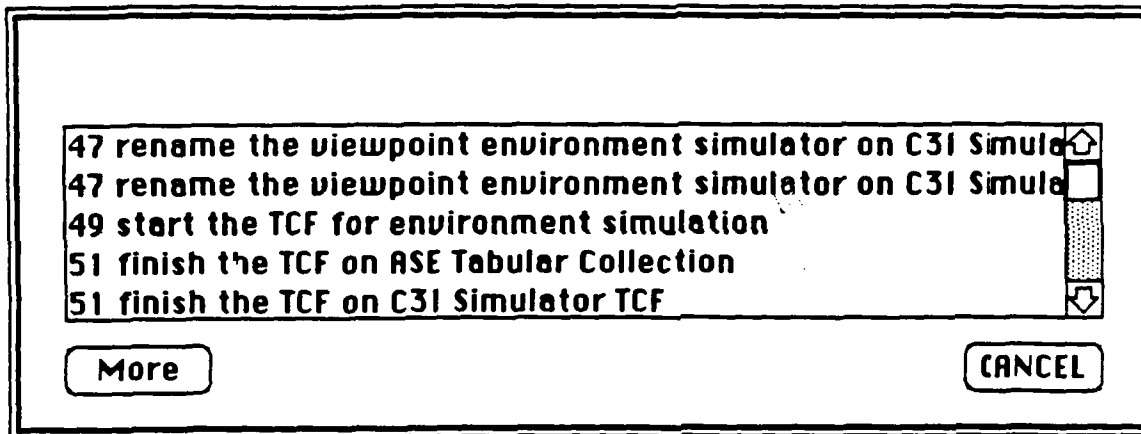
Cause

Viewpoint

Diagram

CANCEL

The user is now able to view the advice individually and ranked in order of preference (priority). Thus the dialog shown below answers the second of the questions of section 1.2 ... "what should I do next?"



47 rename the viewpoint environment simulator on C3I Simula

47 rename the viewpoint environment simulator on C3I Simula

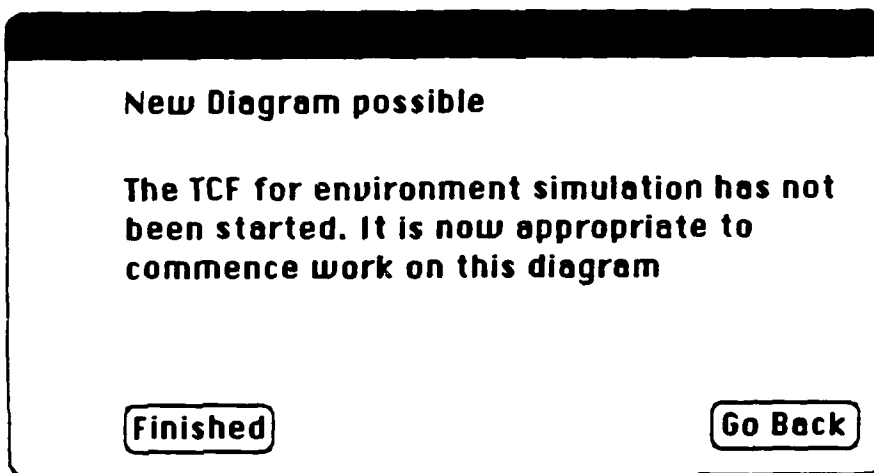
49 start the TCF for environment simulation

51 finish the TCF on ASE Tabular Collection

51 finish the TCF on C3I Simulator TCF

More CANCEL

The number associated with the individual items of advice shows the weighting given to the advice, after normalisation, with preferred advice being given lower weightings. For each piece of advice shown, further explanation may be provided in the form of context sensitive semi-canned text for normative advice such as -



New Diagram possible

The TCF for environment simulation has not been started. It is now appropriate to commence work on this diagram

Finished Go Back

In the case of remedial advice the note is re-displayed, such as -

The data base response to t & c is not generated by the viewpoint ase element

Finished

Go Back

2.10 The Status of the Specification

The user may request that a representation of the current state of the analysis is shown in graphical form. The major aim of this diagram is to provide a representation which makes it "obvious" to the user what should be done next.

This section describes this facility and illustrates the interface provided to the user.

Most methods for analysis and design have, at an early stage, some way of *decomposing* (or partitioning) the problem to be addressed into manageable areas. In CORE, as discussed in Section 1.3, the problem is partitioned into a tree of viewpoints. The analysis then proceeds by collecting information about the viewpoints and examining the data interconnections between the viewpoints. We thus feel that it is natural to view the specification in CORE as consisting of a set of trees, each tree representing a stage of the analysis and each node on the tree showing the status of the diagram associated with the stage and the viewpoint. The nodes are represented as icons with the name of the viewpoint underneath the icon. The particular icon indicates the status of the diagram. The status of the diagram is shown as one of the following -

The status of each diagram is either :-

- 1) *Not needed*. For this particular stage, the diagram associated with this viewpoint is not needed. For example, Single Viewpoint Model Diagrams are not necessary for indirect viewpoints. This is represented by the following icon -



- 2) *Needed but not started*. The diagram is needed for this stage but has not yet been started. This is represented by the following icon -



- 3) *Started but not yet completed.* The diagram has been started but not yet completed. This is represented by the following icons -

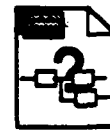


Tabular Collections

Data Structuring

Single Viewpoint Models

- 4) *Started and Notes Attached.* The diagram has been started and notes have been attached to it by the Active Guidance system as a result of anomalies detected on other diagrams. If the diagram previously had the status completed, this is temporarily withdrawn until all the notes generated by the Active Guidance system have been deleted. This is represented by the following icons -

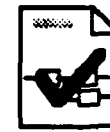


Tabular Collections

Data Structuring

Single Viewpoint Models

- 5) *Diagram Completed.* This is represented by the following icons -



Tabular Collections

Data Structuring

Single Viewpoint Models

An example of a Summary Diagram, summarising the Tabular Collection Stage is shown on Figure 2.7

Tabular Collection Summary

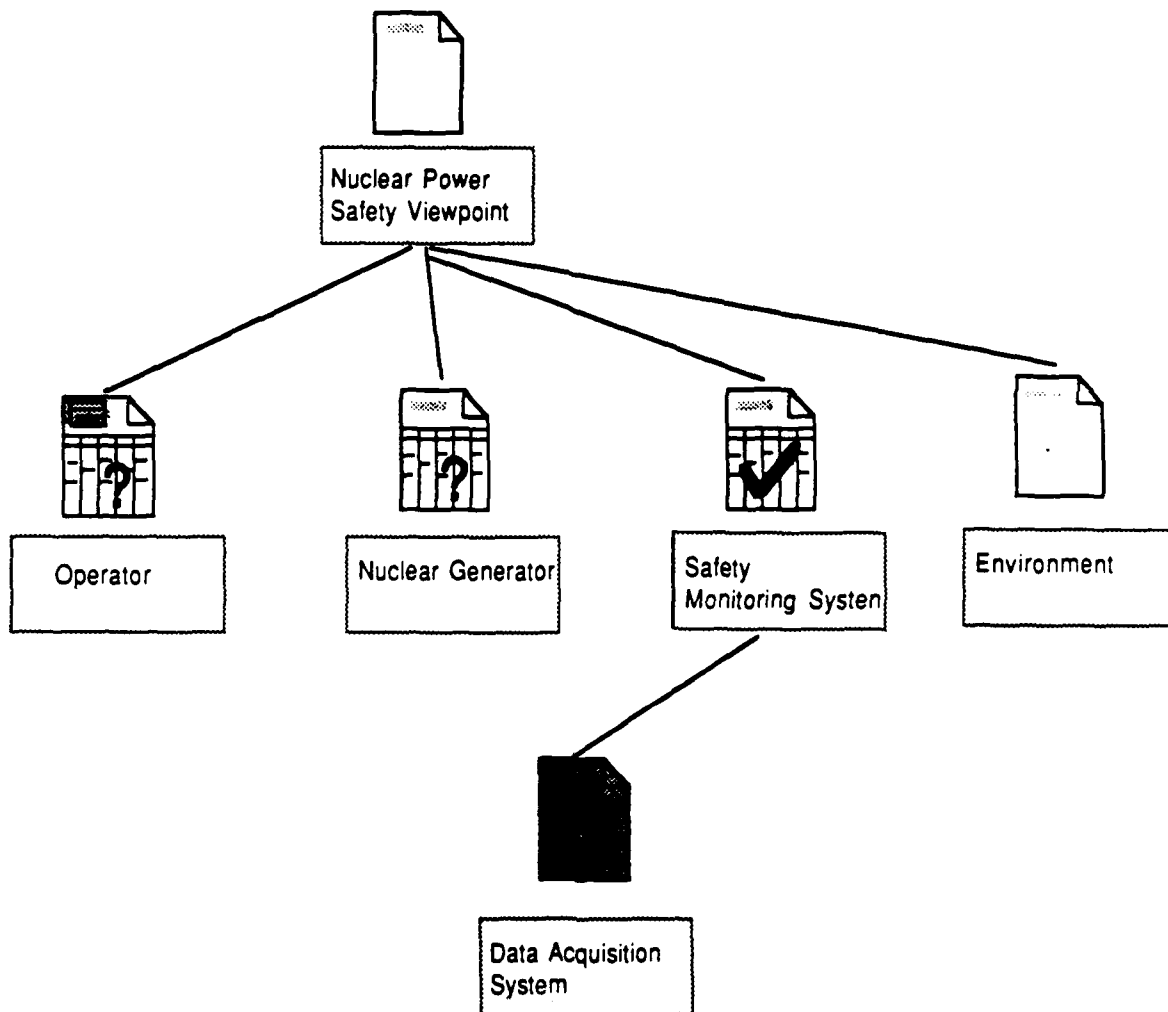


Figure 2.7

The user may view the Summary Diagrams as an extension of the file management facilities made available by the operating system. The icons represent documents and may be manipulated as though the windows were open and selected. Particular facilities implemented during the research for manipulating these icons are -

- The ability to view notes attached to the represented documents.
- The ability to find the descriptions of these diagrams and a summary of the last check produced by the system on these diagrams.
- The ability to check the diagrams using the standard Analyst checking rules.

The Summary Diagrams thus provide the user with an alternative way of viewing the diagrams composing a specification, augmenting the information provided by the operating system and offering application dependent options.

3 Conclusions

Although much of the prototype is CORE-specific, our approach to active guidance could be adapted to any requirements analysis or early design method. The conditions are that the method is truly a method, comprising a grammar of steps and principles for applying them, rather than just a collection of notations. Our approach to active guidance has suggested several avenues for future research. These are discussed in the final chapter.

Chapter 3

ANIMATION OF SPECIFICATIONS

1 The Approach

1.1 Transactions

Many requirements analysis methods involve, at some stage, the identification of actions and data flows within the proposed system (eg. SADT [Ross 1977] and PSL/PSA [Teichroew 1977]). Although all these actions will be related in some way, there will typically be smaller groups of interconnected actions which interact more closely to perform some specific sub-task of the system. We refer to such a group of actions as a transaction. In CORE a Combined Viewpoint Model (CVM) is prepared for each transaction of interest. An example CVM for a patient monitoring system in a hospital intensive care ward is shown in Figure 3.1. A patient is monitored and an alarm is triggered if the readings are outside the specified safe limits for that patient.

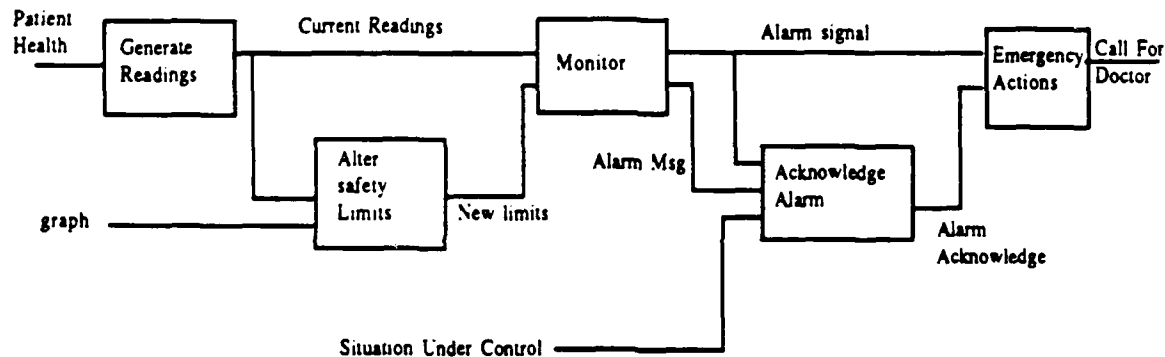
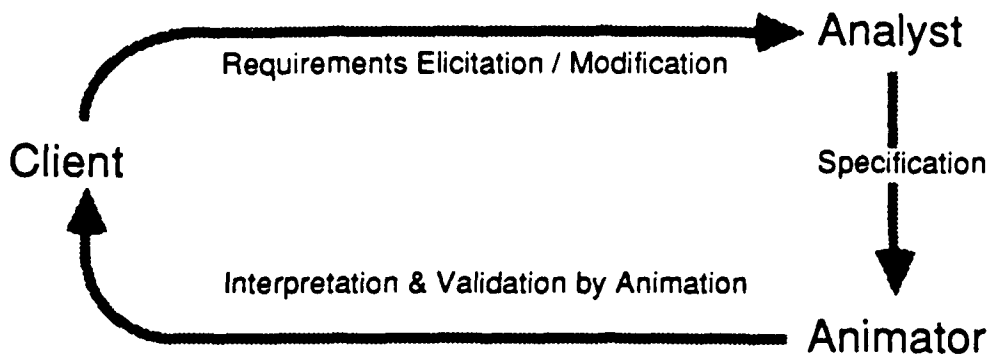


Figure 3.1

The prime use of an animator will be for the validation of transactions, particularly those which involve critical performance or reliability aspects of the proposed system. Animating a transaction is essentially playing out a scenario which may take place in the eventual system. For example, in the patient monitoring system, one important scenario that should be investigated is the situation in which a sensor fails. Playing out this scenario will help answer questions like 'will the doctor be called as a result?', 'what does the ward nurse do when this happens?', or 'will an alarm be triggered to warn someone about this?'. Consequence testing of this nature often shows up loopholes in the specification.

For this purpose, the animator should be capable of interaction with the analysis tools: in this case the Analyst. Analyst specifications can be transformed to for interpretation by the animator. Since the main purpose of the animator is to provide an interpretation of the requirements to the client for validation, the result is likely to be correction and modification of the specification i.e.



The note passing mechanism, discussed in relation to method guidance, provides a convenient means for transmitting the required modifications back into the Analyst.

1.2 Transaction selection strategies.

There are several ways in which a scenario can be animated from a static action/dataflow description. There are different strategies to our approach, which we refer to as **transaction animation**. The strategy adopted not only affects the form of interaction between the user and the animator (i.e. the user interface), but also serves a slightly different function. Consider the following three strategies :

- (1) Animation is divided into two distinct phases - the building of a transaction and the execution of the transaction. A transaction is built by single-stepping through the specification, selecting the actions of interest. When this is done, the actions are executed in turn.

This strategy is useful when the user knows before hand exactly what actions are involved in a particular scenario.

- (2) There is no separate building and executing phases - each action is executed immediately after it is selected. Hence animation involves interactively selecting and executing each of the actions of interest.

This form of animation allows the user to choose alternative decision paths based on the current state of animation. It is also very useful for browsing through a specification in the form of computer-aided walkthrough without knowing before hand which are the actions of interest.

- (3) This strategy is similar to the first, except that the transaction to be animated is generated automatically by the animator, given just the first and last actions in the transaction. This is not difficult since all actions and their connections would have been identified.

Apart from providing a very quick way of building up a transaction, this strategy is also useful for investigating the relationship, if any, between two actions. For example, to find out if rupturing a seal in a nuclear generator will ever lead to a shutdown of the reactor, the user needs only to select 'Rupture Seals' and 'Shutdown Reactor' as the first and last actions respectively. All possible paths, if any, between the two actions will be generated for animation. The problem is, however, that in general too many unwanted and 'uninteresting' action paths can be generated.

Hence, the chosen strategy was the **interactive approach** described in (2) above. In fact, support is provided for both (1) and (2), the latter permitting a transaction to be outlined and run as different scenarios by providing different data values.

An alternative to our transaction animation approach is **data-driven animation** (c.f. [Barth 1986], [Docker 1986]). In transaction animation we are, in each run, typically interested in finding out how a specific part of a system will behave in a particular situation. In data-driven animation however, we take a bird's eye view of the system and see how the values of certain data will affect the system as a whole. So if, in the former case, we are interested in how a failed sensor might lead to a doctor being called, in the latter we look at what the consequences are of a failed sensor on the entire system. To conduct such an animation, the user would first give value(s) to the appropriate data flow(s) in the system to indicate that a sensor has failed. The animator will then generate a list of all actions which have become executable as a result. (For simplicity's sake we will assume that an action is executable when all its input data has been defined, i.e. all its input data flows contain values.) From this list the user can pick out the actions to be executed. The result of executing an action is that its output data will be defined, hence making more actions eligible for animation. This way the effects of critical data rippling through the entire system can be observed.

The main problem with data-driven animation is that, in a similar way to the automated derivation of paths (3 above), the number of actions which can become eligible can be large. Also, the order of execution of actions may be difficult to understand, and requires that the user provides data values for all inputs without additional prompting. For these reasons, we have not followed the data-driven approach.

1.3 Action Definitions

It is clear that to be able to execute an action one needs to associate some form of executable code with the action. We call this the **action definition**. In the simplest form, action descriptions can be expressed as mappings from an action's input data to its output data. Clearly this type of definition is only suitable for describing the very simplest of actions, or if the user intends to use the animator only as a browsing tool and is not particularly interested in the data transformation and processing that normally takes place within an action. For a more realistic model of the system, more sophisticated forms of action descriptions are needed. One approach would be to describe actions using a conventional programming language such as Pascal or C, where an action definition is essentially a program fragment of the proposed system and can be executed by running it through an interpreter or by having it pre-compiled. Although more suitable for describing algorithmic processing of data, this approach requires a knowledge of the language used, and tends to make describing simple actions unnecessarily detailed and complicated. It is also more akin to prototyping and may lead to premature decisions on the use of language, data structures and algorithms.

For the purpose of animation, action descriptions should be kept as simple as possible, but at the same time they should be capable of expressing some form of algorithmic processing. Our compromise is to use the mapping approach as a basis for describing actions, but extend it by providing the user with some basic operators to perform data processing. For example, we can describe an action CheckReading as follows, (variables begin with an underscore, functions are in uppercase):

```

LowerBound  =  _lower
UpperBound  =  _upper
Reading     =  _x

Alarm       =  IF( (_lower < _x) AND (_x < _upper) )
               THEN off
               ELSE on.
```

Note that LowerBound, UpperBound and Reading are inputs whereas Alarm is the output of

CheckReading.

Query the User

In addition, we have found it useful to leave some action definitions "open" in that it is left to the user to make some of the decisions at animation time. In this way a given transaction can be conveniently used to generate scenarios which differ in the decisions that are taken at particular points in the transaction. The user must be prompted for the decision at animation time. Two forms of these open actions have been provided. The simplest means is for certain actions to be left undefined, and for the user to be required to "simulate" the action at animation time by providing the required outputs. This mechanism can be used as a default for actions which are too complex to define, are actually performed by a person, or are not yet well understood and defined (figure 3.2). An alternative is where *part of an action* is to be left for user decision. A simple extension to functions available provides this facility. For example:

Q (Generate new collateral intelligence?)

will generate the enclosed message at the time of action execution. The user responds by selecting *yes* (true) or *no* (false). This result can, if desired, then be used in other functions, thereby providing a powerful and versatile facility in a simple manner.

Animation defaults :

Undefined actions:	Visible actions:
☒ Query the user	☒ All actions
☐ Produce null outputs	☐ Executable actions
☐ Not displayed	

Figure 3.2

1.4 Data and Action (De)composition

An aspect which we have not yet discussed is that of data descriptions. We have thus far assumed that the data consists of only simple values, or have represented structured information, such as reports or graphs, merely by a simple value indicating its presence or some single interpretation of it (such as good, fair, or bad). We have found that this is most often the case, and that animation should not be made too complex whence it comes to resemble simulation. We therefore recommend that data values be kept simple wherever possible.

However, an investigation was conducted to extend data handling to structured data, where the definition could be obtained from data descriptions (cf. data dictionary in Structured Analysis [De Marco 1978]) provided as part of the CORE method. Functions, similar to those for action definitions, were provided to permit users to infer simple values from structured data by defining mappings from the various fields to the simple value. However, this was not adopted since it tended to duplicate the action definition, and complicate use of the animator.

In line with the principle of simple and clear concepts, we rather chose to represent structured data as lists of values. This approach is sufficiently powerful to express any data composition (as lists of lists, if necessary) and is in nicely compatible with the functional description of actions.

The notion of decomposable action descriptions was also investigated. In general data flow style, it would be useful to be able to animate at different levels of detail. For instance, it might be desirable to allow a user to "explode" a selected high level action into its constituent subactions (which would itself be in the form of a subtransaction). This might also require that the data be decomposed into its constituents which are handled by the more detailed actions. Conversely, an analyst may wish to compose detailed actions (ie. a subtransaction) into a higher level action, and to make inferences from detailed structured data to give the required values at the higher level.

In fact, Core does not require that actions at one level correspond to a set of actions at a lower level. This fact, combined with the previous experience of data inferencing lead us to believe that action (de)composition was unlikely to be a fruitful path to follow. This is probably not true in standard techniques, such as SASD and SADT, where pure hierarchies are used.

2. Current Status of the Animator

A prototype animator has been implemented in Prolog on the Apple Macintosh. It conforms to the familiar Macintosh interface and has full graphics support for the generation and manipulation of transaction diagrams.

It was intended that the animator should be integrated with the Analyst workstation so that CORE specifications produced by the Analyst could be animated directly. However, this would have proved to be too slow and cumbersome, and it is doubtful a usable response time would have been achievable with the current Analyst implementation. The integration has therefore taken a weaker form by providing

- (i) a transformation tool to transform Analyst specifications into a form used by the Animator, and
- (ii) a mechanism for posting notes from the Animator back to the Analyst to indicate comments or changes required as a result of animation.

The animator can be used for data flow diagrams in general, but it has been designed to take into account some of the specifics of CORE, such as channel and pool data flows (discussed below).

We now discuss some of the main issues raised by the implementation of the prototype.

2.1 Selection

The first two strategies of animation discussed in section 1.2 are currently supported by the animator. In the first case, a selected action is added to the transaction description and diagram but not executed until all such actions have been added, whereas in the latter case it is executed at the same time as being added to the transaction.

The selection of actions is done via menu commands. Any action can be a candidate for selection at any stage of an animation, thus allowing the user to execute actions in random order. However, in the animation of a transaction, one is typically interested only in a small subset of all the actions in the specification, in particular those actions which are connected to the last animated action. Therefore an option is provided whereby the user can mask out all unconnected actions so

that they are not displayed for selection. In addition, facility is also provided for the user to follow any particular dataflow through a transaction.

When animating a CORE specification, it is not unusual that one is interested only in actions performed by certain viewpoints. For instance, a user may want to generate a transaction (CVM) involving only the first level viewpoints. It is therefore useful if the other viewpoints can be made 'invisible' so that the actions they perform are not presented as candidates for selection. This feature - the masking of viewpoints - is supported by the Animator.

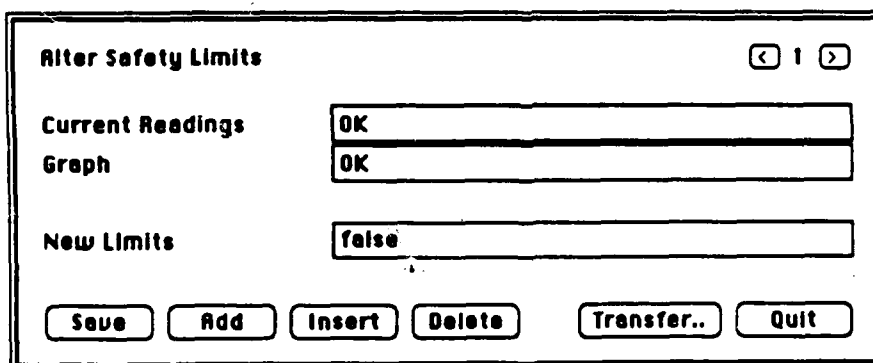
In practice, it was found that the third transaction approach - that of finding a path between two specified actions - was required to suggest possible transactions. A rudimentary tool for finding the shortest path has been integrated into the animator and is used to supplement rather than replace the other selection mechanisms.

2.2 Action Definitions

An action is described by one or more definitions. For simple actions, these definitions can take the form of mappings from inputs to outputs, where the input and output values are simple constants. For more complex actions, the use of variables and built-in functions are provided to enable the description of algorithmic processing of data. These functions include arithmetic functions (+, -, *, /, MAX etc), logical functions (AND, OR, EQ etc) and functions for processing lists HEAD, TAIL etc). As mentioned, there is also a 'Query the user' function which prompts the user for a 'Yes or No' response when it is executed. This facilitates run-time decision making by the user and makes it easier for replaying different scenarios of the same transaction.

For more complex or specialised computations, the Animator allows new functions to be defined as compositions of the built-in functions. These can in turn be used in the definition of other user-defined functions.

Action descriptions are entered in dialogue boxes or forms generated by the animator. A form lists out the inputs and outputs of the action to be described and provides areas for their descriptions to be entered. The example below is the form for describing the action *Alter Safety Limits* (see figure 3.1), which receives *Current Reading* and *Graph* and produces *New Limits*. If the patient readings and condition (*graph*) are satisfactory then there is no need to issue a new set of safe limits for the patient:



Alter Safety Limits	
Current Readings	OK
Graph	OK
New Limits	false
Save Add Insert Delete Transfer.. Quit	

Each form allows the entry of one definition of the action. An action can be described using as many descriptions as are needed.

Definitions entered in this format have to be translated into Prolog rules which are executed during animation. When an action is animated, the animator will take the input values supplied during animation and try to find a definition of the action that will match. The output values specified in the matching definition will then be the result of the animation. In the event that there is

more than one matching definition, the first definition defined will be used. If there are no matching descriptions, an error message is printed.

2.3 Executable Conditions of Actions.

Action descriptions describe *what* an action does. They do not specify *when* an action can be performed, which most often depends on the *availability* of data rather than the actual *values* of the data. In the animator, the user can specify the conditions for an action to be executable (figure 3.3):

- (i) when all its inputs are available;
- (ii) when at least one of its inputs is available; or
- (iii) when a certain combination of inputs are available.

In those cases where action execution is dependent on the input data **values** (rather than just the presence of the data), we consider that the action executes, but can be defined so as to produce no (or null) output. We have found this approach to be adequate for CORE and the examples used so far, but do not know if it is sufficient in general.

The executable conditions of actions are taken into consideration during animation. The user can specify that only executable actions are to be displayed for selection (see figure 3.2); all other actions are then masked out.

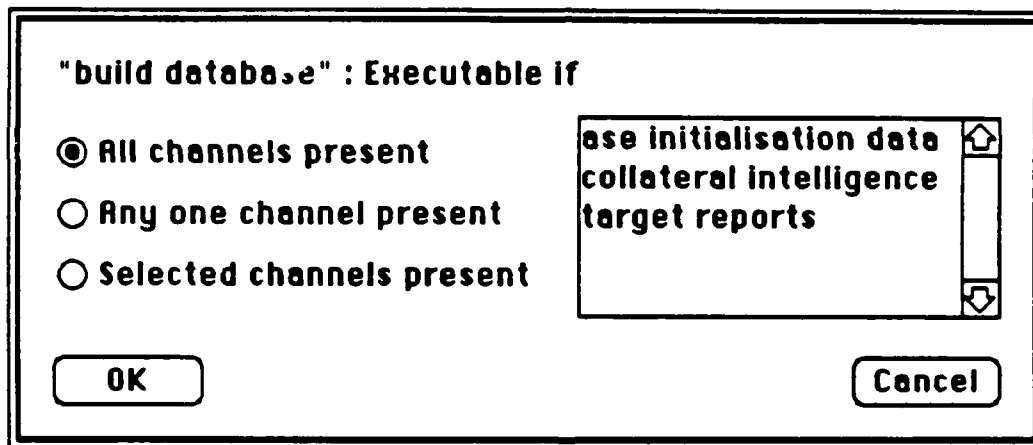


Figure 3.3

2.4 Channels and Pools

CORE allows a data flow to be defined as channel flow or pool flow. A **channel flow** means that every data item generated must eventually be consumed and processed by the connected action(s). This means that channel data arrival at an action is a significant event and could be used to trigger execution of an action. A **pool flow** means that only the latest value of the data is of interest (cf. data base with non-destructive read). Data arrival is analogous to an update and is therefore not eligible as an action trigger. The animator makes use of this information to infer the default rules for action execution. Arrival of data on a channel flow can trigger an action whereas a pool flow can not (figure 3.3).

In the animator, a channel is implemented as a queue of infinite length, with data values being stored in the queue until processed. A pool is implemented as a buffer of length one, so the previous value of the data is overwritten when a new value is generated.

2.5 Transaction Replay

Having constructed and executed a transaction, the user has the option of recording the complete run for future replays. A **run** (or **scenario**) is an instance of a transaction for which the client has selected particular data values for the information flows. A **replay** follows the exact path taken by the original run, and uses the same input data. However the user has the option of altering any of the data values. This provides a quick way of testing a transaction against different sets of input data.

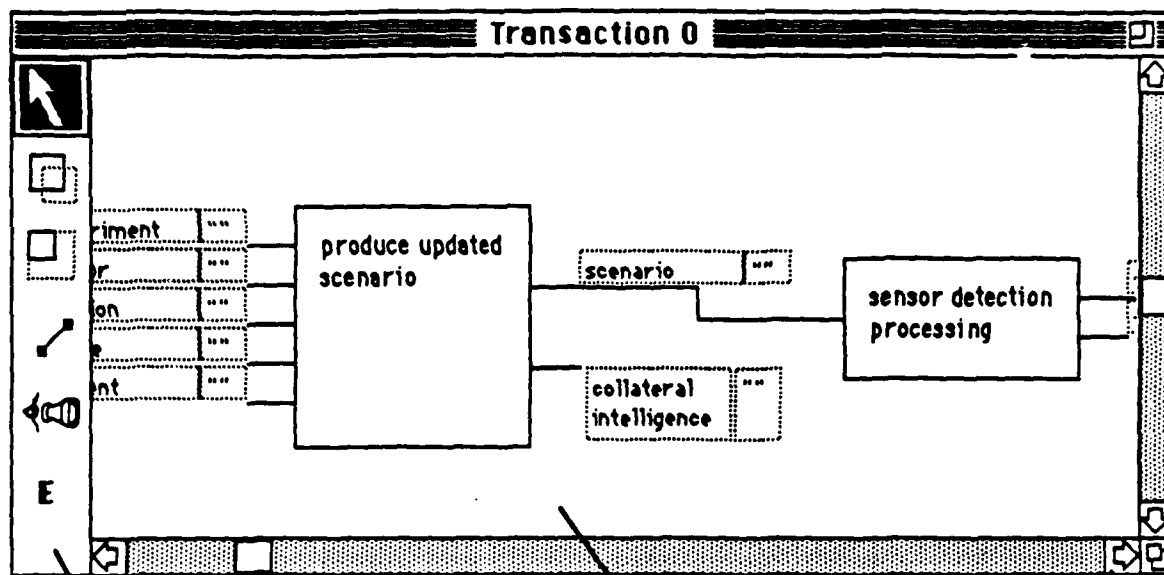
A run can be replayed in one of 3 modes : Go, Go-Go or Step-Step. A replay under the Go mode plays back a run from start to finish without the option of changing input values. Go-Go allows the user to stop at break points in a replay. This feature is useful in the replay of a run where one is interested in the consequence of changing the input values of only a few actions in the transaction. A Step-Step replay is the extreme case of a Go-Go replay. This allows the user to step through the transaction and alter the input values of any action.

2.6 Graphical Interface



The form of interface to the animator prototype was originally purely textual. This provided adequate means for validating the approach, but it was obvious and known that a graphical interface was essential to provide a user with control and feedback in a form compatible with the Analyst and the Core notation. A comprehensive set of graphical tools has now been provided.

The transaction window (figure 3.4) is the main window which provides all the drawing and animation interaction controls necessary. The drawing tools include facilities for line (data) and box (actions) drawing, selection, dragging and resizing. Although simple automatic line drawing facilities are provided, user control is also provided. Similarly, the initial positioning and sizing of action boxes can be controlled by the use of an "anchor": a shadow placed to indicate the position and size of the next action. In addition, data value inspection and alteration, and action definition and execution can all be performed via the graphical interface.



Tool Palette

Viewing pane

Figure 3.4

In order to view the current requirements specification in an overview form, the Core viewpoint diagram (figure 3.5) can be displayed. This same display can then be used select those viewpoints whose actions are to be visible during action selection.

The full set of graphical facilities are described in the Animator User Manual.

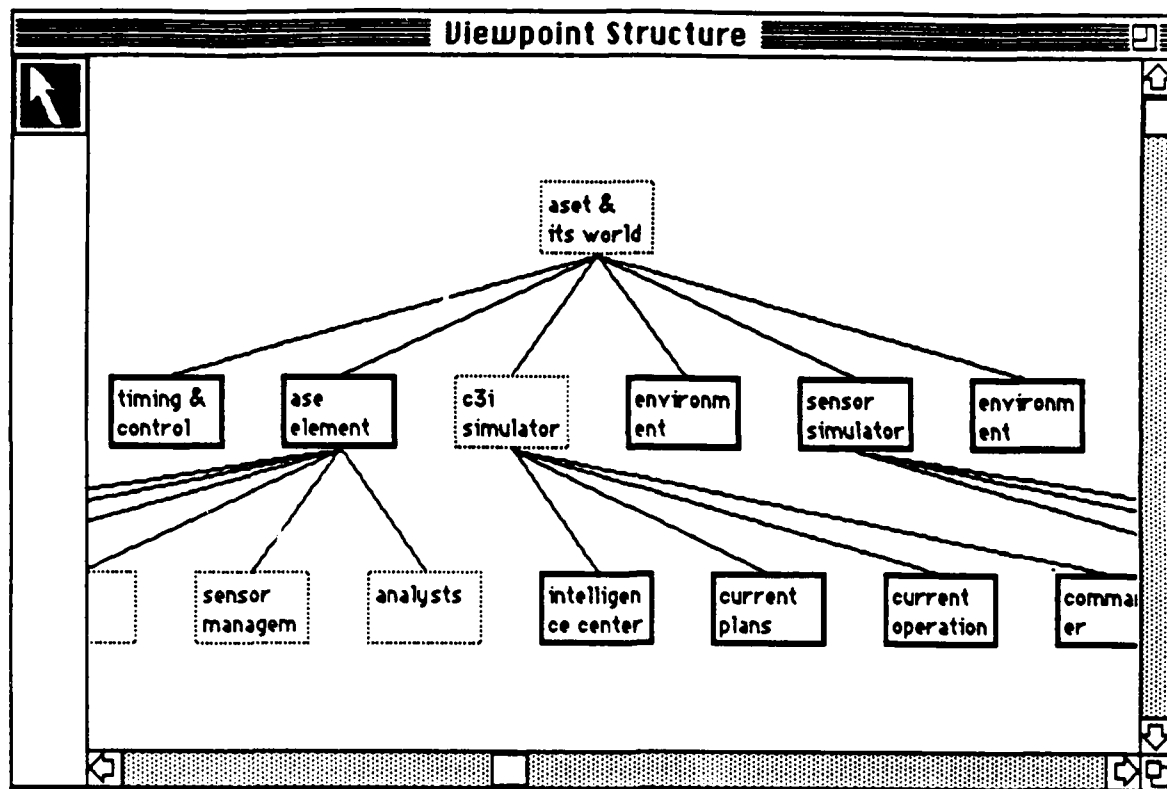


Figure 3.5

2.7 Timing Analysis

One related aspect which is worthwhile pursuing further is that of timing behaviour. The initial approach is to associate time delays with both data flows and actions. Again, using transactions as the unit for analysis, an extended animator would provide information on the timing of events and support some analysis of the expected delays between some causal event and some resulting action i.e. using little more than simple addition. This can be used to provide an estimate of response times. Since there is some stochasticity in the system (eg. some times are specified in ranges, some actions are iterations and alternatives), more complex models are necessary for more accurate timing calculations.

Performance can be calculated directly from the CVM, or in terms of a derived performance model. The performance model is a directed graph in which the nodes represent delay points and the arcs temporal dependencies. There is a trivial correspondence between performance model components and CVM components. The calculation of performance properties of the network from the components depends on a propagation principle. Two could be investigated: a simple model in which maxima and minima are interpreted literally and properties are propagated by addition; and a Gaussian (PERT) model in which component's properties are assumed to describe a normal distribution of timing behaviour, maxima and minima are taken to refer to the mean $\pm 2SD$ and properties are propagated by combining distributions. A third approach which appears promising from the literature is the use of a Markovian mathematical model to derive timing and sizing requirements. McCabe [1985] describes how actions can be represented as process states in a Markov chain and data flows as transitions. By assignaing probabilities to each transition and arrival rates to sources, probabilistic queuing models can be built up and analysed using queuing theory.

Since performance modelling was not considered to be one of the prime areas of investigation, and since the animator can be used to produce transactions of interest, the preferred approach is to attach performance attributes directly to a CVM (transaction). It is our intention only to provide simple timing analysis of the kind necessary to find serious response problems and perhaps performance bottlenecks; this can be better supported interactively as part of the animation of a transaction. Thus each scenario should have a clock that starts at the time of the initiating event and is updated as the user progresses through the transaction. With the proviso that this analysis is only examining a single transaction and ignoring any interaction with other transactions, we believe that this provides a rough but useable form of analysis. A detailed description of the proposed performance extensions for the animator are described in Appendix B. A separate prototype tool which supports the above approach has been implemented but not yet integrated with the animator.

3 Conclusions

We are convinced that animation has a useful role to play in requirements analysis, and that the approach and tool support that we have described offer a simple yet powerful means to this end. *Program animation* has been used successfully to provide "visual feedback" and insight into the actions of programs [London 85]. We expect similar success for requirements analysis, where the importance of correct analysis and specification is even more crucial.

Briefly, animation has been based on the notions of **transactions**: collections of actions which represent some scenario in the specification. Transactions are selected by a process of action-by-action browsing and animation. Action descriptions are given by sets of mappings from inputs to outputs, using some simple built-in functions and user-defined functions. Control of action execution is given by simple rules based on data arrival. Facilities are also provided for transaction replay, with data modification if desired.

The current animator has proved the usefulness of the approach, even though the current tool requires extension in a number of directions. We have tried to avoid requiring that data and action descriptions become too detailed, but, for execution, obviously require some indication of the processing that each action is required to perform. Querying the user during animation has provided a simple yet convenient means for handling complex, changing and human-provided actions. We consider that the notion of transactions and animation provide a sound basis for interpretation and validation, and can be easily extended to permit attachment and interpretation of timing constraints.

One of the most promising extensions still to be investigated is the use of transaction validation predicates to specify the acceptance criteria for a transaction. The user would be encouraged to specify the criteria to be used in accepting a transaction as satisfactory. This could be in a similar manner to that which is used for action definition: by acceptable mappings from inputs to outputs using functions, cases etc. In this manner, the user could add cases as required while trying out a number of scenarios, and/or various scenarios of a transaction could be validated by the animator itself. In addition, this approach would help to elicit the acceptance criteria of the specified system. With the prior integration of the performance facilities discussed above, timing and performance constraint validation could also be elicited and validated.

Chapter 4

Re-Use

1. The Approach

1.1 Introduction

Despite the fact that reuse has been identified as an important area in which economic benefits can be realised the technical literature is relatively poor and concentrates primarily on reuse of program code which is not particularly relevant in this context. There are numerous estimates of the benefits that reuse might bring and some anecdotal accounts of introducing rigorous program code libraries into software development organisations. The literature on object-oriented programming includes some relevant discussion on reuse and inheritance as for example in Smalltalk-80 (Kaeher & Patterson 1986). Tool support receives almost no attention though important and interesting work on programming environments is reported by Yeh et al. (1984) and on support for reuse of software designs, though in a very different style from that discussed for requirements below, by Kundt (1984).

1.2 Reuse in CORE

CORE as it stands incorporates no notion of reuse, indeed it can be argued that the underlying philosophy of methods like CORE which proceed in a "top-down" fashion from the identification of viewpoints, agents or the like, actively militate against reuse which is inherently "bottom-up". Reuse has to be retro-fitted to the method. To do this some preliminary decisions must be made, most notably the choice of the reuseable building block. We have chosen transactions (CVMs) which seem to us to be (i) manageable in size, (ii) sufficiently information rich to offer a return over and above the cost of use and management of a reuse library, (iii) cognitively acceptable. The disadvantage of basing reuse on transactions is that transactions as such are never explicitly manipulated by CORE, unlike for example viewpoints or data flows. Transactions are orthogonal to a decomposition of a system by viewpoint and "drop out" of the analysis as a "by product", albeit a very useful one.

1.3 Analogy and reuse

Analogy is a subject that has received considerable attention in the literature on AI (Winston 1980). An important area in which applications of this work are sought is legal reasoning (Ashley 1984). A feature of legal reasoning is the "precedent problem" discussed by Gentner. An example of this problem is as follows: given a by-law such as "no vehicles on the pavement" which is known to apply to the riding of bicycles on the pavement does the by law apply to Eleanor who has been caught roller-skating in a pedestrian precinct? Alternatively given Eleanor roller-skating in a pedestrian precinct how could one retrieve the riding of bicycles on the pavement from among a mass of other cases relating to snails in ginger beer bottles, matricide and so on?

This problem appears to us to be very similar to reuse of specifications. For example having partially captured details of a system whose requirements we are analysing, say the alarm component of a patient monitor, can we reuse features of the fully analysed burglar alarm component of a perimeter security system?

We have attempted to exploit the similarities between reuse and analogy to suggest suitable strategies for reuse and to provide a coherent model of reuse on which tool support could be

constructed as a base for experimentation.

1.4 Model of reuse

Figure 1 shows a systems block model of reuse. The model indicates the major components of the reuse task and is the basis of the tool TRUE developed to support reuse. In outline: the application concepts are the "primitive needs"; the base contains the previously analysed transactions; the target contains the emerging current analysis in which we wish to obtain the benefit of reuse; strategy is the means by which a suitable transaction is determined and allocation is the deployment of the selected transaction in its new context; views govern the way that the (re)user can access the reuse process and method guides and organises it. These are discussed in detail below and an example is analysed.

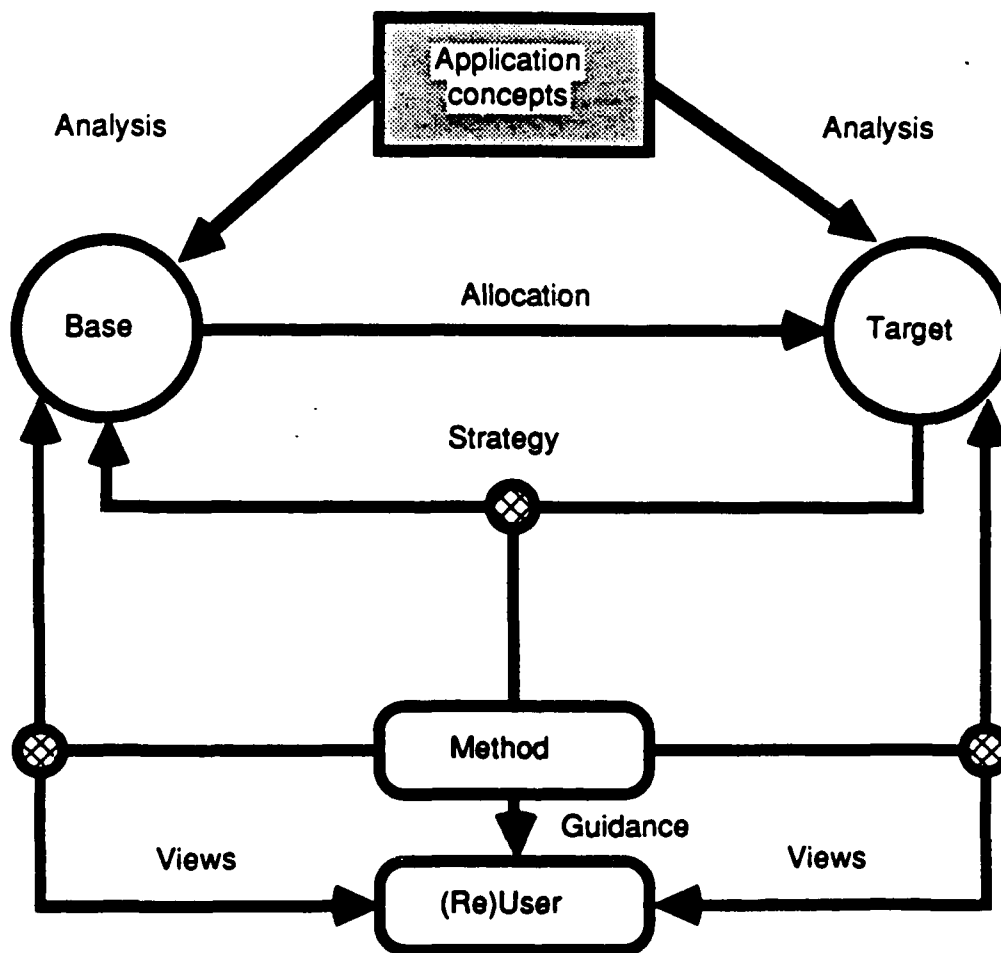


Figure 1: A model of reuse.

2 TRUE: The Reuse Tool

2.1 Application Concepts

An application concept is the starting point for requirements analysis though it varies in form from project to project- it may be a large document packed with technical details or it may be a vague and unarticulated need on the part of a client. For the purpose of our discussion of reuse the application concepts form the "environment" from which domain knowledge is drawn.

Example

To illustrate the model an example, or more exactly a pair of examples are reviewed. The examples are liberally adapted from two well known non-trivial case studies, a patient monitor system and a burglar alarm system, for which there are clear informal specifications in Downes & Goldsack (1982) and McDermid & Ripken (1983) respectively.

Analysis

Analysis is the process of elicitation and representation that transforms the application concepts into a verifiable and validatable form that facilitates design and implementation of the required system. In the case under consideration the process of analysis is that detailed by the CORE method with some minor modifications.

2.2 Structure: Base and Target

There are two structure knowledge bases in our model of reuse shown by the blocks marked base and target. The structure knowledge bases contain the information derived from analysis, that is a set of linked objects and attributes representing a description of system requirements. In our case these take the form of CVM transactions. Each set can be partitioned into those attributes which are derived directly from the CORE analysis, for example actions and viewpoints, and those added to enhance the possibility of reuse such as synonyms and class inheritance links. The addition of these attributes are the minor modifications to CORE referred to above.

2.2.1 Base

The base structure knowledge base contains details of transactions which have been completely analysed and assimilated into the knowledge base with reuse in mind. In legal reasoning analogy this corresponds to the base of precedents such as that in our previous example- "riding bicycles on the pavement is a breach of the by-laws".

Example

In our example the burglar alarm system monitor security status transaction is stored in the base structure knowledge base. Figure 2 shows the CVM diagram for the transaction which is stored in the form specified by the Analyst tool. Some typical added attributes are:

Immediately superordinate class: safety device
 Purpose statement: "To notify of exceptional security conditions"
 Project: airfield
 Action synonyms for poll sensor action: poll, monitor
 Action synonyms for check real alarm action: confirm

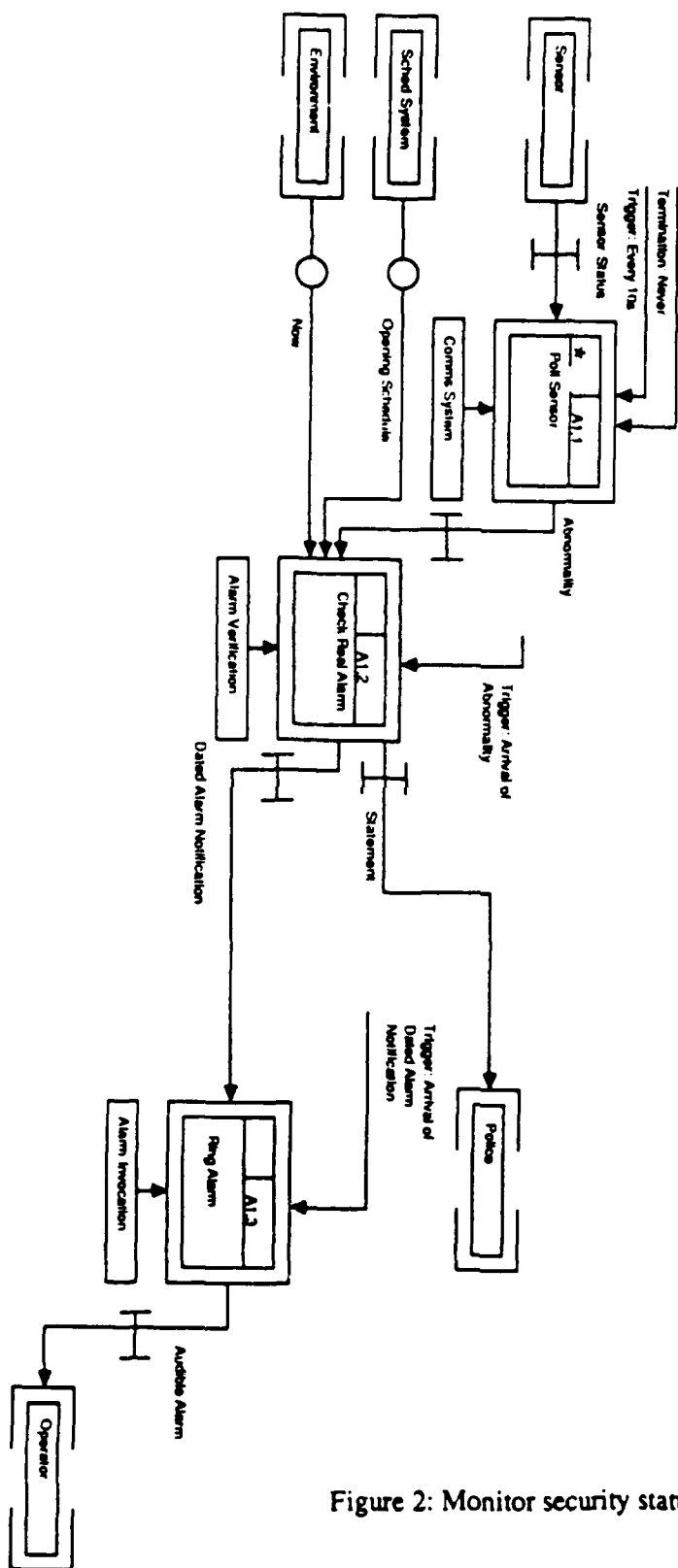


Figure 2: Monitor security status transaction (burglar alarm).

2.2.2 Target

The target structure knowledge base contains the current analysis in its emerging state. In legal reasoning analogy this corresponds to the known facts about Eleanor and her roller-skates. It gives details of the target area for reuse.

Example

The target in our example is the monitor patient condition transaction from the patient monitor example. We shall assume that the analysis has been completed and the target may be viewed as a CVM transaction, this is shown in Figure 3. The objective of reuse in this setting is to find a validated transaction or a transaction for which an established design solution exists. Some typical added attributes are:

Immediately superordinate class: safety device

Purpose statement: "To notify of exceptional health conditions"

Project: hospital

Action synonyms for emergency actions action: secure

Action synonyms for monitor: poll

2.3 Views

Views are perspectives or presentations of the knowledge in the structure base. In this setting the views are defined by information hiding on the structure base. There are two basic groups of views in our model of reuse global and contextual.

2.3.1 Global

Global views are views which look at the structure base from outside the setting of an analysis. These views concentrate on the added attributes. Typical of such a view is a class inheritance browser which permits navigation around the structure base along specialisation/generalisation links. Another global view is the dynamic view of transaction behaviour obtained through animation.

Example

Figure 4 shows as an example a class inheritance specialisation browser for the superordinate class safety-device. There is only one member of this class, burglar alarm in the base knowledge base.

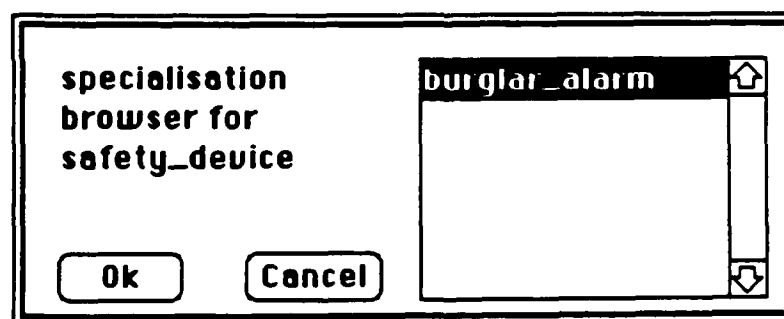


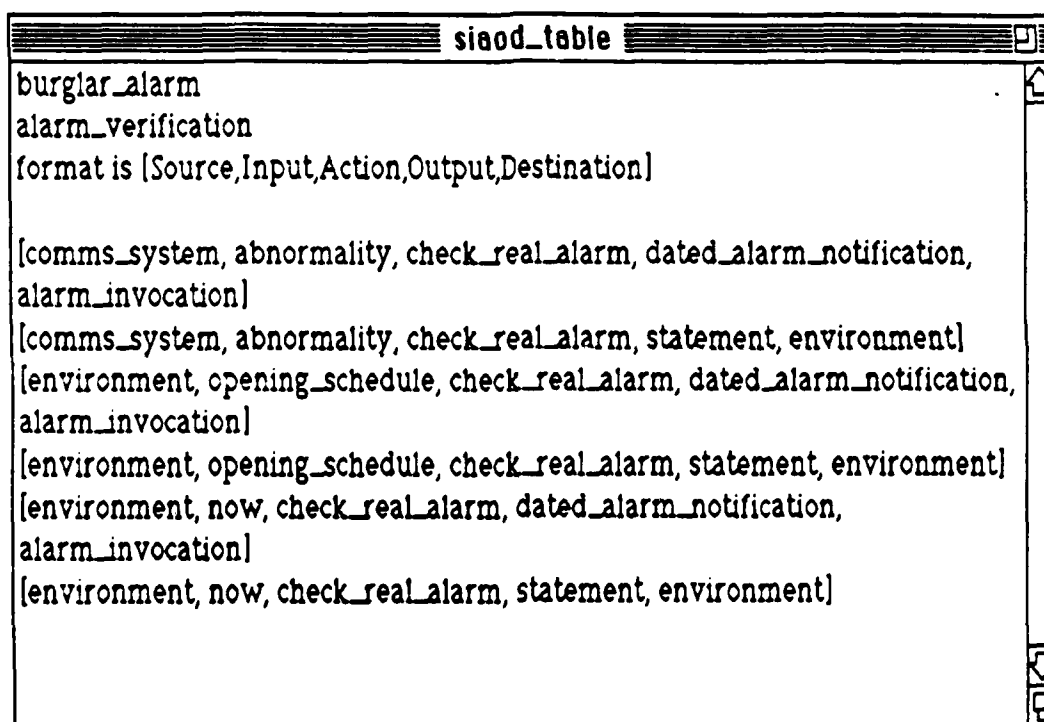
Figure 4: Example of class inheritance specialisation browser.

2.3.2 Contextual

Contextual views look at the structure base through the filter of analysis steps. They are effectively post-hoc reconstructions of these steps. In CORE a fragmentary viewpoint structure can be inferred from a CVM transaction and presented as a view "in the context of the viewpoint structuring step", similarly a set of skeleton tabular collections can be derived from such a transaction, and presented as a view "in the context of the tabular collection step". There are a complete set of contextual views covering the whole CORE analysis.

Example

Figure 5 shows a tabular collection contextual view of the burglar alarm transaction (in Figure 2) for the alarm verification viewpoint.



siaod_table	
burglar_alarm	
alarm_verification	
format is [Source,Input,Action,Output,Destination]	
[comms_system, abnormality, check_real_alarm, dated_alarm_notification, alarm_invocation]	
[comms_system, abnormality, check_real_alarm, statement, environment]	
[environment, opening_schedule, check_real_alarm, dated_alarm_notification, alarm_invocation]	
[environment, opening_schedule, check_real_alarm, statement, environment]	
[environment, now, check_real_alarm, dated_alarm_notification, alarm_invocation]	
[environment, now, check_real_alarm, statement, environment]	

Figure 5: Example of tabular collection view.

2.4 Selection Strategies

Strategic knowledge defines the way in which appropriate, similar or analogous transactions are actually selected. The strategies map between the target and base domain models and as such form the kernel of reuse. We have broken down the practical strategic schemes into some rough and ready categories.

2.4.1 Pattern matching

Pattern matching is the most straight forward of strategies. It relies on finding a similar attribute or group of attributes in the base domain model and in the target knowledge base. Pattern

matching strategies can be enhanced by the use of importance-domination. In importance domination a weight is attached to each "slot" in the structure base schema. The weight corresponds to the assessed importance of finding a pattern-match between the contents of the slot in the base domain model and the contents of a specified slot in the target domain model. Scores from a match across several slots or a multiple match can be combined using a simple combination function. In such a scheme a match between the name of an action performed in the target domain model, which encapsulates its overall functionality, and a listed action synonym in a base knowledge base transaction might rate a moderate score of say 0.6 on a 0-1 scale whereas a match between all hanging data flows would be given a very high score of say 0.9.

High level

CVM transactions, the basic reuse component, are structured objects. Pattern matching on attributes of these objects can be used at the different structural levels. High level pattern matching treats the transaction as a single functional block, a kind of "super" action. This block has a name, net input and output data flows as well as added information such as the name of the project from which it was drawn, class inheritance links, transaction synonyms and so on. High level pattern matching will find a match where the functionality of the base and target systems are similarly distributed. In a CORE analysis of a target if all the component actions of a transaction fall within the immediate set of sub-viewpoints of a single viewpoint then high level pattern matching may find a match with a base for which the same condition holds. If however the functionality of the base is the same as that of the target but for various reasons differently distributed, for example spread at different levels across a large number of viewpoints, it will not always be possible to find a high level match between the two.

In legal reasoning analogy high level pattern matching will recover base cases where the law which decided the base case was the same as that of the target but not the cases where the law was different but the issue was the same and circumstances differently distributed between parties to the case. Thus in our example we might find our bicycle-riding precedent but not a, possibly relevant, case brought against the police for wrongfully detaining a roller skater skating on private property.

Example

Figures 6 and 7 show the patient monitor and burglar alarm transactions respectively, bundled as high level actions. It seems likely from a preliminary inspection that the burglar alarm case is reuseable in the patient monitor context. If we use high level pattern matching we find the burglar alarm transaction through a successful match on the name of an output data flow - audible alarm. This is a "low quality" match and consequently we attach a fairly low weight to it. It is possible that we can obtain further matches by looking at the data structures of the nett input and output data flows. For example in Figure 8 there is a match in the structure of the vital signs and sensor status data flows.

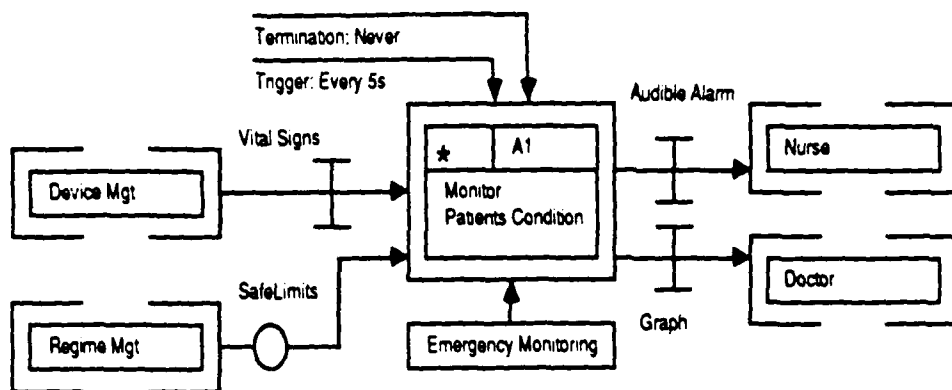


Figure 6: High level view of monitor patients condition (patient monitor) transaction.

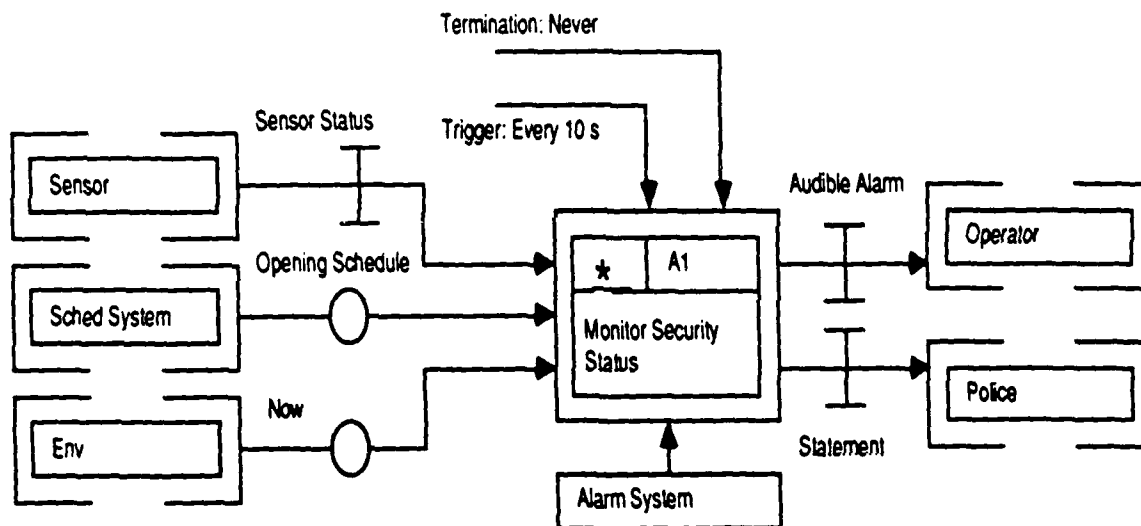


Figure 7: High level view of monitor security status (buglar alarm) transaction.

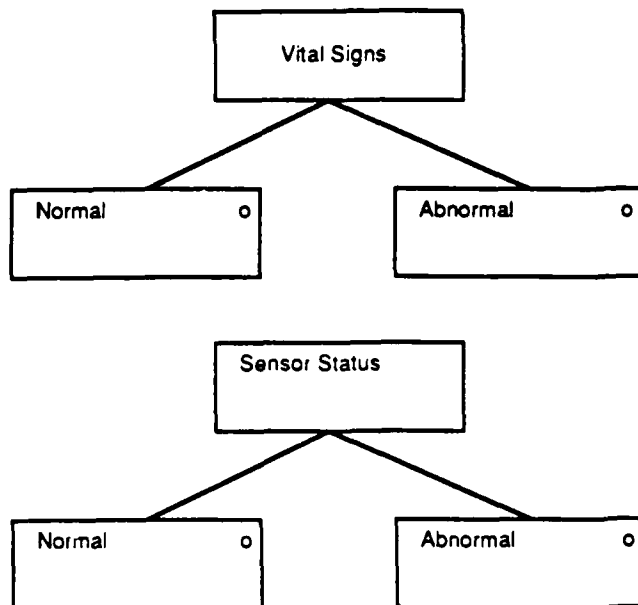


Figure 8: Data structure diagrams of vital signs and sensor status data flows.

Low level

Low level pattern matching takes up where high level pattern matching (on the attributes of the transaction accumulated as a single action) leaves off. It looks at the lower level attributes of the target (the complete set of attributes of the transaction) and seeks matches for them among the lower level attributes of the base. Low level pattern matches are clearly less significant than high level matches but can often turn up interesting and previously unconsidered cases. Low level pattern matching would, for example, have found the wrongfully detained skater through pattern matching on "roller skates". Combinations of low level matches can be strongly indicative of a candidate for reuse.

Example

If we look at the complete CVM transactions low level pattern matches are not immediately obvious. However, having carefully added action synonyms during analysis, the strategy reveals a match, in this case clearly significant, between the Monitor action in the patient monitor and the Poll Sensor action in the burglar alarm by way of the action synonym poll.

2.4.2 Causal chain matching

While the attributes of an object are important in recovering appropriate, similar or analogous objects, common sense reasoning employs other, more focussed, techniques to complement it. Causal chain matching is one of these. The classic illustration of causal chain matching is "the atom is like the solar system". Clearly we do not mean that the nucleus of the atom is like the sun-yellow, hot and massive. Rather, we know that in the solar system the rotation of the planets about the sun is caused by the pull of the sun and from the comparison of the atom and the solar system we want the inference that the rotation of the electrons about the nucleus is caused by the force exerted by the nucleus to be made. It is the causal links, A caused by B, which can be matched

across base and target domain models. Causal chains manifest themselves in CORE as the control structure of CVM transactions. Matching of causal chains in the CORE setting is equivalent to finding transactions with like control skeletons (cliches?).

Example

The two CVM transactions can be reduced to graphs in which the actions are represented by nodes connected by vectors representing exertion of causal force. To clarify matters we can assume that the chains of causation in a transaction follow the arrival of data, reducing the CVM diagram to a network. Where possible action triggers are also used. Matching of graphical networks is a complex field and falls outside the objective of our research, simple minded matching seems adequate for basic tool support though clearly this is an area which merits further investigation. Figures 9 and 10 show the networks for the two transactions. There is no overall match between the skeletons of the two transactions but by inspection a partial match in the sub-structure of the network indicates some reuse possibilities.

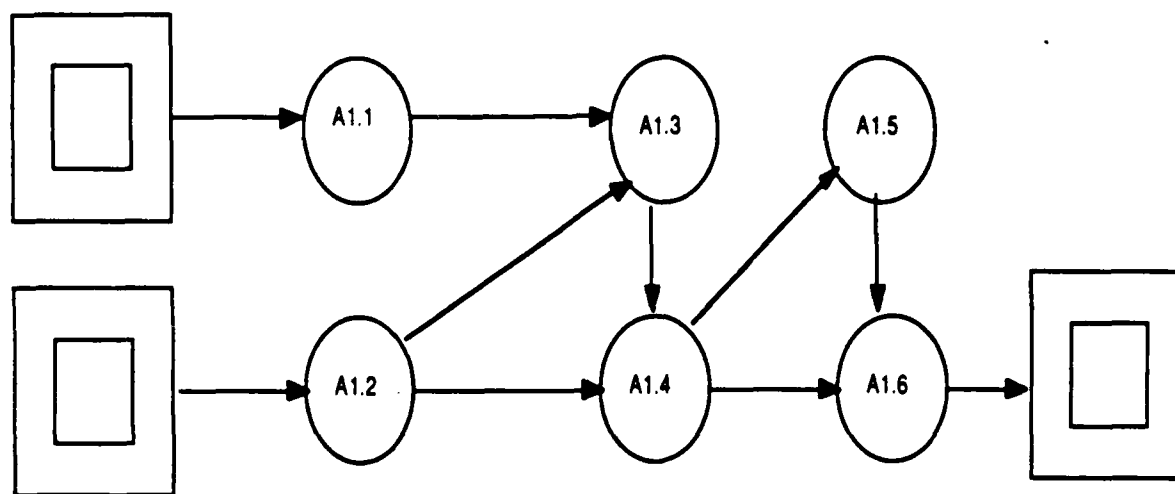


Figure 9: Network for monitor patients condition transaction (patient monitor).

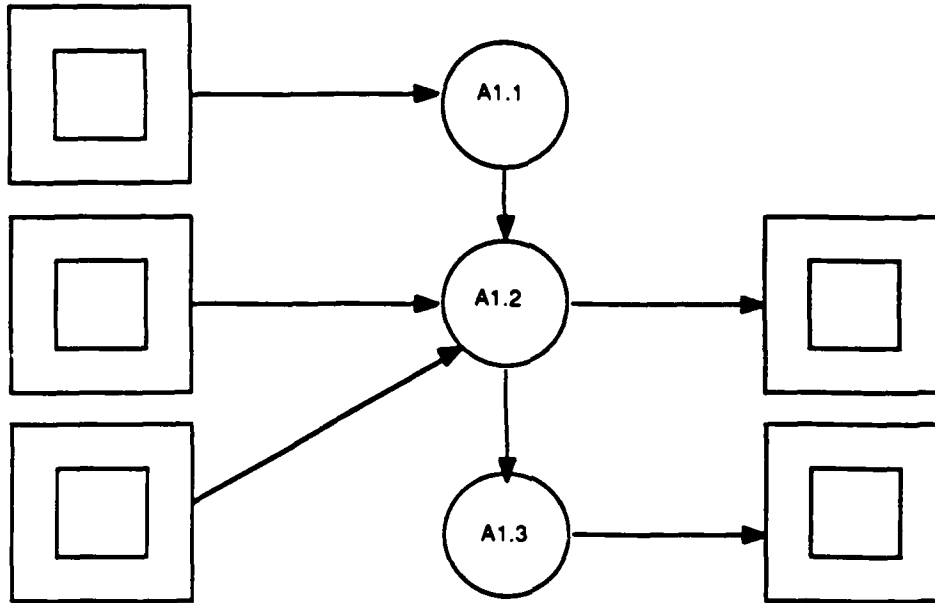


Figure 10: Network for monitor security status transaction (burglar alarm).

2.4.3 Generalisation

Likeness in objects is often the result of family resemblance. Generalisation is a strategy which exploits these resemblances. The Jaguar is a luxury car, knowing this fact we can recover other luxury cars with similar attributes such as the Rolls-Royce and the Bristol. Class inheritance information added to target CVM transactions allows us to find base transactions with common ancestors. Many models exploit generalisation as the sole strategy for reuse, this seems to us to overload a useful technique. Generalisation is powerful only where it is possible to consistently characterise objects and place them in classes, that is where objects have a well understood structure and relationship with each other, also where high level objects are more than names to hang object descriptions off but themselves possess useful content as in the Smalltalk-80 programming environment.

Example

In this case our two transactions are simply related, sharing a common immediately super-ordinate class: safety-device. Figure 11 shows the relevant fragment of the inheritance hierarchy.

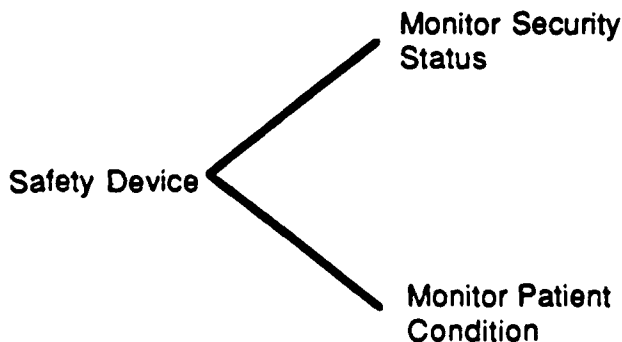


Figure 11: Inheritance hierarchy.

2.4.4 Purpose matching

A product of AI research on analogy is the observation that it is some "purpose" which underlies the facts of a situation and acts as the arbiter of the validity of a generalisation. If we again review Eleanor roller skating in the pedestrian precinct we can see that it is not the attribute of our precedent "bicycles have two wheels" that is relevant but rather the underlying purpose of the by-law "to preserve the health and safety of elderly pedestrians". Thus given that we can provide adequate statements of the purpose of a CORE CVM transaction it is possible to use matching of purpose statements as a strategy for, at least, validating candidate transactions for reuse.

Example

Because of the obvious difficulties of automatically matching purpose statements, browsing is the most practical strategy, particularly where the set of candidates has already been reduced by other means. In our examples a simple direct match of purpose statements is possible as they share "To notify of exceptional security conditions", a very strong match.

2.5 Allocation

Given that a suitable candidate for reuse has been selected from the base it must be mapped across to the target. That is the reused transaction must be put to work. In our model we call this allocation. The allocation task may be broken down into three component tasks which are discussed below.

2.5.1 Restructuring

Restructuring is the redistribution of functionality across the new structure. For example in CORE CVM transactions restructuring is used to resolve inconsistencies between the target viewpoint structure and the distribution of actions in the selected base transaction, in other words allocating the actions of the reused transaction to the new set of viewpoints available in the target context or adds/creates a new viewpoint. Hence restructuring guides the allocation of actions performed in the selected base transaction by a single viewpoint to a target in which there are multiple corresponding viewpoints.

Example

If we decide on reuse in the given example an appropriate restructuring would be (format is Action(Old Viewpoint) —> New Viewpoint):

Poll Sensor(Comms System) —> Ward Nurse

"Poll Sensor currently performed by Comms System Viewpoint will be performed in its new Patient monitor context by Ward Nurse - this seems a suitable reallocation because the Ward Nurse carries out the Monitor action which appears similar."

Check Real Alarm(Alarm Verification) —> Ward Nurse

Ring Alarm (Alarm Invocation) —> Ward Nurse

Boundary (Operator) —> Central Nurse

Boundary (Sched System) —> Doctor

Boundary (Sensor) —> Patient

Boundary (Environment) —> Environment

Boundary (Police) —> Doctor

2.5.2 Plumbing

Plumbing is the connection of the selected base system with the overall target analysis. For example a CORE CVM transaction has hanging data flows and triggering conditions not all of which have sources and sinks in the environment of the system. These data flows and triggering conditions must be made consistent with the data flows and triggering conditions produced or consumed by the surrounding actions. Plumbing sets limits on the extent to which changes in the larger system, consequent on reuse, are allowed to ripple through.

Example

Figure 12 shows the burglar alarm transaction, actions with shaded fill, plumbed into the remnants of the patient monitor transaction. The black squares indicate the main plumbing connections.

2.5.3 Editing

Give that restructuring and plumbing have been successfully completed. There will inevitably need to be some dotting of "i's" and crossing of "t's". In CORE CVM transactions this may mean changing names and perhaps resetting parameters of triggering conditions. Editing allows these changes to be made while maintaining global consistency.

Example

Figure 12 illustrates the completed transaction ready to be deployed

2.6 Method

Methods are rules for guiding and organising activities. In the model of reuse the method controls and ties together views and strategies by prescriptive guidance on their use. The nature of method guidance is dependant upon the state of the target transaction. Typical examples of such method guidance for reuse in CORE are:

"At the tabular collection stage try high level pattern matching on the current set of collected high level attributes, if successful take a global dynamic view of the candidate transactions (through the animator), select the most promising transactions to be viewed in context (a tabular collection view)."

Similarly:

"If no plausible candidates for reuse have been obtained at the tabular collection stage use low level matching on elements of a subset of SVMs, if successful obtain rough confirmation for the candidates by browsing the class structure."

2.7 True: a tool for transaction reuse

True (Transaction reuse) has been developed on the basis of this model of reuse outlined above. It has been used as an experimental vehicle to validate the model. For full details of facilities provided by the tool see the TRUE user guide. The tool contains implementations of all contextual views and a subset of the global views including class inheritance browsing and synonym browsing. Pattern matching strategies have also been implemented though further refinement of

weighting and combination of weightings is anticipated as more examples are tested. Generalisation based strategies have been implemented, causal chain and purpose matching are implemented in a relatively simple minded way though it is hoped that they will be refined in research outside the context of the current project. Allocation has proved difficult to support, currently the system allows allocation and partitioning of the reused transaction and flags outstanding inconsistencies in the emerging new transaction, this is to be the subject of further work.

TRUE is integrated with the Analyst through a translator that converts between the TRUE bases and the Analyst database. No integration with method guidance is currently in place but it is anticipated that this prove to be a minor addition.

3. Conclusions

We conclude that reusability can be added to formatted requirements analysis methods though not in a elegant or "conceptually clean" way. It is clearly preferable to start with a method in which the primitive elements have been selected with reuse in mind. Despite the fact that the tool provides access to powerful strategies and covers the range of practical reuse techniques the structure of CORE effectively prohibits any really extensive reuse. This negative statement is in a way an important and valid conclusion. Reuse cannot be tacked on to existing software engineering techniques but must be built in at conception. Given that reuse in CORE is required the model outlined above, derived from the model of analogy in legal reasoning, can be used to provide a framework for tool support. This model has been partially validated through experience in the ASET case study.in.

Chapter 5

Enhancements to the CORE Analyst

In this section of the Report we shall describe the enhancements to the CORE Analyst not addressed in the main body of the Report. These enhancements fall into three categories, namely the product enhancements funded by Systems Designers plc between Release 1A and Release 2A, functional enhancements implemented between Release 2A and the end of the research not described elsewhere, prototype designs, tools developed under the research funding and superstructure implementations to support the three main research themes.

1. Analyst Enhancements

1.1 Release 2A Enhancements

This section briefly describes the enhancements produced for Release 2A. Details of the exact functionality of these enhancements is given in the CORE Analyst User Guide prepared for RADC.

- * **Porting to Macintosh Plus.** Release 2A runs on both the Macintosh XL and the 1Mb Macintosh Plus with the Hierarchical File System.
- * **The Report Writer.** This is a separate application allowing the production of pre-defined reports concerning the status of the specification.
- * **Changed Project Interface.** A simplification of user entry to the system.
- * **Importing Diagrams.** The ability to import/export diagrams between projects.
- * **User Attributes.** See below.
- * **Demand Checking.** On demand validation of the current diagram and better support for the Local and Project Level checking options.
- * **Automatic Scrolling.** The diagram now automatically scrolls when creating, moving or re-shaping objects.
- * **Method Support.** More rules concerning the CORE method implemented.
- * **Diagram Annotation.** A new object, 'Free Text', is available on all diagrams for textual annotations.

1.2 Enhancements implemented after Release 2A

This section briefly describes the enhancements implemented after Release 2A. Details of the exact functionality of these enhancements is given in the CORE Analyst User Guide prepared for RADC.

- * **User Notes.** This facility allows the user to associate short notes with objects within the specification.
- * **User asked about completion of diagrams.** The Active Guidance system needs to know about the status of diagrams and hence the user is prompted on closing diagrams which have been checked - and no errors found - whether they have completed work on the diagram.
- * **Warning on attempted modification of completed diagrams.** After the user has indicated that work has been completed on a diagram, subsequent attempts to modify the diagram are flagged to the user.
- * **Auto Generate.** The ability to generate initial versions of Data Structuring and Single Viewpoint Model diagrams from Tabular Collections has been implemented.
- * **Extensions of CORE Rules.** The checking rules have been augmented. In particular new checks have been introduced when demand checking diagrams. A rationalisation of the rules activated during demand checking has been performed.
- * **Extensions to the Report Writer.** The information presentation in the Report Writer has been modified and the decomposition checking report has been extended to make further use of information held in Data Structuring diagrams.
- * **Warnings on changing names which have attached descriptions/attributes.** Users are warned what names that have attributes or descriptions associated with them are changed.
- * **Creation of subviewpoint & sub-component lines now independent of direction.** The direction in which the user creates these lines does not now govern the relationship between the objects. This is now deduced from the physical position of the objects on the diagram.

2. Prototype Designs and Tools

Several prototype tools have been developed to complement the work done in the major research areas. These prototypes are described and initial evaluations, where appropriate, are given.

The implementation of semi-automatic transformation from one representation to another has been done. The worth of this tool in general has been considered since various transformations which are possible are not necessarily useful. A particular case in point would be the generation of a Tabular Collection for a particular viewpoint from existing Tabular Collections for viewpoints at the same level. This is not considered a good thing since the construction of Tabular Collections manually indicates to the analyst(s) where inconsistent views exist.

We have considered two different aspects to transformation, namely :-

Direct object mapping: In this case we have a **target** and a **source** diagram. Some or all of the objects on the source diagram are mapped onto objects on the target diagram. An example of this would be the above mentioned Tabular Collection to Single Viewpoint Model transformation.

Transformation by synthesis: This is a more complicated way of autogenerating a diagram and here we consider information from **multiple source diagrams** combining together to form the target diagram. An example of this would be the synthesis of a Single Viewpoint Model diagram from the Tabular Collection and Data Structuring diagrams. We could imagine the time ordering being deduced from the derivation order of the data flows, the selection/iteration of actions being deduced and the introduction of control flows where iterative actions are constructed.

Several prototypes were produced and discarded during the research. The alternative mechanism for the control of animation - modeling a dataflow machine architecture - has been discussed earlier.

The choice of representation for the normative model (see Section 2) was difficult. The Inference Engine for the MAL normative model also suggested several alternatives. An implementation using a forward-chaining diagnostic inference engine was prototyped in Poplog but this has been suspended in favour of the current approach of describing the normative model as Prolog rules that are executed directly.

3. Superstructure Developments

The animation of transactions, in particular the performance analysis, needs information not normally collected/represented during a CORE analysis. A facility to associate arbitrary attributes (or properties) with objects within the specification (viewpoints/dataflows/actions) was designed and implemented. This facility enables a user to associate a named attribute (e.g. Data Arrival Rate) with object-types and to give values for particular objects. The user can also define some validation criteria for the attribute values. This implementation was used as the basis for the Release 2A User Attribute facility. The implementation of attributes only allows simple values of data (string and integer) to be associated.

The ability to associate notes with specification objects was recognised as an important superstructure requirement for all aspects of the RADC research. A generalised note handling subsystem was designed under the research contract and an initial implementation was done by the Analyst Development Team as part of the Release 2A product. The note facility implemented allowed the creation and deletion of notes attached to objects. (This was used for the 'Check Now' Release 2A Facility.) The functionality of the note handling has been extended to allow the creation of user-defined notes and the ability to browse through notes attached to objects. This forms the basis of the prototype remedial advice (q.v.) user interface. Although we have a general design for notes which allows arbitrary types of notes to be created, the current implementation only handles notes of type text. The system also has a generalised interface to allow other tools, such as the Animator, to produce a file containing notes and for this file to be loaded into the specification.

The interface to the auto generation facility and the production of Summary Diagrams required that graphics objects be created and drawn under the control of the Prolog components of the Analyst. In order to achieve this a general interface between Prolog and the Pascal graphics system was designed and implemented.

The graphics interface designed is not a collection of general purpose drawing facilities, as in the Macintosh Quickdraw subsystem. The interface deals with Analyst objects (e.g. indirect viewpoint boxes, channel data flows, etc) and contains primitives enabling the creation, selection, moving etc of these objects.

Chapter 6

The ASET Case Study

The ASET (Advanced Sensor Exploitation Test Facility) case study is a large requirements analysis. The analysis was based on The Advanced Sensor Analysis/ Design Report (RADC-TR-80-168). The system under consideration in this report is an environment and test harness for advanced sensor integration. Such systems analyse and manage data fusion tasks with many diverse sensors. The ASET consists of a basic sensor exploitation system and simulated environment that can be used to explore the behaviour of this form of sensor integration under a variety of battle scenarios.

1. ASET Objectives

The ASET case study as a part of the TARA project has two goals. The primary goal is to test ideas on tool support for requirements analysis by a realistic case study. A secondary goal is for the case study to act as a standard base for comparisons between tools and methods.

The question of whether the case study has been an adequate proving ground for our work is discussed in the other sections of this report. The use of this single large case study as a means of comparison will be evaluated when the TARA tools are examined in the context of other related work sponsored by RADC. As this is a strategic target that lies outside the immediate bounds of the project it is worth noting a degree of scepticism about the results of any comparisons made on the basis of the ASET study alone. This scepticism is not because the comparisons are likely to be impressionistic, this is a serious problem across the field of software engineering, but because we are unconvinced that the requirements domain of the study was suitably all embracing.

Going beyond these goals, in the course of analysing the ASET requirements, we probed the limitations of CORE itself -for the ASET case study as it was presented and as a method in general. As part of this investigation we have reached a number of conclusions, most importantly about how requirements analysis case studies can best be presented. We not included discussion of minor inconsistencies, omissions and errors in the ASET case study such as differences in data flow names and so on.

2. Form Of The Case Study

The ASET documents can, for our purposes, be divided into four major parts. The first contains a "function analysis", this outlines what is to be done to develop, primarily design, and evaluate Advanced Sensor Environments in the form of text and SADT (Ross) diagrams. The second contains the functional decomposition/ candidate system model of the ASET itself primarily in the form of data flow (DeMarco) diagrams and accompanying text. The third, contained in an appendix to the main body of the report, is a PSL/PSA description of the ASE element - the main functional sub-system of the ASET. Finally the report contains a record of decisions as to hardware and software infra-structure and the usual introductions and conclusions.

The function analysis is useful for scene setting but concentrates for the most part on the structure of the ASET *project* as distinct from the ASET *system*, which is the object of our requirements analysis, and can consequently be disregarded. Similarly the hardware and software infra-structure are of little direct importance to this review. We can concentrate on the kernel of the case study contained in the candidate system model supported by the PSL/PSA description of the ASE element.

3. CORE Requirements Analysis: Limitations of the ASET Specification

Too much attention has in general been paid to the vexed question of what is a requirements specification as distinct from a design. From a pragmatic standpoint there seems to be a large grey area in which specification and design are intertwined. Equally there are objects which are clearly requirements and objects which are clearly designs. The classical distinction is that requirements are the *what* of system construction while design is the *how*. While this distinction is appealing for pedagogic purposes it is next to useless for making practical classifications.

A better division can be made by regarding a design as containing information about a system - an optimised set of constraints on implementation while a requirement specification contains information about the way that information about the system is distributed in the world - the sources of constraints. It is not the properties of the representation scheme that determines if a description is of requirements or design but rather its use as dictated by a method. Thus for example the CORE viewpoint hierarchy specifies requirements when it describes the viewpoints as "possessors" of requirements and their relation to each other, in other words when it describes the structure of knowledge about the system. The same technique can be used to represent the structure of a system itself and in that case is a design.

CORE is a requirements analysis method in as much as it aims at identifying transactions. A transaction is the basic unit of identifiable user interaction and thus it is the key category to which constraints apply and can be readily validated. The division of actions and their distribution across the viewpoint hierarchy is a byproduct of this process and can, indeed probably should, be changed during a following design.

The candidate system model, as given, is based on a wholly functional breakdown of the projected system and hence falls in the no mans land between requirements specification and design. This means that to use it as the base for a requirements analysis is not realistic. Even were it a proper requirements specification it would be difficult to derive a CORE specification from it. CORE cuts up the world into a particular set of categories while another requirements analysis method may use a different possibly orthogonal set. For CORE to work as it should, the start point should be a needs statement or an application concept not a pre-existing requirements specification.

CORE was developed for requirements analysis with interactive elicitation from multiple requirements sources as its primary aim. The ASET study is a single static document presenting little opportunity for testing the consequences of validation nor using the full range of method heuristics. Scale alone does not make a case study realistic it only tests clerical aspects of a method.

4. The Analysis

The analysis was carried out at both sites (SD at Camberley and IC at the South Kensington campus). The viewpoint hierarchy was agreed between the sites at an early stage though frequent revisions were required. Members of the project team separately assumed responsibility for sub-trees of the viewpoint hierarchy. Regular meetings were held to determine policy, note difficulties and to minimise naming inconsistencies. SD assumed responsibility for maintaining the evolving specification on the Analyst and reconciling minor inconsistencies as they arose. David Galley (SD) and Graham Brown (SD) played the roles of customer authority and user authority respectively.

CORE comes in several different flavours as set out in "CORE-the method", as described by Mullery (LNCS 190), as embedded within the Analyst and so on. We have adopted a pragmatic variant designed to expedite the ASET case study while keeping as far as possible to a version of CORE compatible with testing the project tool set.

5. Discussion Of The Use Of CORE

5.1 Viewpoint structuring

The difficulties reviewed in the sections above presented themselves most clearly in the viewpoint structuring step. The breakdown in the existing document is functional whereas CORE viewpoints have a structural flavour (This feature of CORE is often misunderstood. CORE viewpoints are not functional sub-systems but are more akin to "agents" or "roles" in a system.) In a normal requirements analysis with the possibility of direct elicitation and a degree of domain knowledge on the part of the analysis team it is possible to freely select the shape of the viewpoint hierarchy. We have felt constrained to maintain as far as possible the existing functional structure and merely provide a rough translation into CORE terms. This is of course good enough for tool evaluation but it is not CORE as it ought to be presented. For example consider the ASE element with the functional blocks Build Data Base, Manage Data Base, Exploit Data Base, Route & Filter and Manage Sensors and contrast this with a CORE viewpoint which is "something you can pretend to be ... a lift, a door, a passenger".

5.2 Tabular collection

A well known difficulty which inexperienced users of CORE encounter is handling data flows which are internal to a viewpoint and thus to a tabular collection form. Generally if you have selected your viewpoints correctly this is not too much of a problem. The occasional internal data flow indicates the possibility of a memory item and where it cannot be easily eliminated a note can be kept for a later design stage. Though CORE does not handle data storage very well internal data flows should not have been the problem they turned out to be in the ASET case. The cause for this was the poor viewpoint breakdown presaged in the discussion of viewpoint structuring. Actions identified as performed within a viewpoint appear again as sub-viewpoints because of the exigent functional viewpoint structure. Where this happens in a single viewpoint more than once a clash occurs between the tightly coupled sub-viewpoints and the elimination of internal data flows. For an illustration of this see the breakdown of the ASE element.

5.3 Data structuring

Data structuring has also proved difficult in the ASET case study though not by reason of the viewpoint breakdown. Some data flows in the original analysis, but not in the PSL/PSA description are loosely or inconsistently labelled - a normal occurrence in requirements analysis. In addition their relation to each other is not clear; for example the data originally specified as files but appearing in CORE as data flows such as Commanders Requirements, Surveillance File, Current Mission, Tasking Queue, Correlated Reports and so on. In a proper requirements analysis this would be an area in which we would have relied upon interviewing and document analysis to determine a solution. Unfortunately this was not available to us consequently leading to a possibly faulty analysis.

6 Discussion Of The ASET Specification

6.1 Timing and Control

The case study notes that the functions of the ASET are complexly interleaved in time and consequently establishes a requirement for synchronisation and coordination. A requirement of this

form is relatively easy to represent using CORE. Unfortunately the case study goes on to provide a specific solution to the requirement by routing all data between functional sub-systems through timing and control. By this means they establish a disguised (and probably not very sensible) design decision: in what is supposed to be a requirements document, make life difficult for themselves because this solution confuses their data flow diagram oriented presentation and make life near impossible for the CORE analyst.

In our analysis we have reversed the decision and distributed Timing and Control across the viewpoint hierarchy eliminating what we feel is design detail. In addition we have not shown synchronisation signals associated with each action but have kept an overall note to the effect that they should be included. Consequently our presentation appears clearer than the original and the completed transactions are simpler and more representative of the projected system behaviour than would otherwise be the case.

6.2 User interface

The ASET system includes a number of operators, analysts and users all of whom contribute to the systems operation. Without details of their function distinguished from the function of the sub-system with which they are associated, notably omitted from the document as it stands, the options are limited for handling them in CORE. We have variously absorbed the operators into a suitable viewpoint, introduced a parallel indirect viewpoint or split the viewpoint in two. Where possible we have sought to do away with explicit user action on the grounds that including it constrains design.

Screen "mock ups" and similar are often included in industrial requirements documents as in the ASET case study concept of operation but rarely feature in requirements analysis methods. The abstract specification of user system dialogues perhaps using grammar based notations might be an improvement to methods giving a user focus at an early stage of software development.

7. Gains from the use of CORE

CORE has been a vehicle for our investigations of issues in tool support rather than being the main subject in and of itself. As such CORE represents the minimal basis for our work. It is inevitable therefore that the gains of using CORE are scantily covered in this report. Certainly short comings of formatted methods are well known and CORE problems with the ASET case study amply rehearsed above. There are significant gains in the use of CORE over the existing functional style analysis:

We identified further inconsistencies which suggests CORE is more thorough - or at least the graphic representations makes inconsistencies more visible and the heuristics are superior.

The viewpoint decomposition, flawed as it is, is a substantial improvement on the structuring of analysis as it stands.

CORE raised significant issues such as the handling of timing and control.

The use of CORE was felt to link well with the management discipline required to coordinate the effort across sites.

CORE gave all the participants in the analysis a good "feel" for the envisaged working of ASET.

CORE and the Analyst easily handled problems of this size without problems or excessive clerical overheads.

Chapter 7

Conclusions and Further Work

The work in the TARA Project and described in this report has concentrated on three major approaches to the support of requirements analysis: method guidance, animation and reuse.

Method guidance is supported by a method model used to describe the sequence of method steps that should be followed. This model is directly interpreted by the tools to provide advice and reasoning. It is used in conjunction with rules used for consistency checking to provide remedial advice.

The **animator** provides facilities for the selection and execution of a transaction to reflect the specified behaviour given a particular scenario. Actions are described in terms of input-output relations. Simple rules can be specified to control the execution of actions. Facilities are provided to replay and interact with transactions.

Reuse is supported by facilities for identifying candidate transactions from a reuse database. The search strategies provided include browsing in an inheritance structure, different levels of pattern matching, causal chain matching (matching of the underlying control structures), and purpose matching. Support is then provided for the allocation of the selected fragment to the target environment.

The approach was tested by implementing a prototype set of tools for the CORE method and the Analyst workstation. A major case study, the ASE (Advanced Sensor Exploitation) test environment, has been analysed and specified using CORE, the Analyst, and the tools described above.

This section briefly evaluates the results in each of the three areas, and provides some indication of the directions for further work.

1. Method Guidance

1.1 Evaluation

The evaluation of Active Guidance was conducted using a set of snapshots of the specification, the ASET Case Study, and comparing the recommendations of an expert in the CORE method with the recommendations given by the Active Guidance System. The details of the evaluation are given in Appendix C.

Each evaluation starts with an overview, provided by the analyst, of the state of the specification.

The expert was given paper copies of the specification and asked to recommend tasks, in a priority order, that the specifiers should do together with the reasons for doing the task and why the task is considered important. The expert was also asked what the current area of activity was.

The advice system was invoked on the specification. In most cases the guidance system was run several times with varying priorities. The results of the active guidance system are documented.

Each evaluation ends with a conclusion explaining any differences between the expert's recommendations and the recommendations of the guidance system. Corrections to the active guidance system were incorporated wherever possible.

Conclusions

- In general the remedial advice advised, in almost correct order, the advice made by the expert. The expert would, however, on some occasions take a more strategic view and recognise the existence of higher-level objects in the specification such as database, people.
- Many rules about CORE or modifications (relaxations) of existing rules are used by CORE experts but are not present in the Analyst.
- The experts feel free to slightly modify the syntax of CORE to emphasise a particular facet of the specification. This is not possible in the Analyst.
- The experts also feel free to introduce new semantic interpretations which, although mostly 'obvious' to humans, cannot be interpreted by the tool.
- The experts felt that the combination of Active Guidance and Summary Diagrams was useful in understanding the state of the specification but extensions could be incorporated into the Summary Diagram to further investigate the status of diagrams - last modified date, version number etc.

1.2 Conclusions and Further Work

We have concentrated on providing advice based on the *structural* contents of a specification. We believe that we have covered most of the obvious structural defects that can occur within a specification and provided strategies for their correction but have discovered other structural anomalies which could be detected without recourse to semantic analysis during our evaluation. We have categorised these into six categories :-

1) Defects pertaining to confusion about types and instances.

Within CORE (and other methods) the data items within dataflow diagrams represent types of data and do not show particular instances. This leads, however, to 'unnatural' dataflow diagrams when an action is updating an instance of a dataflow - typically a pool dataflow representing some form of data storage - and is effectively producing a new instance of the data item using the previous instance together with new information. It would appear that the 'best' representation within CORE of such a real-world behaviour is non-obvious and that an automated tool could, using simple structural heuristics, identify this phenomenon and recommend remedial action.

2) Defects concerning local underspecification.

These defects are readily detectable by the current system but work would be needed to implement the relevant checking rules. Defects in this class would be actions on Single Viewpoint Models which do not have a channel dataflow as an input (how then is the action triggered?), actions which have multiple channel inputs etc. Many of these problems are dealt with in the Animator but the integration of such checks within the on-line system remains to be proved.

3) Defects based on misnaming and/or analysts specifying at different levels of abstraction.

The problems of different names for the same object and referring to an object using the

name of one of its components can, we believe, be largely overcome at a structural level. The current system enables the user to associate *synonyms* (as a user defined attribute) with an object but no account is taken of this in the consistency checking rules. We believe that this facility should be built-in at a low level within the system - in particular, declarative rules concerning consistency should not be aware of the existence of synonyms but should have the insertion done automatically - and that loose heuristics should allow the use of sub-component names to be inserted automatically.

4) Related Defects arising from mis-typing of names.

The provision of a simple, that is non-semantic, pattern-matcher for names could also be introduced. This pattern-matcher could use techniques already proven in Word Processing systems to provide the user with alternatives based on well understood algorithms - common two character transpositions, common misspellings, using plural rather than singular forms etc - to attempt to find the correct name. The extension of these techniques into sentence fragments - this is what data and process names typically consist of - would, we believe, prove fruitful.

5) Structural reconfiguration.

It is unlikely that the distribution of functionality remains static in large specifications and that defects will be detected during the production of a specification due to re-allocation of major portions of functionality. Whilst it would be possible for an Active Guidance system to detect and attempt to correct such a situation, we believe that a better approach is to provide the user with a facility somewhat similar in nature to aspects of the re-use tool. This facility would allow the user to re-distribute the functionality of an existing specification over a different Viewpoint Structure (or in other methods a Functional Decomposition). This work, however, needs further research since it is likely that reconfiguration will not produce a one-to-one mapping between the old and new configurations and the checking between the two representations will not be trivial.

6) Imbalance of Analysis.

An examination of the specification may reveal functional areas which are comparatively sparse or densely packed. This information could be used by the tool to suggest areas that require amalgamation or decomposition.

We believe that much more work should be done further analysing the structural content of a specification and that major benefits for, comparatively, little effort could be gained from the production of such a system.

No attempt during this research contract has been made to base an Active Guidance System on the semantics of the specification. We have considered the names of objects within the specification to be atomic entities and no attempt has been made to form links between specification elements except as dictated by the specifier.

2. Animation

2.1 Evaluation

As described, the animation facilities are mainly of use in the later phases of CORE, during transaction analysis (combined viewpoint modelling), constraints analysis (such as performance), and validation. Although some effort has been made to use the animator during these phases, it has been difficult for us to assess objectively the overall benefit of the approach in the case study. This

really requires use by a 'customer/user authority' with the necessary application expertise. Nevertheless, we did conduct an evaluation, and offer the comments below both as a set of 'biased' comments, and as a *guide to evaluation* of the animator, and urge application experts to conduct a more thorough evaluation.

2.1.1 Overall Evaluation

The main overall aspects of animation considered for evaluation were as follows:

- Overall validity of animation as a method of specification interpretation?
- Overall validity of animation as a method of specification validation?
- Overall validity of animation as a method of requirements elicitation?

Interpretation:

Transaction analysis involves the selection of 'interesting' transactions which are selected to perform "what if" enquiries i.e. to trace through causal chains, to ensure stimulus-response connections, or to set up test cases. (In pure CORE, this also includes the definition of transactions wholly contained in a single viewpoint, which are then combined at a higher level when moving up the viewpoint hierarchy.)

The main test was whether or not animation of transactions helped to *select* interesting transactions, and hence to display *interactions* and *effects* not obvious from the specification. Since Core specifications concentrate all the early effort in dividing the specification into different viewpoints (to aid elicitation and contain complexity), the unexpected interactions were expected at viewpoint boundaries. We therefore especially followed transactions which crossed viewpoint boundaries.

As mentioned above, it was difficult for us as application novices to make good transaction selections, and we also felt that evaluation of the interpretation facilities is very subjective. Nevertheless, the animator certainly helped us to understand and clarify the relationship between the different viewpoint specifications. This was particularly useful as the viewpoints had been specified by different members of the research project teams.

Validation:

The simpler form of validation is *inconsistency* testing. Those inconsistencies not detected by the Analyst Tool became obvious during animation. These were expected to be those that are "syntactically" acceptable but, for example, make use of a particular term in different ways in the particular domain. Despite the careful formulation of the case study, the animator did highlight many inconsistencies between the viewpoints, many of the form of misnaming, level mixing or mistyping (see comments 3 and 4 in the further work of guidance, 7.1.2).

The more complex validation is that which tests whether or not the specified behaviour is as required by the end user/client. This relies on testing for *acceptable specified behaviour*, and required input from an application expert. We strongly believe that the animator satisfies this role, but can not really test this on the case study. In addition, we believe that an extended animator would certainly aid validation of performance behaviour.

Elicitation:

Due to the absence of application experts, it has not been possible to elicit the extra information required to help and to evaluate its contribution to the overall specification. However, the intention was as follows:

Animation requires a more formal definition of the *actions*. These definitions should be compared with the natural language action descriptions to see whether or not they contribute to the elicited information.

In the extended animator, *performance* information was to be more carefully elicited and integrated (using attributes). Again, there is the need to test whether or not this information adds to the useful elicited information in the specification.

Acceptance criteria (see "transaction validation predicates" below) could provide useful information to indicate the purpose and constraints expected of each transaction. These should be evaluated to judge whether or not this information is useful for testing other runs of the same transaction, for describing the transactions to the designers, and for use as part of the acceptance tests of the proposed system.

2.1.2 Evaluation of Detailed Aspects

Many detailed aspects of our particular approach to animation were examined. A summary of the findings is given below:

- the use of mappings and functions for action definition:
This was both appropriate and simple, and was felt to be superior to other approaches such as detailed imperative pseudocode. Experience did lead to introduction of function *Q(message)* for user response during animation.
- transaction selection by animation:
With the introduction of the graphical interface, transaction definition followed by scenarios was more found to be as useful, but this could have been due to the difficulty we experienced as novices in identifying meaningful transactions.
In practice, it was found that the third transaction approach - that of finding a path between two specified actions - was also useful to suggest possible transactions. A rudimentary tool for finding the shortest path has been integrated into the animator and is used to supplement rather than replace the other selection mechanisms.
- action triggering by data arrival:
This appeared to be adequate for animation.
- graphical interface and interaction facilities:
This is essential for the use of animation as an interpretation and validation tool.
- data interpretation by functions:
This was implemented but found to be too complex and was therefore not used.
- animation feedback (correction) by note posting:
We have had insufficient experience to draw any conclusions.

2.2 Conclusions and Further Work

We are convinced that animation has a useful role to play in requirements analysis, and that the approach and tool support that we have described offer a simple yet powerful means to this end. The Animator has proved itself to be a very promising tool in aiding the clarification and validation of Core requirements.

We have tried to avoid requiring that data and action descriptions become too detailed, but, for execution, obviously require some indication of the processing that each action is required to perform. As mentioned, querying the user during animation has provided a simple yet convenient means for handling complex, changing and human-provided actions. We consider that the notion of transactions and animation provide a sound basis for interpretation and validation, and can be easily extended to permit attachment and interpretation of timing constraints. Further work is required to integrate and evaluate this approach.

Furthermore, we believe that the usefulness of the Animator is not restricted to this area of application (CORE requirements) or manner of use. For instance, the tool provided could usefully be applied to any similar form of data flow specification, such as is used in Jackson Systems Design Method (JSD) and Structured Analysis of SASD. Furthermore, as well as requirements validation, we believe that animation has a useful role to play in the elicitation process. This point is further elaborated below.

One of the most promising extensions still to be investigated is the use of **transaction validation predicates** to specify the acceptance criteria for a transaction. The user would be encouraged to specify the criteria to be used in accepting a transaction as satisfactory. This could be in a similar manner to that which is used for action definition: by acceptable mappings from inputs to outputs using functions, cases etc. In this manner, the user could add cases as required while trying out a number of scenarios, and/or various scenarios of a transaction could be validated by the animator itself. In addition, this approach would help to elicit the acceptance criteria of the specified system. With the prior integration of the performance facilities discussed above, timing and performance constraint validation could also be elicited and validated.

3. Reuse

3.1 Evaluation

The difficulties encountered in satisfactorily evaluating the conceptual basis of reuse and the TRUE tool reflect the overall difficulties encountered in accommodating reuse within CORE. Evaluating reuse depends upon having a large stock of base transactions as candidates for reuse, the ASET case study alone does not provide this. Experience in reuse of program code fragments suggests that the benefits of reuse only become fully evident where the reuse scheme is widespread. The commercial market in reusable AdaTM code, for example, has not taken off because of an insufficiently large base of available packages.

Our limited experience shows that the importance dominated low level pattern matching was the most effective strategy given transactions that have a direct likeness such as those within the ASET case study. The other strategies such as causal chain matching which proved successful on selected cases outside the ASET case study remain to be properly evaluated and this is clearly an area of further research interest. Also still to be evaluated is how a method should be structured to guide the activity of reuse.

3.2 Conclusions

Reuse in CORE appears at first sight a useful area to explore. Our conclusion, however, is that unless a method is constructed with reuse in mind, retro-fitting reuse is difficult and achieves little. TRUE provides access to all the facilities and techniques which provide support for reuse but, however full the technical armoury, reuse in CORE is misconceived. The TARA philosophy of basing tool investigations on a well understood method which is effective for method guidance and animation reaches its limits in studying reuse. Further research in this area should be concerned with developing methods that have reuse built into them but avoid the problems of "bottom-up"

specification.

During an interim review the observation was made that specifiers take previous specifications off a shelf to help in constructing a new specification. On reflection this practice - standard among requirements analysts - has actually very little to do with reuse and a lot to do with gaining domain knowledge. The domain knowledge may be important to the progress of the analysis but does not generally appear directly within the specification. Where it does appear it has been wholly reinterpreted and restructured prior to being deployed. Research in the use of domain knowledge in specification is an interesting and useful area for future research.

These conclusions may seem negative in tone, but blocking off unpromising avenues of research is as important as exploring new directions. On the positive side we have formulated a useful model of reuse and embedded it in a tool that allows the investigator to use the full range of significant reuse strategies. We have investigated the similarities between analogy and reuse, an area which has interesting research prospects. We have brought to attention new and powerful reuse strategies such as causal chain matching which merit further attention.

REFERENCES

- Ashley, K. (1984); "Reasoning by Analogy: A Survey of Selected AI Research with Implications for Legal Expert Systems"; (In) Walker, C. "Computer Power and Legal Reasoning"; West Pub Co. 1985; USA.
- P.Barth, S.Guthery, D.Barstow, 'The Stream Machine : A Dataflow Architecture for Realtime Applications', Proc. 8th Int. Conf. Software Engineering, IEEE Comp. Soc. Press, 1985.
- Bartlett, A.J., Cherrie, B.H., Lehman, M.M., MacLean, R.I. and Potts, C. The Role of Executable Metric Models in the Programming Process Rome Air Development Center, 1984.
- B.W.Boehm, Software Engineering Economics, Prentice Hall, 1982.
- T. De Marco, Structured analysis and system specification, Yourdon Press, 1978.
- T.Docker, G.Tate, 'Executable Data Flow Diagrams', in P.J. Brown and D.J. Barnes (eds.) Software Engineering '86, Peter Peregrinus, 1986
- Downes, V. & Goldsack, S. (1982); "Programming Embedded Systems in Ada"; Prentice-Hall; UK.
- J.Doyle, 'Truth Maintenance Systems for Problem Solving', MIT AI-TR-419, January 1978
- A.C.W. Finkelstein and C. Potts 'Structured Common Sense: The elicitation and formalization of system requirements' in P.J. Brown and D.J. Barnes (eds.) Software Engineering '86, Peter Peregrinus, 1986
- P.Hammond, M.Sergot, 'A PROLOG Shell for Logic Based Expert Systems', Proc. BCS Expert Systems Conference, 1983
- S.Hardy, A.Sloman, 'Poplog - a multi-purpose program development environment', University of Sussex, 1982
- M.A.Jackson, Principles of Program Design, Academic Press, 1975
- Kaehler, T. & Patterson, T. (1986); "A Taste of Smalltalk"; Norton, USA.
- A.Klausner, T.E.Konchan, 'Rapid prototyping and requirements specification using PDS', ACM SIGSOFT Software Engineering Notes, 7 (5), 1982.
- Kundt, K. (1984); "Pegasus — A Tool for the Acquisition and Resue of Software Designs; Proc. COMPSAC 84; pp288-293; IEEE Comp Soc Press.
- R.L.London, R.A.Duisberg, 'Animating programs using Smalltalk', IEEE Computer, 18 (8), August 1985.
- T.S.E. Maibaum, S. Khosla and P.Jeremaes 'A modal [action] logic for requirements specification' in P.J. Brown and D.J. Barnes (eds.) Software Engineering '86, Peter Peregrinus, 1986
- T.J. McCabe, et al, 'Structured Real-Time Analysis and Design', COMPSAC-85, IEEE, October 1985, pp40-51.

References

- J. McCarthy and P.J. Hayes 'Some philosophical problems from the standpoint of AI' in B. Meltzer and D. Michie (eds.) Machine Intelligence 4 Edinburgh Univ. Press, 1969.
- McDermid, J. & Ripken, K. (1983); "Life Cycle Support in the Ada Environment"; CUP; UK.
- G. Mullery, 'CORE - a method for controlled requirements specification', Proc. 4th Int. Conf. Software Engineering, IEEE Comp. Soc. Press, 1979.
- D.T.Ross, 'Structured Analysis (SA): a language for communicating ideas', IEEE Trans. Soft. Eng., SE-3 (1): 1977.
- B.A.Sheil, 'Power Tools for Programmers', Interactive Programming Environments, ed. B.I. Barstow, H.E. Shrobe, E. Sandewell, McGraw Hill, 1984
- E.Shortliffe, Computer-Based Medical Consultations : MYCIN, Elsevier, 1976
- M. Stephens, K. Whitehead, 'The Analyst - a workstation for analysis and design', Proc. 8th Int. Conf. Software Engineering, IEEE Comp. Soc. Press, 1985.
- W.R.Swartout, 'XPLAIN: A System for Creating and Explaining Expert Consulting Programs', Artificial Intelligence Journal. Vol 21, Number3, 1983.
- Systems Designers p.l.c. CORE manual 1986
- D.Teichroew, E.A.Hershey, 'PSL/PSA: a computer-aided technique for structured documentation and analysis of information processing systems', IEEE Trans. Soft. Eng., SE-3 (1), 1977.
- Winston, P. (1980); "Learning and Reasoning by Analogy"; Comm. ACM; 23,12, pp689-703.
- Yeh, R.; Mittermeir, R.; Rossopoulos, N. & Reed, J. (1984); "A Programming Environment Framework Based on Reuseability"; Proc. IEEE Int. Conf. on Data Engineering; pp 277-280; IEEE Comp Soc Press.
- P.Zave, 'An operational approach to requirements specification for embedded systems', IEEE Trans. on Software Engineering, SE-8 (3), 1982.

Appendix A

Formal Specification of Part of CORE

Viewpoint Structuring

predicates

post-viewpoint-structuring
-- viewpoint structuring has been done

actions

viewpoint-structuring

sorts

viewpoint-hierarchy
viewpoint-level
-- positive integer

constants

vpt-hierarchy	: viewpoint-hierarchy
curr-level	: viewpoint-level
1	: viewpoint-level

enabling conditions

-- Viewpoint structuring can only be done after problem structuring

post-problem-structuring -> per(viewpoint-structuring)

action definitions

-- After viewpoint structuring, there is a unique viewpoint hierarchy and -- the current viewpoint hierarchy level is 1. (i.e. below the root node).

(exists! vh)([viewpoint-structuring]
post-viewpoint-structuring &
vpt-hierarchy = vh &
curr-level = 1)

Tabular Collection

sorts

viewpoint
tabular-collection-form

predicates

post-tabular-collection : viewpoint
-- Tabular collection has been performed on the viewpoint

indirect : viewpoint
-- The viewpoint is an indirect viewpoint

post-tc-level : viewpoint-level
-- Tabular collection has been completed at that level of the
-- viewpoint hierarchy

functions

level-of : viewpoint -> viewpoint-level
-- The viewpoint is on that level of the viewpoint hierarchy

tcf : viewpoint -> tabular-collection-form
-- The tabular collection form produced by tabular collection
-- for that viewpoint

actions

tabular-collection : viewpoint
-- Do the tabular collection for that viewpoint

enabling conditions

-- Tabular collection for a direct viewpoint on the current level of the
-- viewpoint hierarchy is permissible (provided it has not already been
-- done) after viewpoint structuring.

post-viewpoint-structuring &
~post-tabular-collection(v) &
~indirect(v) &
level-of(v) = curr-level ->
per(tabular-collection(v))

action definitions

-- Tabular collection produces a tabular collection form for the given -
-- viewpoint.

(exists! v-tcf) (
[tabular-collection(v)]
tcf(v) = v-tcf &
post-tabular-collection)

assertions

-- Tabular collection is complete at a given level if tabular collection has -- been performed on all
direct viewpoints at that level.

(~exists v)(~indirect(v) &
~post-tabular-collection(v) ->
post-tc-level(level-of(v)))

Data structuring

predicates

doing-svm : viewpoint-level
-- Single viewpoint modelling is underway at that level

doing-ds : viewpoint-level
-- Data structuring is underway at that level

worth-ds : viewpoint-level
-- It is deemed worthwhile doing data structuring at that level

post-data-structuring : viewpoint
-- Data structuring has been performed on that viewpoint

post-ds-level : viewpoint-level
-- Data structuring has been performed at that level

actions

data-structuring : viewpoint

enabling conditions

-- Data structuring is permissible on a direct viewpoint at the current
-- level provided single viewpoint modelling is not already underway,
-- data structuring has not already been done, and data structuring at that
-- level is deemed worthwhile.
post-tc-level(curr-level) &
~doing-svm(curr-level) &
~post-ds-level(curr-level) &
worth-ds(curr-level) &
level-of(v, curr-level) &
~indirect(v) ->
per(data-structuring(v))

action definitions

[data-structuring(v)] post-data-structuring(v)

assertions

-- Where there are two different viewpoints at a given level, such that one
-- has been subjected to data structuring and the other, a direct
-- viewpoint, has not, data structuring is said to be underway.
(exists v1, v2) (
v1 ≠ v2 &
level-of(v1) = level-of(v2) = l &
post-data-structuring(v1) &
~indirect(v2) &
~post-data-structuring(v2) ->
doing-ds(level-of(v1))

-- If there is no such other direct viewpoint, data structuring has been
-- finished at that level.

```
(exists v1, ~ exists v2) (  
  v1 ≠ v2 &  
  level-of(v1) = level-of(v2) = 1 &  
  post-data-structuring(v1) &  
  ~indirect(v2) &  
  ~post-data-structuring(v2) ->  
    post-ds-level(level-of(v1))
```

Single viewpoint modelling

sorts

single-viewpoint-model

predicates

post-single-vm : viewpoint
-- Single viewpoint modelling has been performed on that
-- viewpoint

post-svm-level : viewpoint-level
-- Single viewpoint modelling has been completed at that level

doing-svm : viewpoint-level
-- Single viewpoint modelling is underway at that level

functions

svm : viewpoint -> single-viewpoint-model

actions

single-vm : viewpoint

enabling conditions

-- You can do single viewpoint modelling for a given viewpoint when
-- you have finished tabular collection at that level, and provided that
-- data structuring is not underway at that level.

```
level-of(v) = curr-level &  
post-tc-level(curr-level) &  
~doing-ds(curr-level) &  
~post-single-vm(v) ->  
  per(single-vm(v))
```

action definitions

```
-- Single viewpoint modelling produces a unique single viewpoint model
-- for the viewpoint in question.
(exists! v-svm) (
  [single-vm(v)]
  svm(v) = v-svm &
  post-single-vm(v))
```

assertions

```
-- Where there are two different viewpoints at a given level, such that one
-- has been subjected to single viewpoint modelling and the other, a direct
-- viewpoint, has not, single viewpoint modelling is said to be underway.
```

```
(exists v1, v2) (
  v1 ≠ v2 &
  level-of(v1) = level-of(v2) = 1 &
  post-single-vm(v1) &
  ~indirect(v2) &
  ~post-single-vm(v2) ->
  doing-svm(level-of(v1))
```

```
-- If there is no such other direct viewpoint, single viewpoint modelling
-- has been finished at that level.
```

```
(exists v1, ~ exists v2) (
  v1 ≠ v2 &
  level-of(v1) = level-of(v2) = 1 &
  post-single-vm(v1) &
  ~indirect(v2) &
  ~post-single-vm(v2) ->
  post-svm-level(level-of(v1))
```

Combined Viewpoint Modelling

sorts

transaction

predicates

interesting	:	transaction
post-combined-vm	:	transaction
action-in-transaction	:	action X transaction

functions:

performed-by : action -> viewpoint

actions:

combined-vm : transaction

enabling conditions:

- Those transactions that involve critical performance or reliability
- aspects of the proposed system (interesting transactions) need to be
- analysed. This can only occur if all viewpoints involved in the
- transaction have been subjected to single viewpoint modelling and data
- structuring has either finished or is not worthwhile for that viewpoint.

```
(exists t) (  
  interesting(t) &  
  (for all a) (  
    action-in-transaction(a, t) &  
    performed-by(a) = v  
    post-single-vm(v) &  
    ~worth-ds(v)  
    -> per(combined-vm(t))))
```

```
(exists t) (  
  interesting(t) &  
  (for all a) (  
    action-in-transaction(a, t) &  
    performed-by(a) = v  
    post-single-vm(v) &  
    worth-ds(v) &  
    post-data-structuring(v)  
    -> per(combined-vm(t))))
```

action definitions:

[combined-vm(t)] post-combined-vm(t)

Appendix B

Performance Animation in CORE

1. Terminology

The following definitions of terms apply.

- Component.** A component is a control flow, a data flow or an action used in an SVM or a CVM.
- Path.** A path is an ordered sequence of components. Note that a transaction comprises one or more paths.
- Start-point.** A start-point is the first component on a path.
- End-point.** An end-point is the last component on a path.
- Performance characteristic.** A performance characteristic is the semantics of a time-related aspect of behaviour of a component.
- Performance attribute.** A performance attribute is a CORE/Analyst attribute representing a performance characteristic of a component.

2. Introduction

For most applications it is necessary but not sufficient that they perform correctly - they must also carry out their functions on an acceptable timescale. For applications such as interactive information systems occasional failure to meet response criteria can be annoying but is not serious, while in other applications, such as safety monitoring systems or intensive care patient monitoring systems, failure to respond in time could be disastrous.

'Performance' loosely refers to timeliness of response, and 'performance modelling' refers to representing time aspects of behaviour of the target system. It is shown how extensions to CORE can be made to model performance in a way that is in sympathy with current CORE concepts. Section 3 presents the objectives of performance modelling and how it can be carried out. Section 4 considers the performance information that can be handled and how it can be incorporated into the 7 stages of CORE. Section 5 describes how performance can be represented, and section 6 describes aids for performance analysis and tool support, particularly extensions to the Animator. Appendix 1 is the User Guide for the Animator extensions to handle performance modelling.

3. Objectives

The aim of the performance modelling extension is to increase the usefulness of CORE by capturing performance information along with the rest of the requirements, and representing performance in the models of the system that is developed during CORE analysis. The main benefits are the following.

- a) The customer will be able to verify that performance criteria have been correctly considered.
- b) The consequences of performance criteria may lead the customer to reconsider the

requirements, for example to avoid excessive delays by changing the path of some of the data through the application system.

By taking account of overall performance requirements it should be possible to specify the performance of individual actions and data paths. This information can be checked by the analyst to ensure that detailed performance requirements collectively lead to acceptable system performance, and used by the designer as a guide to how parts of the system should be implemented.

4. Incorporating performance information in CORE

The extension is designed to handle two types of performance information: requirements given by the customer authority and assumptions made by the analyst in order to check that the performance requirements can be met.

REQUIREMENTS

The customer authority views performance from the point of view of the overall system as seen from outside. Performance will typically be expressed in two ways.

- a) Delay or response time, for example 'the delay between a patient's heart rate going outside defined limits and the generation of an alarm must not exceed ten seconds.
- b) Throughput or traffic, for example 'the system must have a one-minute cycle time, during which data must be collected from the 250 sensors and processed'.

ASSUMPTIONS

Assumptions are working hypotheses made by the analyst rather than the customer authority, and are made in order to complete the analysis. For example, in a particular SVM some delays may be known from the requirements or the constraints, but in order to check an overall delay time the delays in some actions must be assumed. Such assumptions will have an impact on other SVMs that contain the same components, so possible conflicts must be checked.

A strict distinction must be made between assumptions and the other sources of performance information, since assumptions are made for the convenience of the analyst and do not reflect the view of the customer authority.

AFFECT ON THE 7 STAGES OF CORE

The impact of the performance modelling extension on the 7 stages of core are as follows.

- 1) Problem Definition. Some global aspects of performance may be quantified at this stage. For example the purpose of the system may be to process up to 800 blood samples each day, or to raise an alarm within one minute of the level of carbon dioxide in a coal mine exceeding a given threshold by 10%.
- 2) Viewpoint Structuring. It is unlikely that performance information will be captured in this stage, which is concerned with organisation rather than technical matters. The main purpose of Viewpoint Structuring is to establish the structure of the system, and to identify the customer authorities who will provide information about the different parts.
- 3) Tabular Collection. Information for the Tabular Collection is obtained from each Viewpoint Authority, and it is at this stage that information about performance requirements is likely to be available. Since CORE does not define a formal way of recording performance requirements in the

TCF, additional attributes will be needed for this purpose.

4) **Data Structuring.** Normally it is not very meaningful to associate performance with data structures. There may indeed be a side effect on performance due to the number of iterations of a data structure or its size, but the only relevant information in a Data Structure Diagram is concerned with order of derivation rather than time delays.

5) **Single Viewpoint Modelling.** This is the most appropriate stage in which to consider the performance of actions and data paths. At the stage of deriving an SVM from a TCF information is added about control flows and time ordering, so the effect of the performance of individual components on overall system performance can begin to be seen.

It will be necessary to extract performance requirements from information collected with the TCF in order to derive aspects of performance associated with SVM components. If this information can be recorded in a formal way tools could be provided to aid analysis, for example to identify critical paths through the SVM for control and data flows that are the subject of performance requirements.

6) **Combined Viewpoint Modelling.** This stage of CORE probably offers the most scope for performance modelling, since the purpose of a CVM is to show in one diagram a transaction complete from end to end. A CVM should be produced for every transaction for which performance criteria are available.

Analysis of the CVMs will show all the components that have an influence on the required performance, and will reveal which of those components occur in more than one CVM. At this stage assumptions will be made for components that do not have associated performance details and are on paths for which the overall required performance is specified.

Cross checking will be required when a component that exists in more than one CVM is modified, to ensure that the modification does not invalidate any of the performance requirements for the overall system. At this stage analysis will be fairly simple, for example to identify critical paths and to ensure that in all cases the sum of individual delays comprising a transaction is equal to the given total delay. Comprehensive analysis techniques, for example based on queueing theory, require information on parallel processing and scheduling algorithms which is not available before the design stage.

7) **Constraints Analysis.** In addition to requirements the customer may impose constraints that have a bearing on performance, such as 'the furnace takes one hour to come up to temperature', or 'the datalink between the branch office and the head office has a maximum data rate of 2,400 baud'. Such information can be used to determine delays in actions and dataflows, and may imply performance requirements.

5. Representations of performance

The customer's view of performance must be represented appropriately at each of the stages above. For example the customer's concept 'response time for a query' may be represented as a series of delays in actions and data flows in CORE, and may be modelled as queues, priorities and scheduling algorithms at lower levels of design. Performance is represented as a series of delays along paths of interest, taking into account the effect of data rates and values, and resulting queue depths.

At the level of SVMs and CVMs performance information can be used to identify the start-point and end-point of paths of interest, and the corresponding SVMs or CVMs can then be derived by combining all paths that have common start and end points.

At the component level performance characteristics can always be represented as delays. Higher-level concepts such as throughput or traffic can be represented as iterations of delays, ie. as the product of the component delay time and the number of repetitions for a single transaction.

There are typically two components of delay: intrinsic delays (usually fixed) due to the nature of the component or its implementation, and delays depending on the dynamics of the system, such as data values or the amount of data involved. The delays associated with the different components are the following.

a) Channels. A channel can have a delay (access time) due to its implementation, for example if it is mapped onto a physical communication system. The delay may also depend on the volume of data involved, and so may not be constant.

A channel is logically a queue, with a producer action feeding data in and a consumer action taking data out, so it can introduce a delay with respect to an item of data if there are one or more data items ahead of it in the queue. If T_c is the time for the consumer to process one item of data, and N items have been produced since the one of interest, then the time delay before the data item of interest is available is $N * T_c$. The value of N will depend on the time pattern of inputs to the producer, and the time taken by the producer and the consumer to process each item of data.

b) Pools. A pool can have a delay due to the access time or the update time. As for channels above the delay may depend on the volume of data involved.

c) Control flows. A control flow usually represents synchronisation of an action with an event, which may be internally generated by the system or may be an external event such as a hardware signal or a time (for example 'repeat every 10 seconds'). A control flow may therefore impose an additional delay on the starting time of an action, which must be calculated and taken into account.

d) Actions. The delay in an action can be due to processing time, which will depend on the algorithm and processor type used in the implementation, and may also depend on the data as above. Actions may also have an intrinsic delay, for example the time for a furnace to reach working temperature or the time to derive the average of 10 heart beats.

6. Aids for performance analysis

The analyst can use either an analytical (mathematical) model or a simulation model to gain an insight into how the proposed system can meet performance requirements.

The analytical model is a static model that consists of a network of nodes (actions, dataflows and control flows). Performance will be represented as expressions, in terms of delays at nodes, for delays along paths of interest.

The simulation model is a dynamic, operating replica of the proposed system. Each element needs to contain information about the behaviour being represented, as in the Animator.

Three tools are suggested: a path analyser and CVM generator, and a timing expression generator to support analytical modelling; and extensions to the Animator to support simulation modelling. These tools are outlined below.

6.1 Path analyser and CVM generator

The purpose of this tool is to identify a transaction related to a performance requirement. Using a start-point and an end-point the tool would find all paths that link the two and generate a CVM (or SVM) containing all the actions on the paths found.

The method of use is to take each performance requirement in turn and identify the start-point and end-point of the corresponding transaction. This information will be provided to the tool via the user interface.

If the performance requirement is captured in a TCF the start-point and end-point will probably be the names of an input and output which can be used directly, and the transaction will be contained within the SVM for the viewpoint. If the performance requirement is more general it may be necessary for the analyst to decide which start-point and end-point to use, and they may be in different viewpoints.

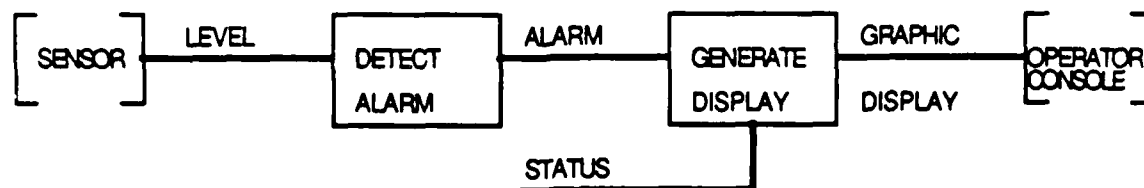
Start-points and end-points could be the environment viewpoint, control or data flows, or actions. An example of an action start-point is 'from when heart performance analysis detects an abnormality'.

As well as scanning forwards along a path, the tool will identify all the inputs to each action in the derived CVM and backtrack until an input to the viewpoint containing the action is reached. The reason is that other inputs (ie those not directly on the identified path) may influence performance. For example, consider the action to detect heart-rate abnormality in the patient monitoring system. This action will certainly have heart rate as an input, but it might need some other parameter, such as blood pressure, in order to determine whether the heart rate is within normal limits. Thus, if there is a delay in obtaining blood pressure, the action on the critical path to detect abnormality will also be delayed.

6.2 Timing expression generator

This tool creates an analytical model of a transaction of interest. The expression derived could be a simple sum of delays of individual components along a path, or it could take account of statistical timing information.

As a simple example consider the case of an alarm signal, for which the transaction might start with a sensor to detect the external signal, and follow a series of flows and actions as shown below:



Suppose we know from the constraints that the sensor takes two seconds to read the input level, and it takes two seconds to generate a display. We might also know that the data flow "level" is a pool which adds negligible delay. This leaves the time taken by the action "detect alarm" and the data flows "alarm" and "graphic display" unknown, so let us refer to these as T_d , T_a and T_g respectively. We can now write an expression in the simplest case for the time from the input signal going into an alarm state to the alarm appearing on the operator's console as:

$$2 + T_d + T_a + 2 + T_g$$

If we now take into account the additional dataflow "status" going into the action "generate display", in the worst case this data could have arrived immediately before the dataflow "alarm", which must therefore wait two seconds while "status" is being processed.

The expression could take account of iterative actions representing traffic, and multiple paths between the end points of the transaction when the critical path cannot be identified.

For the main input the tool will use Prolog assertions derived from the BDEX database as for the Animator. The output could similarly be a set of assertions that could be used directly by the extended Animator (and could be translated the other way for storage in the BDEX database), or it could be a text file giving information to help the user generate the CVM manually.

The tool could have an interactive phase to prompt the user for missing information such as timing attributes or number of iterations of an action during the transaction.

There will usually be insufficient information to decide which path has the longest delay between the start-point and the end-point of a transaction, so an expression will be generated for all paths found. the expression will be used at the design stage, when more information will be available to identify critical paths and components.

6.3. *Animator extensions*

The Animator could be extended to provide a simulation model in which the performance of a system is represented in terms of delays and queues. The aim is to enable the user to validate a system taking time into account.

The Animator currently uses a model based on how actions transform inputs to outputs and the dataflows between actions, and allows the user to check the logical behaviour of a transaction. When a transaction has been defined and checked, the extensions allow the user to enter assumed time delays for actions and dataflows, the values and frequencies of external inputs to the transaction, and then execute the model to verify whether response time and throughput requirements can be met with the assumed values.

The effect of delays and frequency information is the following:

- 1) when an action becomes executable its outputs are not generated until the simulated delay time has elapsed;
- 2) when a data item is put onto a channel, it is not made available at the output queue until the simulated delay time has elapsed;
- 3) external inputs to the transaction are made available at a rate synchronised to simulated time.

When the extended Animator is run, additional output is produced showing simulated elapsed time, the depths of queues on channels, and showing for each action whether it is waiting to run, or running and waiting for the delay to expire before generating its output data.

Output can be in one of three forms. The simplest form, which is all that will be implemented initially, will be textual information in the default output window. This output can be redirected to a file for later printing and analysis. A more user-friendly output for some cases is to provide a table on the screen giving the same information, but being updated continually during the progress of an animation run. The third form comprises a window for each component of the transaction being animated, in which the information is continually updated as for the tabular output described above. In any case, the user can select which components of the transaction are to be displayed in the output.

The information in the output is sufficient for the following to be derived:

- 1) response time, which is the simulated elapsed time between data being available at one point

in the transaction and the required response;

- 2) maximum throughput, which is the maximum data rate that can be generated at an external input without unacceptable queue buildups within the transaction;
- 3) location of bottlenecks, which are identified by the channel on which queue buildups occur when the maximum throughput is exceeded.

A simplifying assumption is made that pools have no delays, so no modification to the way pools are handled is required. If it appears later to be necessary, pools could be implemented to have an update and an access delay.

The description below relates to an extension that is not too ambitious, but nonetheless should demonstrate the value of performance modelling. The main simplifications are to produce text-only output and to specify all timing information interactively to the Animator. When the principles have been proved, further extensions could be defined:

- 1) to capture some timing information earlier in the CORE analysis, along with the rest of the system requirements, in the form of attributes.
- 2) to display the performance model graphically.

THE USER INTERFACE

To avoid too many problems at once for the user, performance modelling should be carried out only on a valid specification. Therefore a transaction should be defined and validated using current Animator facilities before any of the extensions defined here are used.

Starting from a validated transaction, the following steps are necessary to carry out performance modelling:

- 1) define external inputs to the transaction,
- 2) define action delays,
- 3) define channel delays,
- 4) link external inputs to actions or dataflows that need them,
- 5) set up the run parameters, such as break points,
- 6) run the model.

Extensions needed to menus and windows for the above steps are the following:

- 1) External inputs are defined in terms of "generators", which describe the values and timing of data that will be used as inputs to the transaction. A new window DESCRIBE GENERATOR in the ACTION menu is needed that has two fields as described below.

The first field defines the values to be produced by the generator, which may be given as a list of values, the name of a separately-defined list of values, or a named function. Lists of values will be used for example when the effect of specific values needs to be tested, such as the list of numeric values (1, 2, 4, 4, 9, 10, 11). Functions could be used for example to produce a sequence of "true" and "false".

The second field defines the timing in terms of two parameters. The first parameter defines

either a constant or variable time interval between generated values. For constant timing, the field is a numeric value or a numeric expression giving the number of time units (eg seconds or milliseconds) between successive values being available. For variable timing the field can contain the name of a function, for example to define random timing over a given range. Initially only a numeric value or expression will be allowed, defining a constant time interval. The second parameter gives the units of time: "h", "m", "s" or "ms" representing hours, minutes, seconds and milliseconds respectively.

A corresponding window DELETE GENERATOR is needed in action menu to allow the cancellation of generators.

Generators are defined independently from where they are used so that one generator can supply data to more than one input of a transaction, or different generators can be 'connected' to an input for different Animator runs.

2) Action delays are defined in the DESCRIBE ACTION window from the ACTION menu. A new field is needed that can accept two parameters, a delay time and the units of time as described for generator in 1) above. A function could be used for the delay time to express for example that the delay depends on the values of one or more inputs to the action. An example is an action that has a numeric input and a logic input representing 'true' or 'false'. When the logic input is 'true' the time to process the numeric input is 10 time units, but when the logic input is 'false' the time to process the numeric input is 2 time units.

An action delay may depend on the criteria for the action to execute, so the values put into the field could be different for each input specification given in a DESCRIBE ACTION window for a particular action.

3) A new window DESCRIBE DATAFLOW is needed, to allow the user to supply channel delay information in a similar way to action delay information described in 2) above. Since the top level menu area is already rather crowded, this new window could be accessed from the ACTION menu for now.

4) External inputs will be linked to actions by selecting LINK GENERATOR from the ANIMATE menu and then selecting in turn all actions that have an external input. The input field will accept the name of a generator defined in 1) above.

5) Run parameters will be set up in a new window CONTROL PARAMS accessed from the REPLAY menu. This window will have five fields to define:

- the duration of run in simulated time, expressed as a numeric value and the units of time as described in 1) above
- speed (fast, medium or slow), which may be useful to slow down the animation for visualisation purposes
- the interval in simulated time units between reports, and a list of components (generators, actions and channels for which information is to be presented
- the form of reports (textual in the default output, tabular or windows, as described in 1. above
- breakpoints (pauses to allow the user to review the state of the animation):
 - each time a given number of time units has elapsed
 - when a given action produces output
 - when a given queue reaches a given depth

6) The model is run using a new item GO WITH TIME in the REPLAY menu. The switches

GO-GO have the same meaning as before, but if the STEP switch is set, the Animator will run one cycle at a time instead of one action at a time as at present.

IMPLEMENTATION

The overall operation of the Animator is cyclic, where each cycle corresponds to the operations carried out when the next significant event occurs. A significant event is defined as the delay time having expired for an action, a dataflow or an external input, so that fresh data is available at some point in the transaction.

The operations carried out during a cycle depend on the status of the external inputs, channels and actions in the transaction. Status values are defined below.

Generators (external inputs)

- 1) IDLE, which means that no data is available.
- 2) ACTIVE, which means that the simulated elapse time to generating the next data item has expired and the data is available. The time is defined in the generator definition, for example one data item to be made available every 15 seconds.

Channels

- 1) IDLE, which means that no data items put onto the channel are waiting to be transferred to the output queue.
- 2) BUSY, which means that one or more data items put onto the channel are being delayed by the given channel-delay and are not yet available at the output.

Actions

- 1) IDLE, which means the action cannot execute because the conditions for execution (phase 4 below) are not met.
- 2) RUNNING, which means that the inputs have been consumed and the outputs will be generated when the given delay time has expired.

Each generator, action and item in a channel has two associated parameters as follows.

- 1) DELAY_TIME represented as a number of time units.
- 2) TIME_LEFT, which is the number of time units left before the delay time expires. In the case of a generator the start of the delay time corresponds to when the last data value became available, in the case of an action, the start of the delay time corresponds to the last change of status from IDLE to RUNNING, and in the case for an item in a channel the start of the delay time corresponds to when the item was put onto the channel.

A channel has a FIFO queue to hold the data available at its output. The FIFO has an associated parameter ITEM_COUNT giving the queue depth.

Animator cycles

The operations of the extended Animator for each cycle are carried out in seven phases as follows.

- 1) Determine the shortest `TIME_LEFT` for any generator, channel or action not in the 'ideal' state. This value is the elapsed time since the previous cycle.
- 2) Decrement the `TIME_LEFT` for all generators, items in busy channels and running actions by the elapsed time since the previous cycle determined in 1) above.
- 3) If the `TIME_LEFT` for any item delayed in a `BUSY` channel has reached zero, put the item onto the output queue and increment the `ITEM_COUNT`.
- 4) If the `TIME_LEFT` for any generator has reached zero, make the next value available at all inputs to which it is connected and reset the `TIME_LEFT` to the next value as given in the generator definition.
- 5) For all `RUNING` actions whose `TIME_LEFT` has reached zero, execute the action, generate the output and set the status to `IDLE`.
- 6) Evaluate the criteria for execution of all actions in the `IDLE` state and, for those which are executable, set the status to `RUNNING` and the `TIME_LEFT` to the value corresponding to the criterion that was satisfied. Decrement the `ITEM_COUNT` for all channels for which a data item has been consumed.
- 7) Backtrack to step 6 until no more actions are able to execute.

When evaluating criteria for execution of an action, a data item is assumed to be available at the output queue of a channel if the `ITEM_COUNT` is greater than zero.

OUTPUT FORMAT

Performance model output is generated according to the criteria given in `CONTROL PARAMS` (user interface step 5 above), or when a break point is reached. The information available is the following.

- 1) The reason for the report (eg. 'break' or 'elapsed time').
- 2) The total simulated elapsed time since the beginning of the current animation run.
- 3) For each external input, the number of data items generated but not consumed.
- 4) For each channel, the status (`BUSY` or `IDLE`), the number of items delayed in the channel if it is busy, and the parameters `DELAY_TIME` and `ITEM_COUNT`.
- 5) For each action, the status (`RUNNING` or `IDLE`) and the value of the parameters `DELAY_TIME` and `TIME_LEFT`.

The following are examples of the information displayed in individual windows for the third type of output.

General information

BREAK: Output from action Alter Safety Limits
TOTAL TIME:: 327 s

External inputs

INPUT NAME :	current limits
VALUES NOT CONSUMED:	23
DATA AVAILABLE:	no
LAST VALUE CONSUMED:	99

If DATA AVAILABLE = yes, then the last field will be

CURRENT DATA VALUE;

Channels

CHANNEL NAME:	new limit
STATUS:	idle
DELAY:	1
TIME LEFT:	0
DEPTH:	2
CURRENT VALUE:	88

TIME LEFT refers to the item that will next be placed in the FIFO queue. DEPTH is the ITEM_COUNT, ie the number of data items in the FIFO queue waiting to be consumed. Current value refers to the value available at the head of the FIFO queue.

Pools

POOL NAME:	graph
CURRENT VALUE:	32

Pool information is given for completeness, but since pools are defined as having no delay only the current value is of interest.

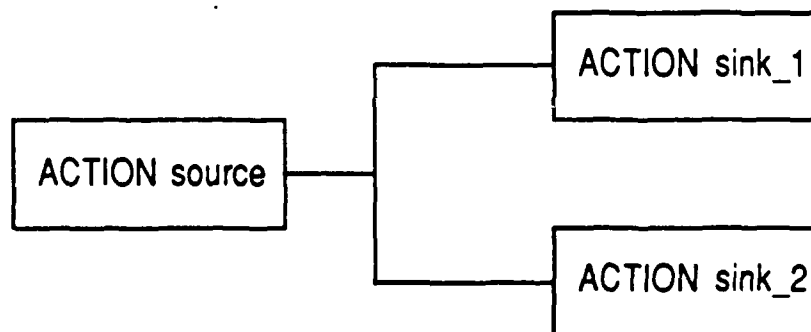
Actions

ACTION NAME:	alter safety limits
STATUS:	active
DELAY:	5
TIME LEFT:	3

Input and output values are not given here since they are available in the appropriate External Input, Channel and Pool windows.

IMPLEMENTATION NOTES

Channels are allowed only one output connection since 'fan out', ie multiple destinations, cause problems about when the item currently at the head of the queue is consumed. Consider the simple case below in which action sink_1 runs more frequently than action sink-2



Suppose a new item appears at the head of the queue and is read by action sink_1. The item should not be removed since action sink_2 has not had a chance to read it yet, but what happens if action sink_1 again becomes ready to read the next item from the channel? If it reads the same item again this is clearly not the intended semantics, but if the top item is discarded so that sink_2 can read the next then the item is lost to action sink_2.

The alternative is to have two FIFO queues, one for each of the actions sink_1 and sink_2, effectively making two channels. For the first version of the extended Animator it will be required to specify the two channels explicitly in such a case, since this simplifies implementation and renders the output display more explicit and understandable.

Appendix C

Demonstration Notes

C.1 Animator Demonstration Script

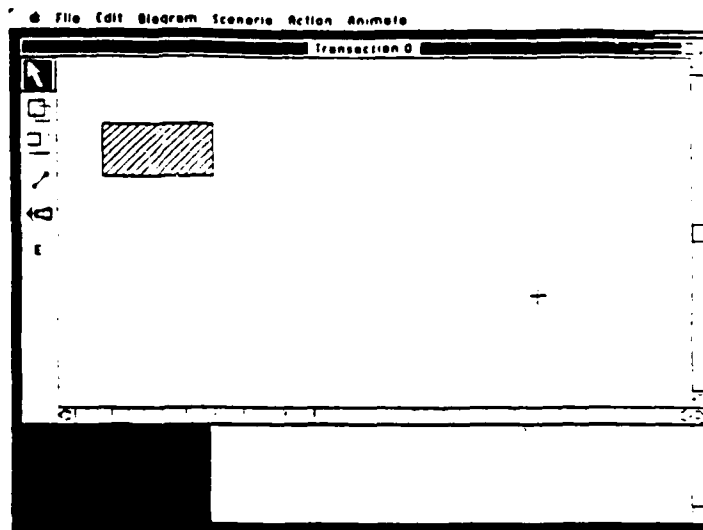
C.2 Method Guidance Demonstration

Animator Demonstration Script

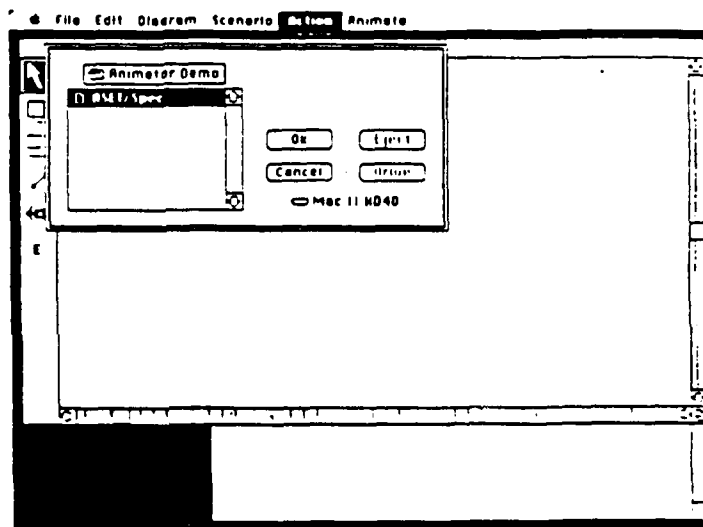
Keng T. Ng
Department of Computing
Imperial College of Science & Technology
180 Queen's Gate
London SW7 2BZ

Telephone 01 589 5111
Telex 261503

Start MacProlog by double-clicking on the LPA MacProlog icon. When the MacProlog menu bar appears, choose *Load...* from the File menu. Select *Animator 3.1* and then click OK to load the Animator program. After a brief delay, the Animator desktop will appear as on the right.

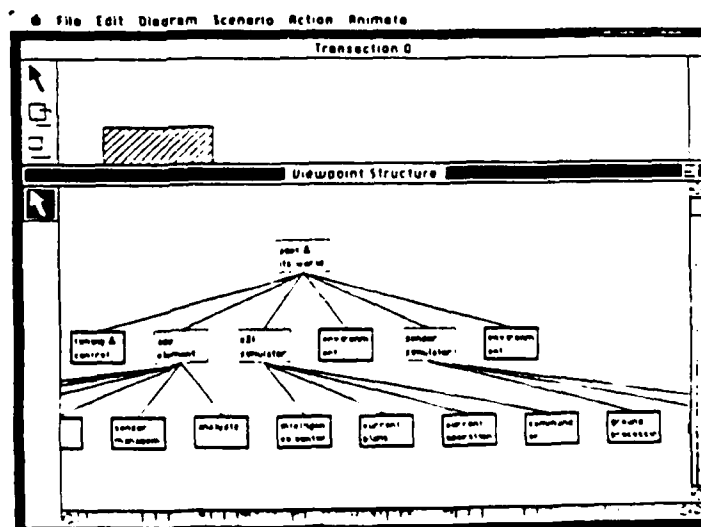


The next thing to do is to load the ASET specification. Choose *Load specification...* from the Action menu. In the dialog box presented, click on *ASET:Spec* and then the OK button.



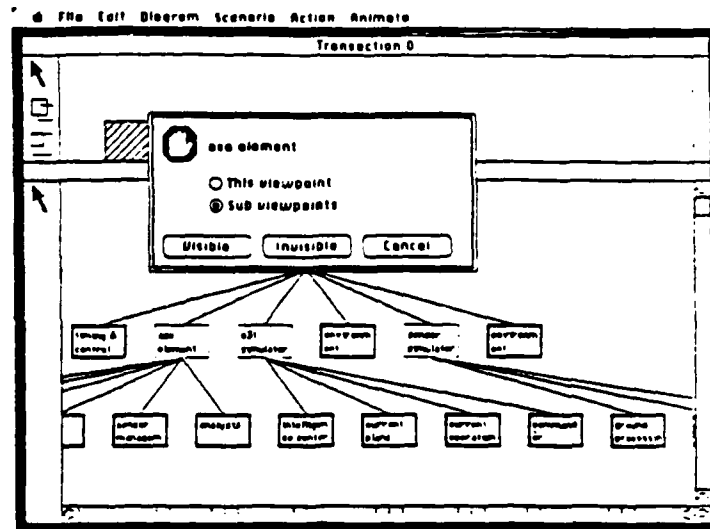
By default, the Animator allows only actions in the leaf viewpoints to be selected for execution. To change this, choose *Show viewpoints* from the Animate menu. This creates a new window showing the viewpoint hierarchy. Viewpoints drawn in bold are visible viewpoints, i.e. those whose actions are selectable.

Our example consists only of actions belonging to viewpoints in the first level of the viewpoint hierarchy. We will make these viewpoints visible and the rest invisible.



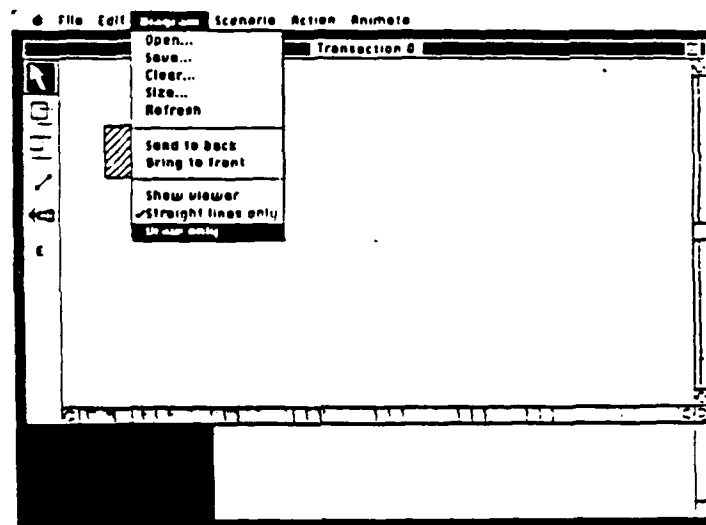
Hold down the Option key and click on the viewpoint *ase element*. This will generate a dialog box as shown in the diagram. Click on the *Invisible* button. This makes this viewpoint visible and all its sub-viewpoints invisible. Repeat this for the viewpoints *c3i simulator* and *sensor simulator*.

Choose *Hide viewpoints* from the Animate menu. From now on when we want to execute an action, only actions belonging to the visible viewpoints are presented.

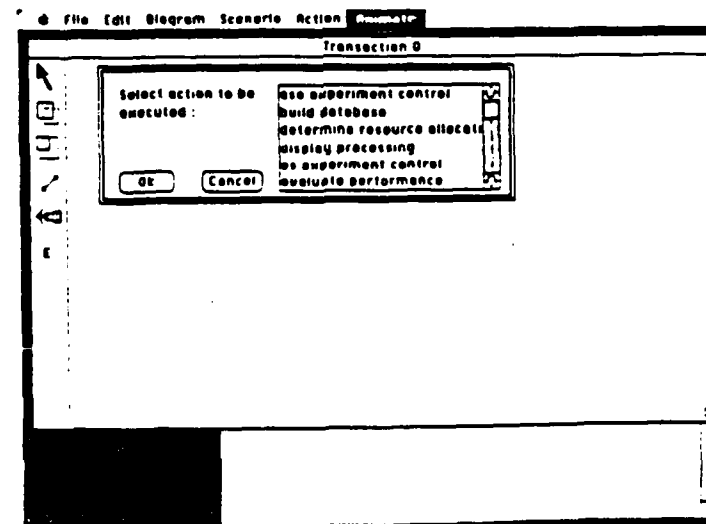


A transaction is created by selecting, one by one, the individual actions that make up the transaction. By default, an action will be executed after it is selected. For this demo however, we will use the alternative strategy of animation i.e. the *Draw Only* strategy.

Choose *Draw only* from the diagram menu. A tick now appears to the left of this menu item. This means that from now on any actions selected will be drawn in the transaction window but not executed.

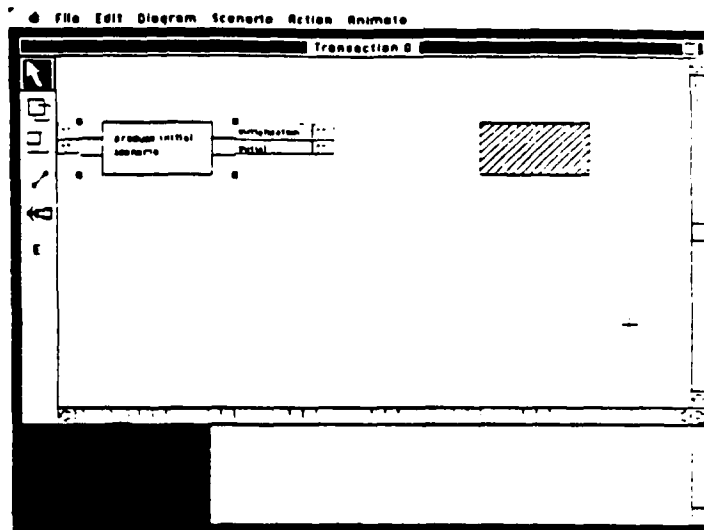


To select the first action in a transaction, choose *Any action* from the Animate menu. This will bring up an alphabetically-sorted list of all the actions performed by the visible viewpoints.

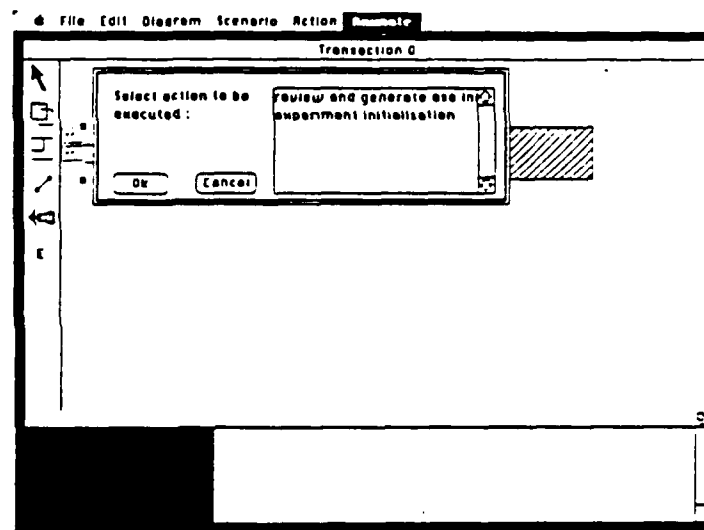


Select *produce initial scenario* and then click on the OK button. The action box of *produce initial scenario* will appear in the transaction window, together with its input and output data names and values.

At this point, we may want to try out some of the graphical tools (eg. enlarging the action box to spread out the input and output stubs, moving and enlarging data names etc.).

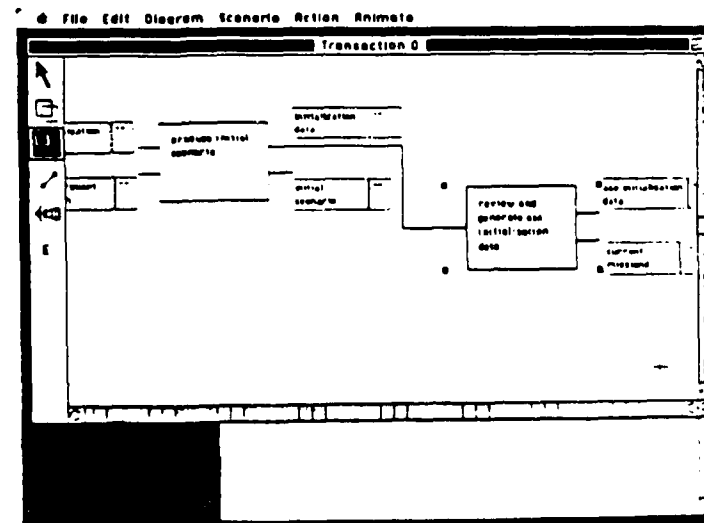


Choose *Next action* from the Animate menu. We will get a scrolling list displaying all visible actions which receive inputs from *produce initial scenario*. In this case there are only two, namely *experiment initialisation* and *review and generate ase initialisation*.

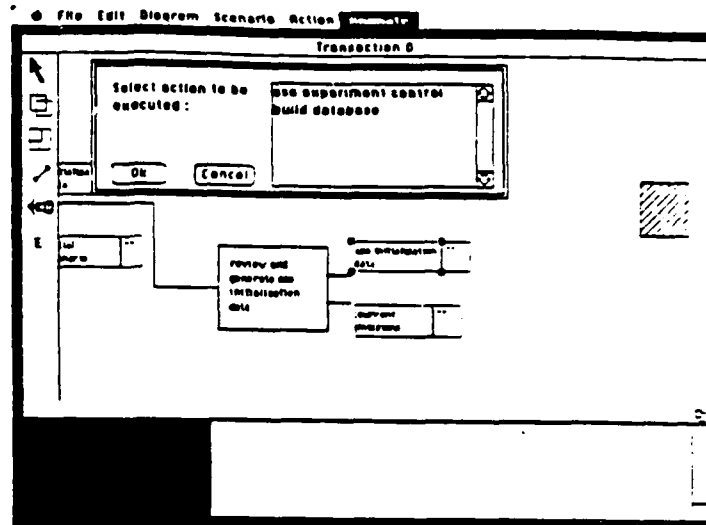


Select *review and generate ase initialisation* and click OK. It will be added to the transaction window. The dataflow *initialisation data* will be automatically drawn between the two actions in the transaction.

At this point we may want to show that the dataflow remains connected when we move any of the actions. Also, we can show how the line tool can be used to reshape the dataflow. The screen dump on the right shows the transaction diagram after it has been tidied up.

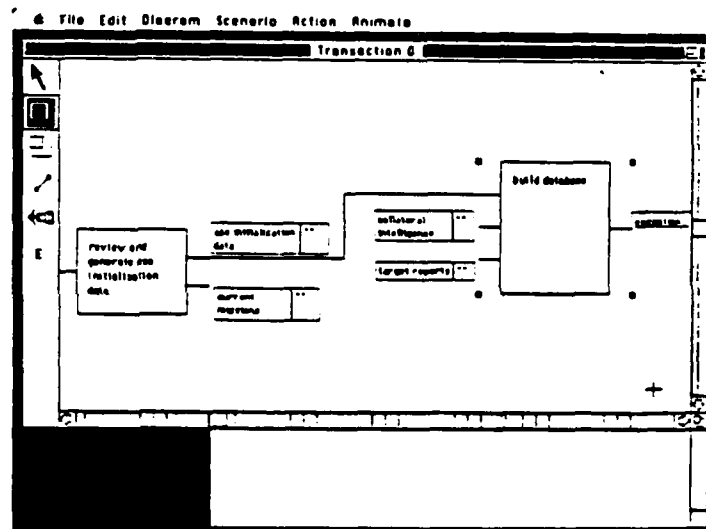


The Animator allows us to follow a particular dataflow in a transaction : click on the arrow tool in the transaction window, then click on the data name *ase initialisation data* which is the 1st output of *review and generate ase initialisation*. Now choose *Next action* from the Animate menu. The actions displayed in the scrolling list are those that receive *ase initialisation data* as input.



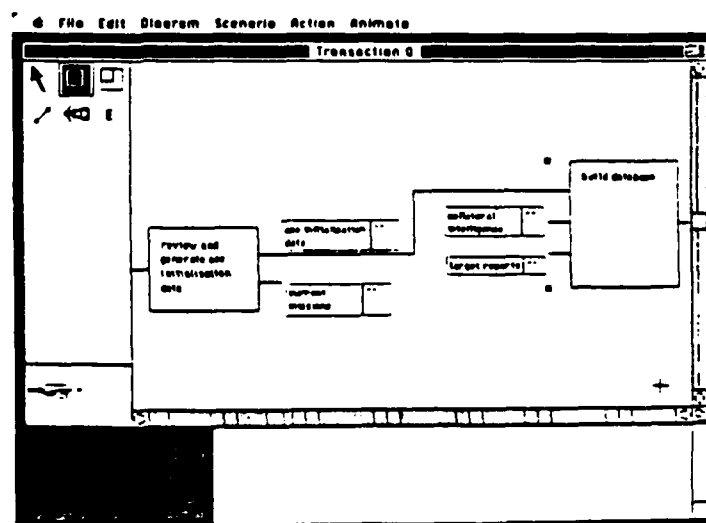
Select *build database* and click OK. *Build database* will be drawn in the transaction window.

At this point we may want to tidy up the diagram so that all data names and values are clearly visible.



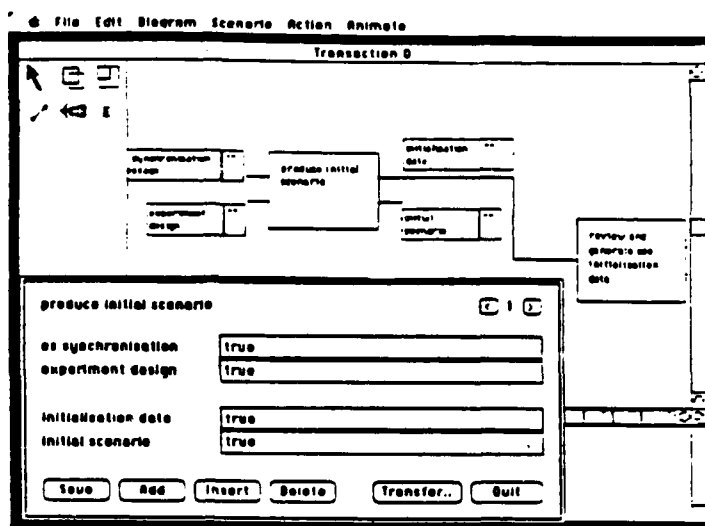
To get an overall view of the diagram that has been drawn, choose *Show viewer* from the Diagram menu. The tool palette will be increased to 3-tool wide to accommodate the viewer. We can vary the size of the tool palette (and hence the viewer) by dragging the line dividing the palette and the viewing pane.

Use the viewer as a quick way of scrolling the diagram.



For this demo, we will first define the actions in our transaction before we execute them. We will define them such that all their outputs will have the value *true* if all their inputs are *true*; *false* otherwise.

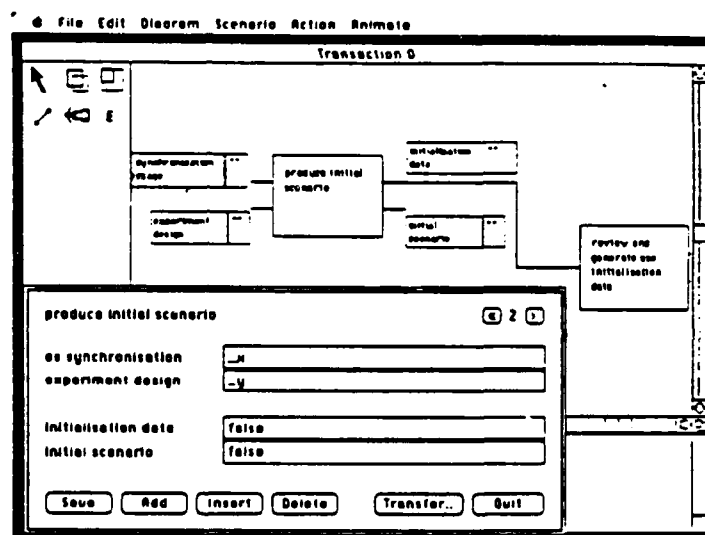
Click on the *Execute* tool in the tool palette of the transaction window. Then hold down the option key and click on the action box of *produce initial scenario*. This will bring up the definition window of this action. Enter *true* in the text areas of both its inputs and outputs, then click the Save button to save this definition.



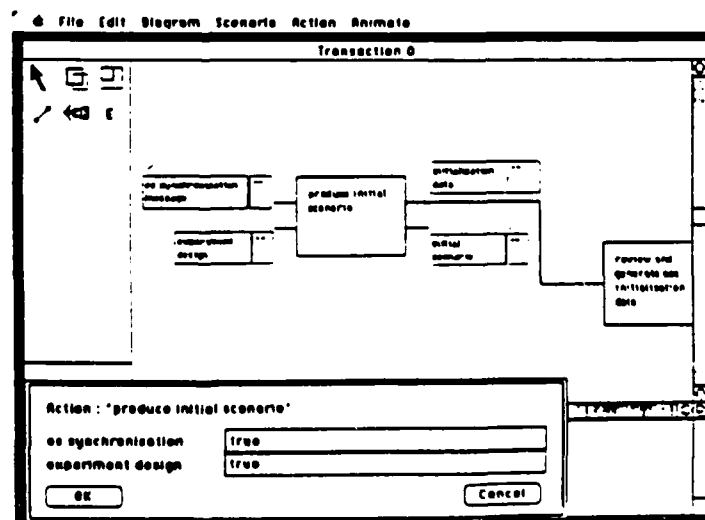
To add a second definition for produce initial scenario, click on the Add button. Enter *_x*, *_y*, *false* and *false* in the 4 text areas. This says that whatever the input values are, the two outputs will have the value *false*.

Click on the Save button to save this definition, then the Quit button to remove the definition window.

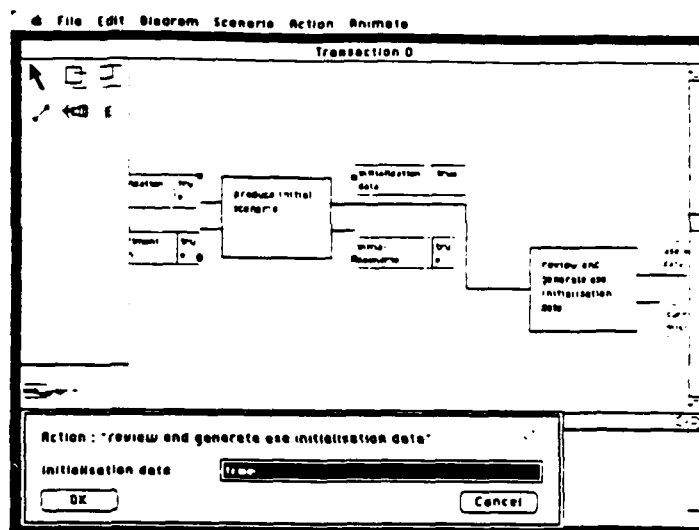
The other two actions in the transaction can be similarly defined.



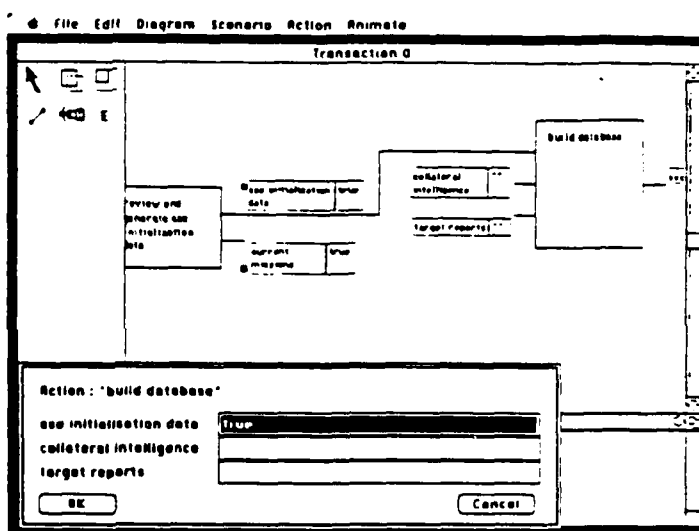
We are now ready to execute the actions. Click on the *Execute* tool in the tool palette of the transaction window to make it the current tool. Then click on the action box of *produce initial scenario*. A dialog box will appear to prompt for the input values of *produce initial scenario*. Enter *true* in each of the empty text areas and then click the OK button to execute the action. Notice that the data values in the transaction will be updated.



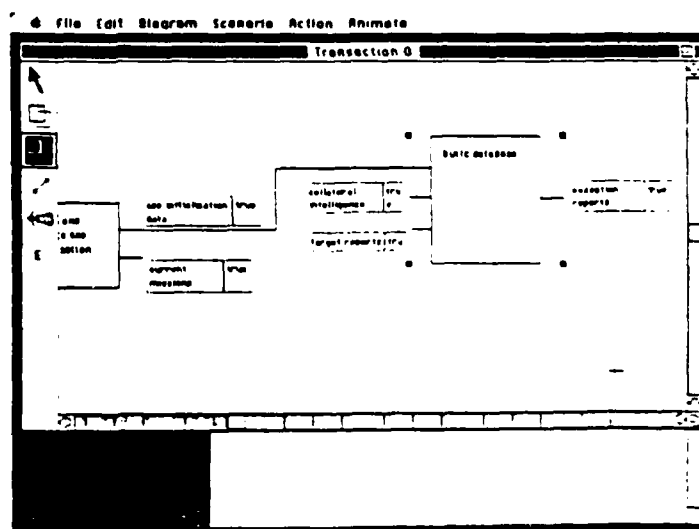
Now click on *review and generate ase initialisation*. A dialog box will appear to prompt for its input values. Notice that the value of *initialisation data* produced by *produce initial scenario* has been picked up by this action. Click on the OK button to execute the action.



To execute *build database*, click on its action box with the *execute* tool. A dialog box appears with the value of *ase initialisation data* pre-filled. This is the value generated by the previous action. Enter *true* in the other blank text areas and click OK.



We have just played out the scenario in which all input data has the value *true*. If we want to play a new scenario, choose *Clear scenario* from the Scenario menu to clear the current scenario and then repeat the last 3 steps with different input values. Alternatively, we can replay the scenario in Go, Go-Go or Step-Step mode. Refer to the Animator User Guide for details of how to replay a scenario.



Method Guidance Demonstration

1. Demonstration Instructions

2. Demonstration Comments

Figures

Key to Instructions:

Outline

- menu

Bold

- menu or dialogue option

- selection

The Script

The following pages contain the operations required to run the demonstration of the enhanced CORE Analyst. The demonstration is partitioned into six major functions :

- 1.0 Opening the demonstration Project
- 2.0 Opening and reviewing an example Tabular Collection
- 3.0 Auto-Generating ammending and checking a Single Viewpoint Model
- 4.0 Re-opening the Tabular Collection and reading the notes attached.
- 5.0 Auto-Generating a Summary diagram of the Tabular Collection stage
- 6.0 Getting advice on the Specification, and changing the advice heuristic priorities.

The Demonstration instructions are in a tabular form as Follows :

Ref. Number	Menu (if applicable)	Menu option or Selection	Object to select	<i>Figure Number</i>
----------------	--------------------------------	-------------------------------------	------------------	--------------------------

Menu s always appear in **Outline** script

Menu options appear in **Bold** type face

Selection is denoted by the symbol <

The Object on which the action is performed is written in plain text unless it is a dialog item, in which case it is denoted by using the **system font**.

Finally at the far right of the page, a *Figure Number* is shown for those actions that have corresponding screen dumps (appended to this document).

- 1.0 Opening the Project
- Project Open
- Copy of ASET Demo 2
- < **Open** *Figure 1*
- < **Continue** *Figure 2*
- 2.0 Opening a Sample Tabular Collection
- File/Print Open
- ASE Tabular Collection
- < **Open** *Figure 3*
- 2.1 Previewing the TCF
- Page Layout Reduce to Fit
- < *Figure 4*
- 2.2 Closing the sample TCF
- File/Print Close
- 3.0 Auto-Generating a Single Viewpoint Model
- File/Print New *Figure 5*
- < **Single Viewpoint Modelling**
- < **OK**
- ASE Element SVM *Figure 6*
- ASE Element
- < **Auto Generate**
- < **OK**

3.1 Setting up the paper for the new SVM

File/Print **Page Setup**
 < **A4 Letter**
 < **Landscape**
 75 %
 < **OK**

Figure 7

3.2 Switching Off interactive Checking

Check **Checking On**

3.3 Tidying up a portion of the diagram and introducing a new T & C component

< < < You're on your own here !!

3.4 Previewing the Ammended SVM

Page Layout **Reduce to Fit**
 <

Figures 8&9

3.5 Selecting the new data flow

< ASE message to T & C

Figure 10

3.6 Setting up Check Parameters/Options

Check **Project**
Check **Warnings**

3.7 Invoking a check on the data flow

Check **Check Now**

Figure 11

3.8 Closing the new SVM

File/Print **Close**

4.0 Re-Opening the ASE Tabular Collection and finding notes attached

File/Print **Open**

 ASE Tabular Collection
 < **Continue**

Figure 12

4.1 Reading the Notes attached to the TCF

	Find	Notes		
			New Data Introduced	<i>Figure 13</i>
		<	OK	
		<	Finish	<i>Figure 14</i>
5.0	Generating a new Summary for the Tabular Collection Stage			
	File/Print	New		<i>Figure 15</i>
		<	OK	
		<	Viewpoint Structuring	<i>Figure 16</i>
		<	Tabular Collection	
		<	OK	
			TC Summary	<i>Figure 17</i>
		<	Save	
5.1	Setting up the Paper			
	File/Print	Page Setup		
		<	A4 Letter	<i>Figure 18</i>
		<	Landscape	
			50 %	
		<	OK	
5.2	Selecting the icon representing the ASE Element TCF			
		<	ASE Element	
5.3	Finding the notes attached to this diagram			
	Find	Notes		
			New Data Introduced	<i>Figure 19</i>
		<	OK	
		<	Finish	<i>Figure 20</i>
5.4	Selecting the icon representing Timing & Control			
		<	Timing & Control	
5.5	Finding the check summary for this diagram			
	Find	Check Summary		<i>Figure 21</i>

5.0	File/Print	Close	
6.0	Getting Advice on the Specification		
	Check	Advice...	<i>Figure 22</i>
		<	Rank
		<	start the TCF for envir....
		<	More
		<	Finish
			<i>Figure 24</i>
6.1	Changing the Priorities of the Advice		
	Check	priorities...	
		<	type
		shift <	40 %
		<	Increase
			<i>Figure 25</i>
6.2	Reviewing the changed priorities		
	Check	priorities...	
		<	Cancel
			<i>Figure 26</i>
6.3	Observing the effect on advice		
	Check	Advice...	<i>Figure 27</i>
		<	Rank
		<	add the data ase response.....
		<	More
		<	Finish
			<i>Figure 29</i>

Comments

1.0 Opening the Project

The dialog shown in figure 2 may take some time to appear. This is because CORE Analyst is creating a hierarchical structure of the diagrams in the specification.

2.0 Opening the ASE Tabular Collection

The ASE Element TCF is chosen because it is central to the specification and demonstrates the features we wish to exhibit in the auto-generated SVM

3.0 Auto-Generating the SVM

The ASE Element SVM takes about 2 minutes to auto-generate on the Mac II. Note that this is a preliminary draft of the SVM and the following types of information are usually added in the SVM stage :

Pool and Channel attributes to data-flows

Time ordering

Possible action decompositions

Internal data flows

No attempt is made to establish a good layout of the diagram beyond what may be observed. However, this first cut is guaranteed to be consistent with the TCF and might take 30 minutes to generate by hand.

3.2 Switching Off Checking

While drafting a diagram it is useful to switch off interactive checking by Unticking the Checking On option in the Check Menu

3.3 Tidying the SVM

arranging the outputs of this diagram is quite simple and only takes a couple of minutes. The inputs are slightly more tricky and may be saved for a later day.

It is now that we observe a hole in the specification and would use the Re-Use tools and Animator to find and exercise a suitable fragment that will fill the gap.

This fragment is now entered by hand.

3.6/7 Checking the new data flow

The CORE Analyst checks to see whether the SVM is consistent with the TCF for the ASE Element and tries to ensure that everything connected to the data has been defined for the viewpoint. ... It hasn't and so the error messages shown are displayed.

4.0 Re-Opening the ASE Element TCF

While checking was taking place, active guidance has been busy placing notes (akin to yellow post-it stickers) on the affected diagrams. We may read this note as shown.

5.0 Generating a Check Summary

In order to gain a wider view of the impact of our change, it is best to generate a summary diagram. This takes about 90 seconds (on the Mac II).

5.2 Selecting the ASE diagram

Scrolling across the paper, we see that most TCFs remain unaffected, but that the ASE TCF has been down-graded and the rectangles represent the notes.

5.3 Finding the Notes

We may read the notes from the summary diagram as shown.

5.4 Finding a check summary

It is also possible to obtain the results of the latest Check summary

6.0 Generating advice

The advice is a mixture of normal (normative) CORE stages to be completed or started and recommendations for error correction (remedial) based on the notes that have been generated. The numbers have no absolute meaning, but show *relative importance/urgency* of the advice. The most important advice is to start the environment simulation TCF since this may well have ramifications on the other TCFs on this level.

6.1 Priorities

The listed attributes have the following implications :

action This factor controls the weightings given to the type of action that is recommended. Typical actions are the to start or complete diagrams, rename or delete data items etc. If the user believes that the type of action is important, then the action factor should be given a high priority.

density This factor becomes significant when advice is grouped into diagrams, viewpoints or causes. The density factor indicates the number of recommendations for a given diagram (or viewpoint or cause). A high density factor will give prominence to diagrams with a large number of recommendations.

level This factor is concerned with the distance of the recommendation from the current focus of interest. The current focus of interest is a combination of the level within the viewpoint hierarchy and the stage of the specification. Thus, for example, the system may believe that the current focus of interest was :- Doing Tabular Collections for Level 1 Viewpoints.

In this case recommendations for actions concerning Level 2 and below viewpoints would weigh less highly than actions concerning Level 1 viewpoints and actions concerning Single Viewpoint Models would weigh less highly than actions concerning Data Structuring.

object The significance of the object of the recommendation is controlled by this factor. Recommendations involving actions should be considered before those acting on data, advice pertaining to Tabular Collections should be weighted higher than advice associated with Single Viewpoint Models. A high object factor will increase the weighting of recommendations towards the fundamental objects in CORE (viewpoints, actions and data) and the earlier stages in the specification.

type There are two types of recommendations generated by the Analyst. The first concentrates on the state of diagrams within the specification and the second proposes solutions to the errors detected by the Analyst whilst checking diagrams. It is recommended that the latter errors are corrected before proceeding with the remainder of the specification. A high value of the *type* factor will ensure that recommendations concerning error-correction will be ranked higher than other recommendations.

6.3

Re-examining advice

Increasing the effect of remedial advice now promotes the recommendations to do with notes to the top of the list.

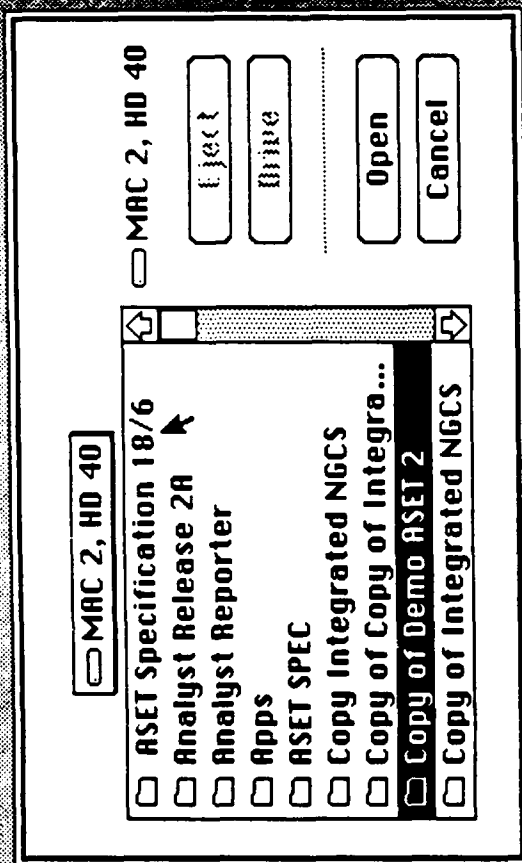


Figure 1

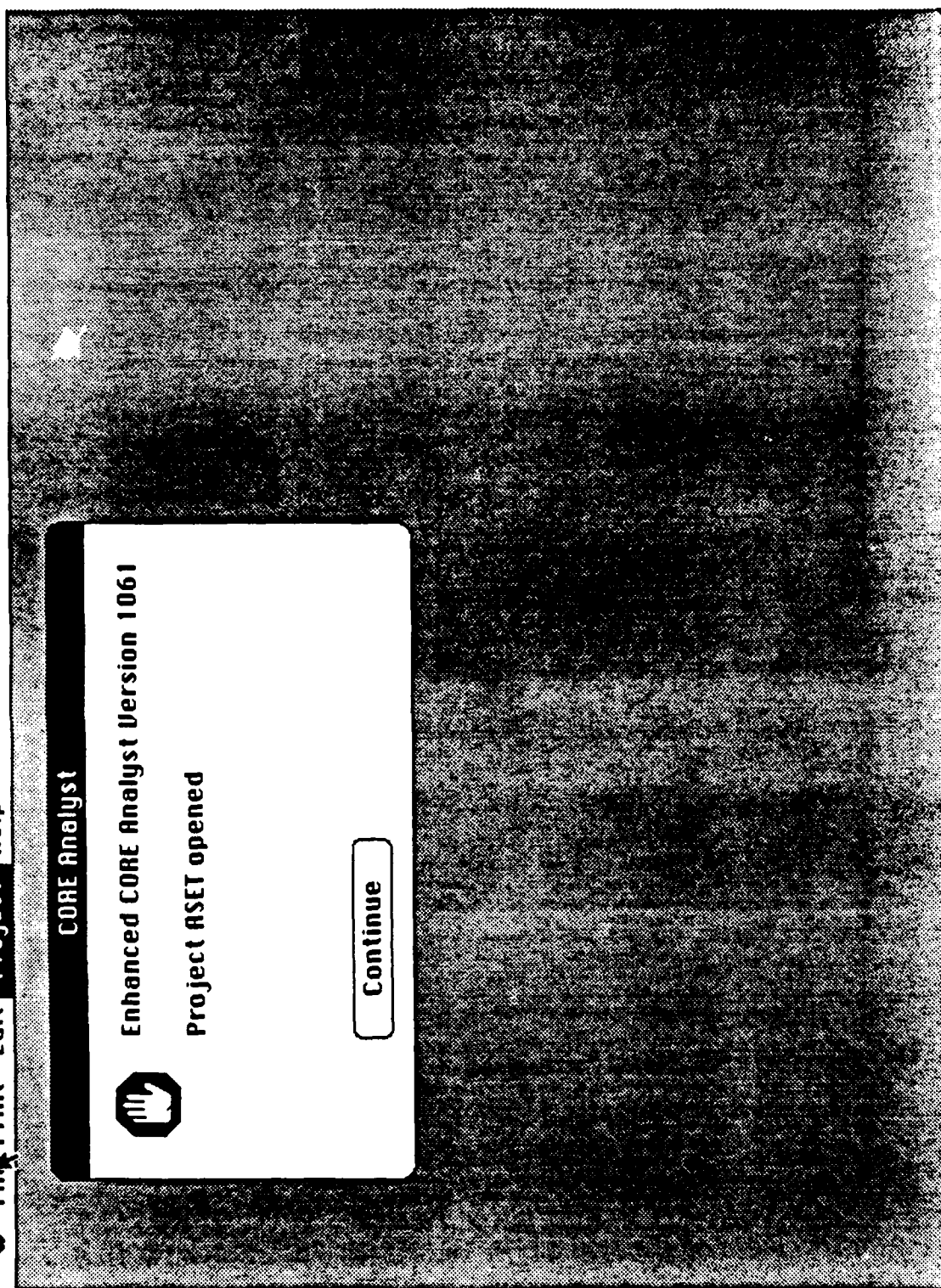


Figure 2

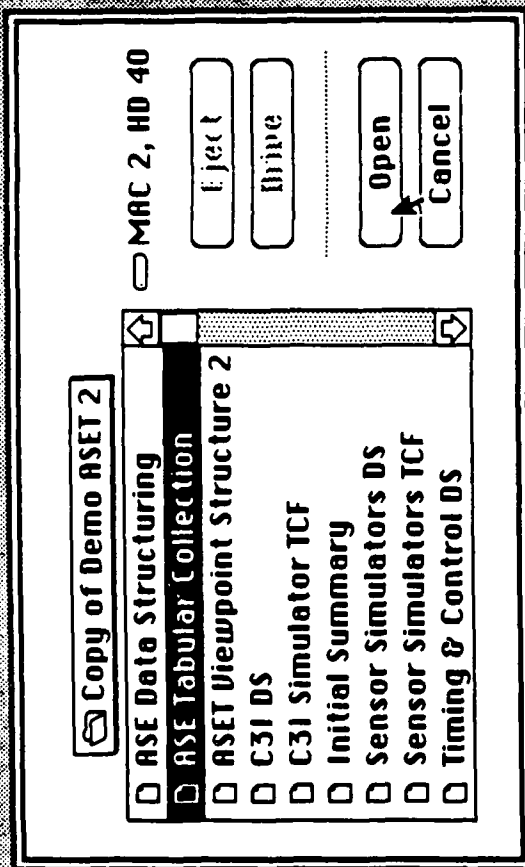


Figure 3

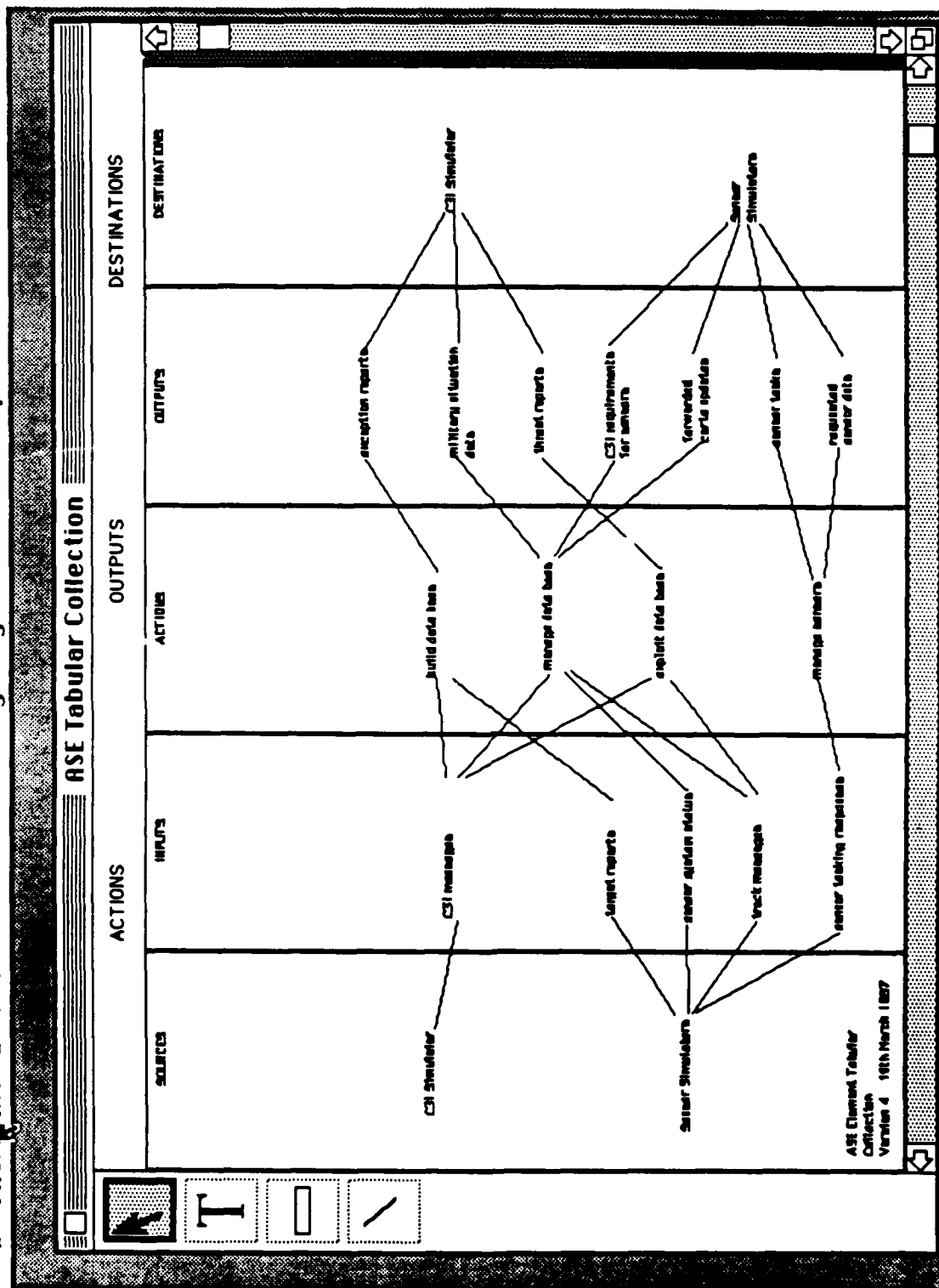


Figure 4

Choose Diagram Type

Please Choose the Type of Diagram

- ☐ Viewpoint Structuring
- ☐ Tabular Collection
- ☐ Data Structuring
- ☒ Single Viewpoint Modelling
- ☐ Combined Viewpoint Modelling
- ☐ Summary Paper

OK

CANCEL

Figure 5

File/Print Edit Project Help

New Diagram

Single Viewpoint Modelling

Diagram Name ASE Element SUM

Viewpoint Name ASE Element

Author

Password

☒ Automatically Generate

Ok Cancel

Figure 6

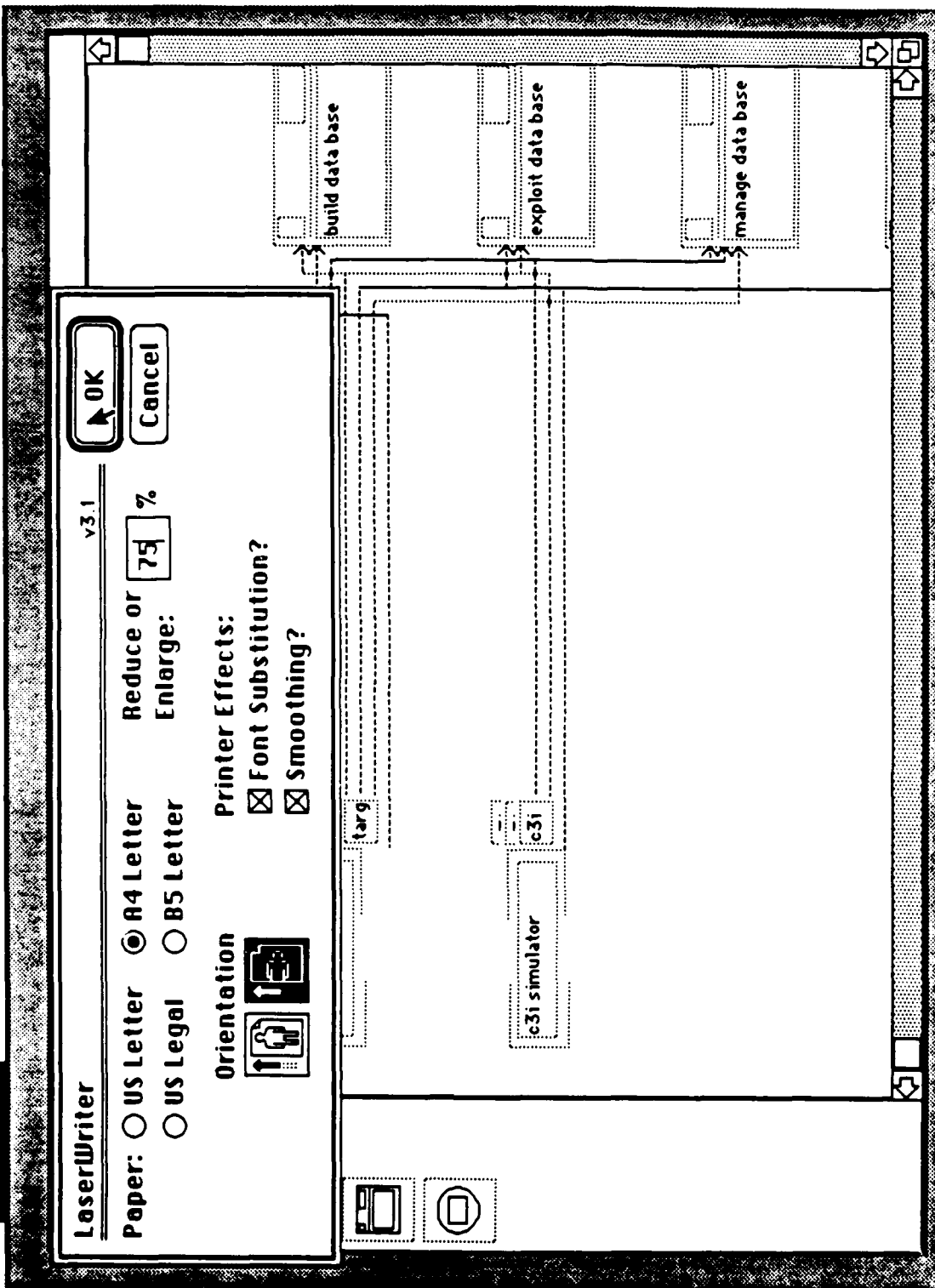


Figure 7

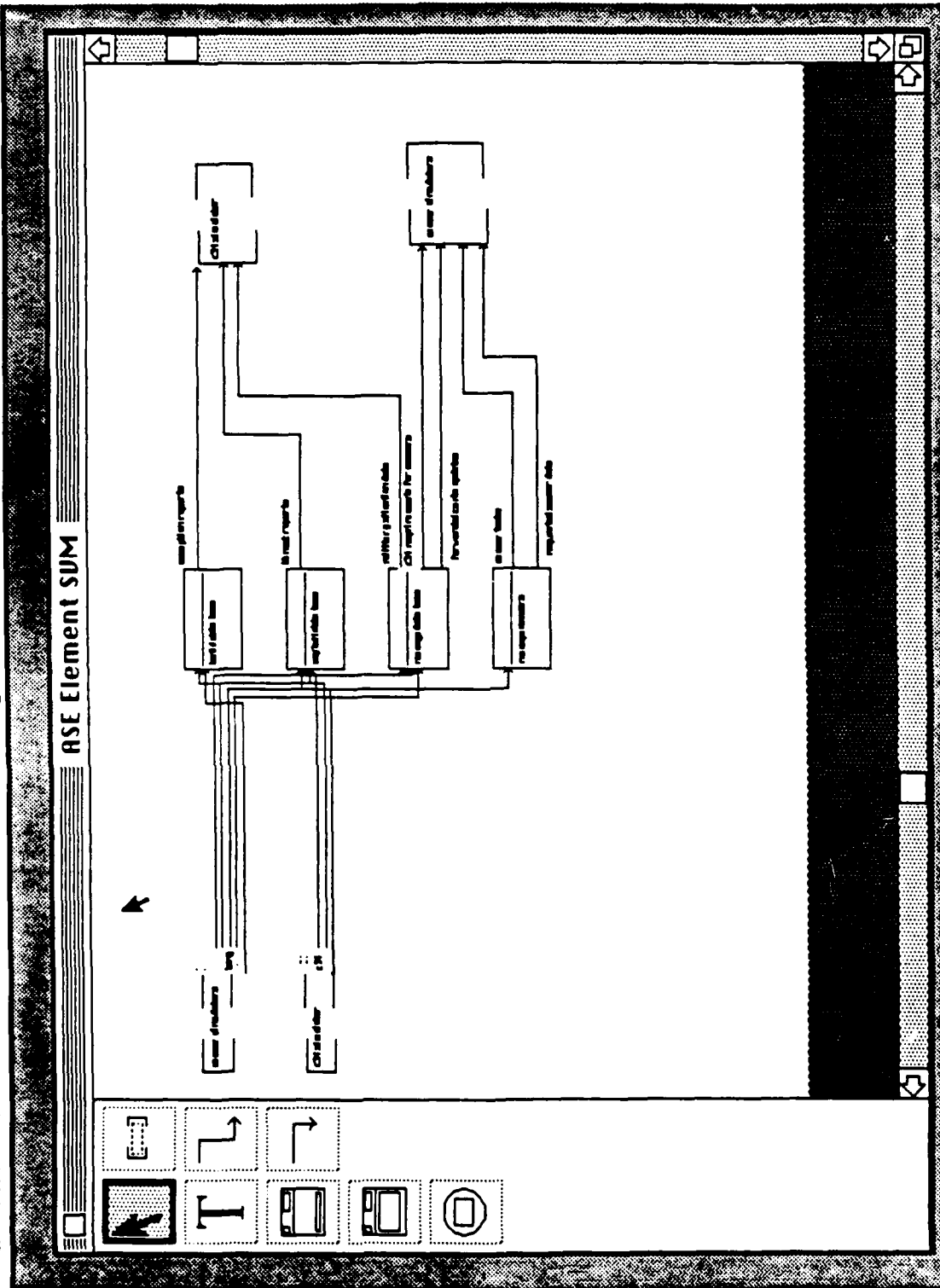


Figure 8

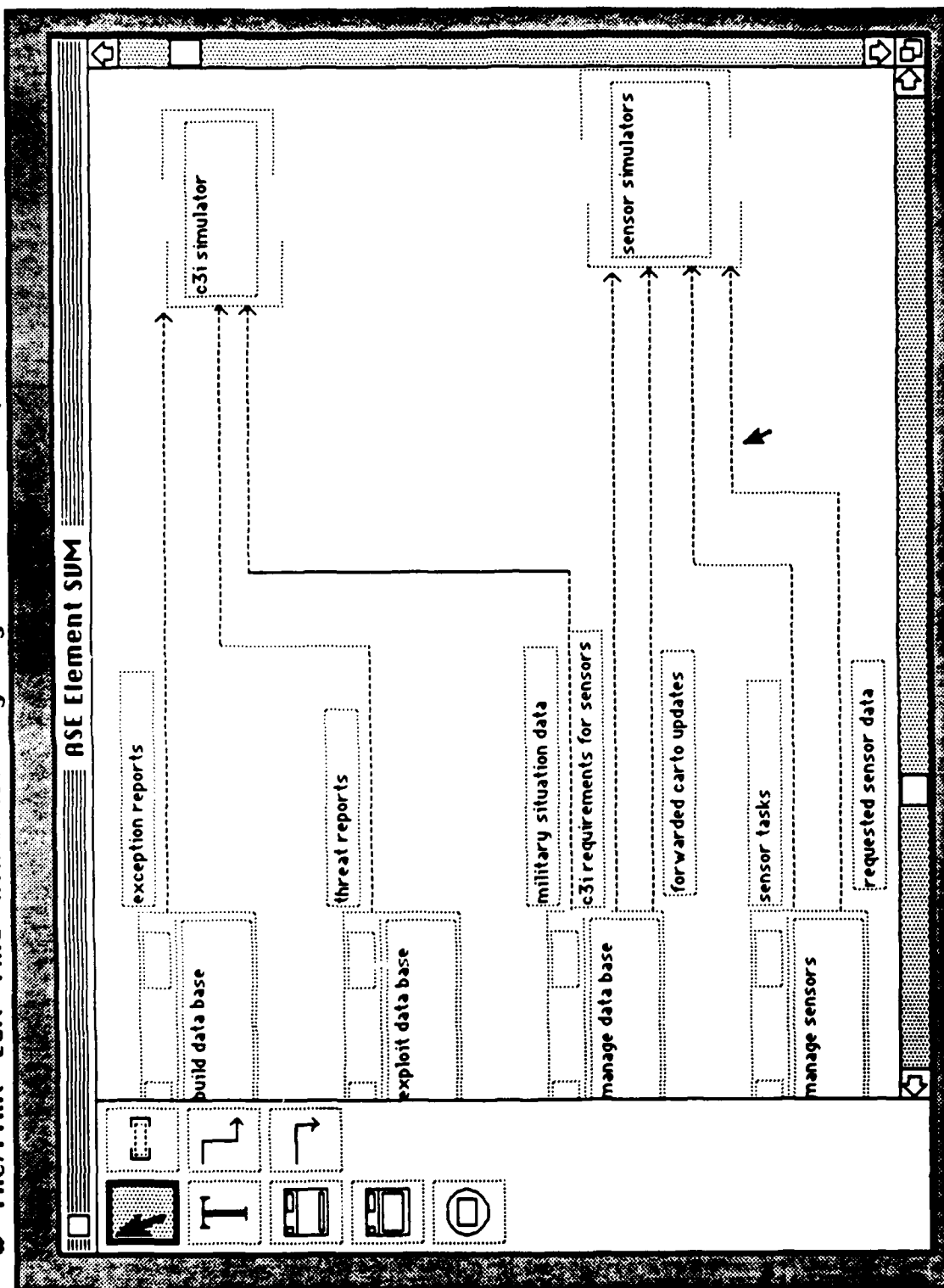


Figure 9

ASE Element SUM

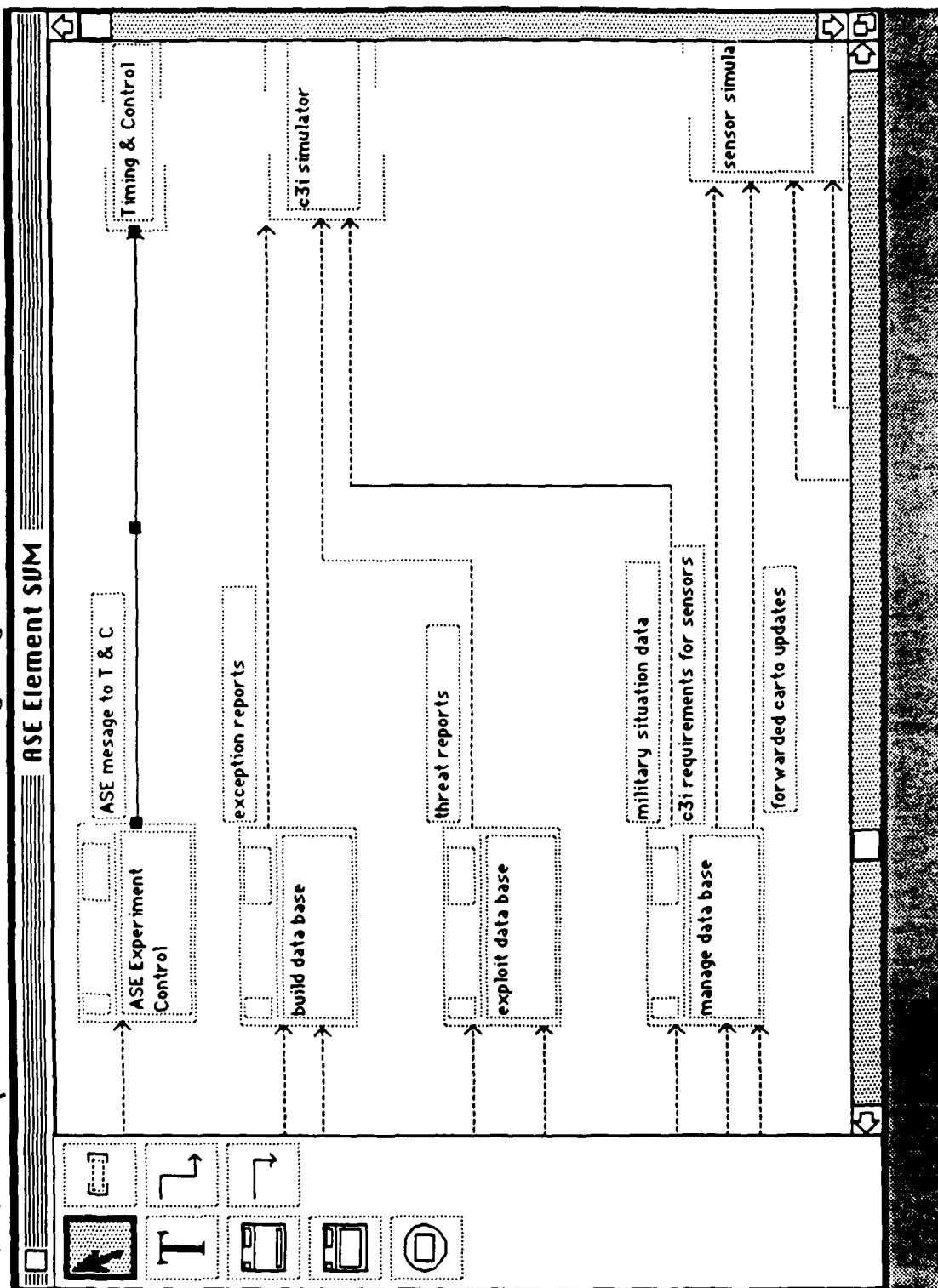


Figure 10

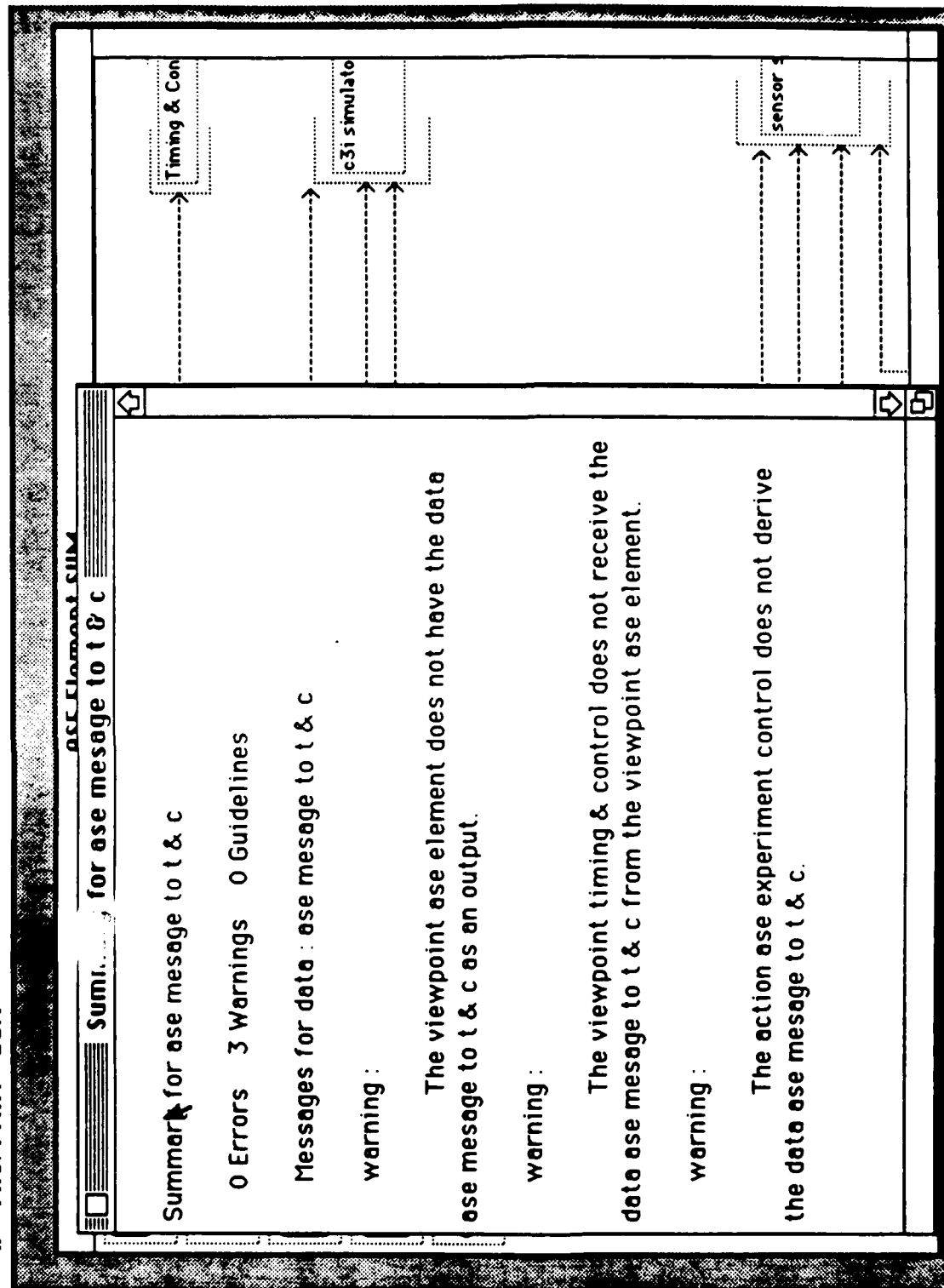


Figure 11

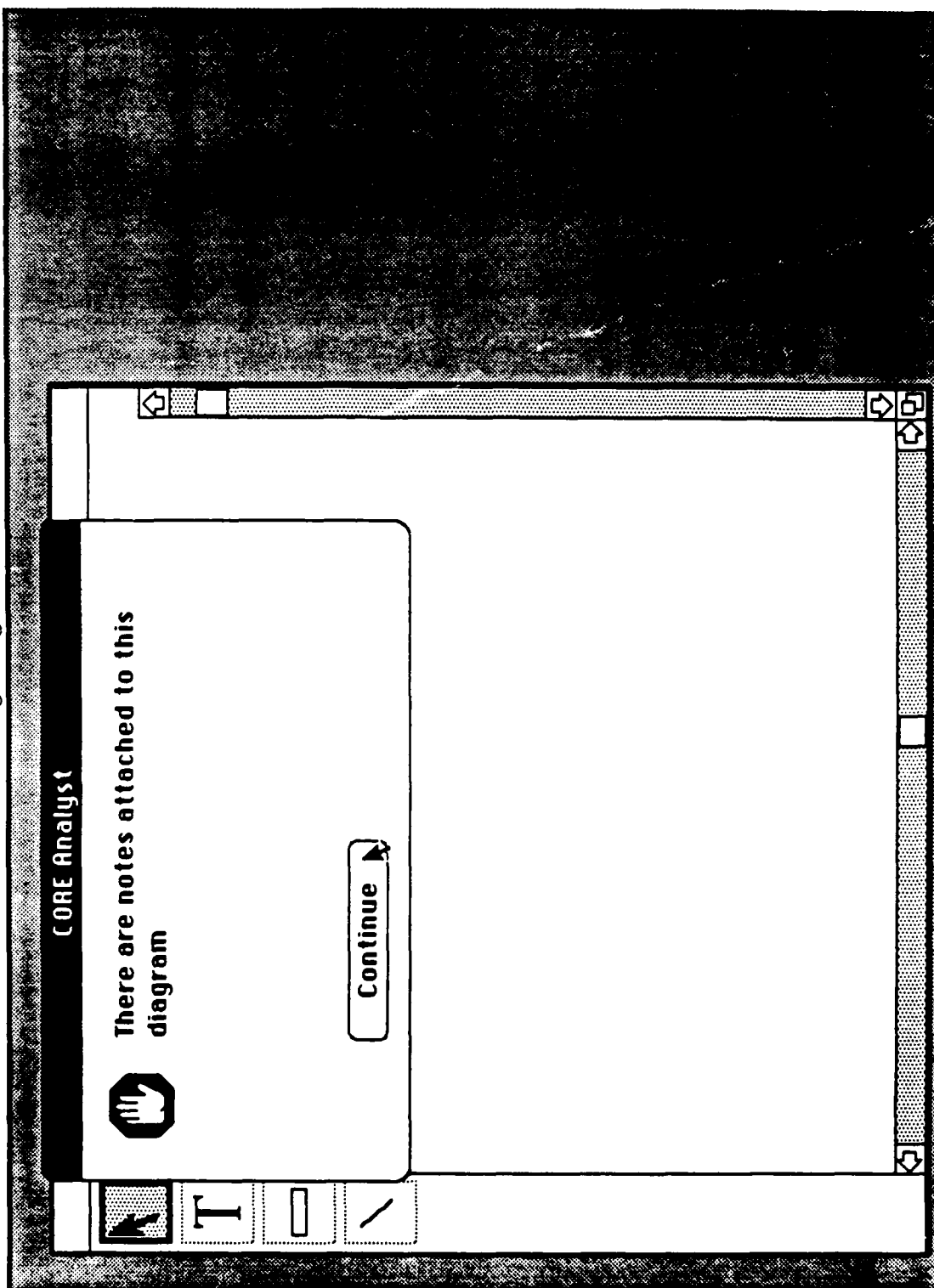


Figure 12

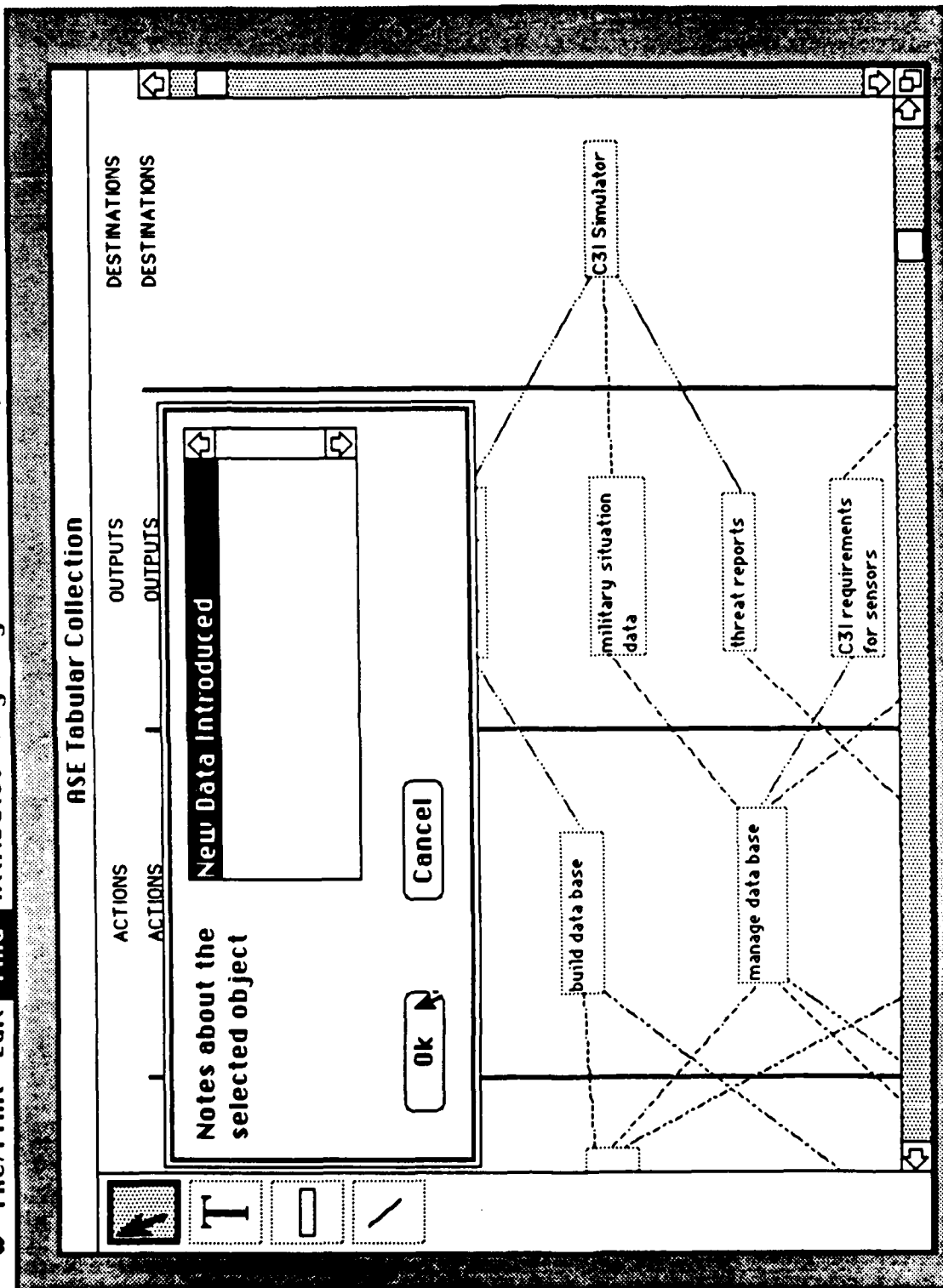


Figure 13

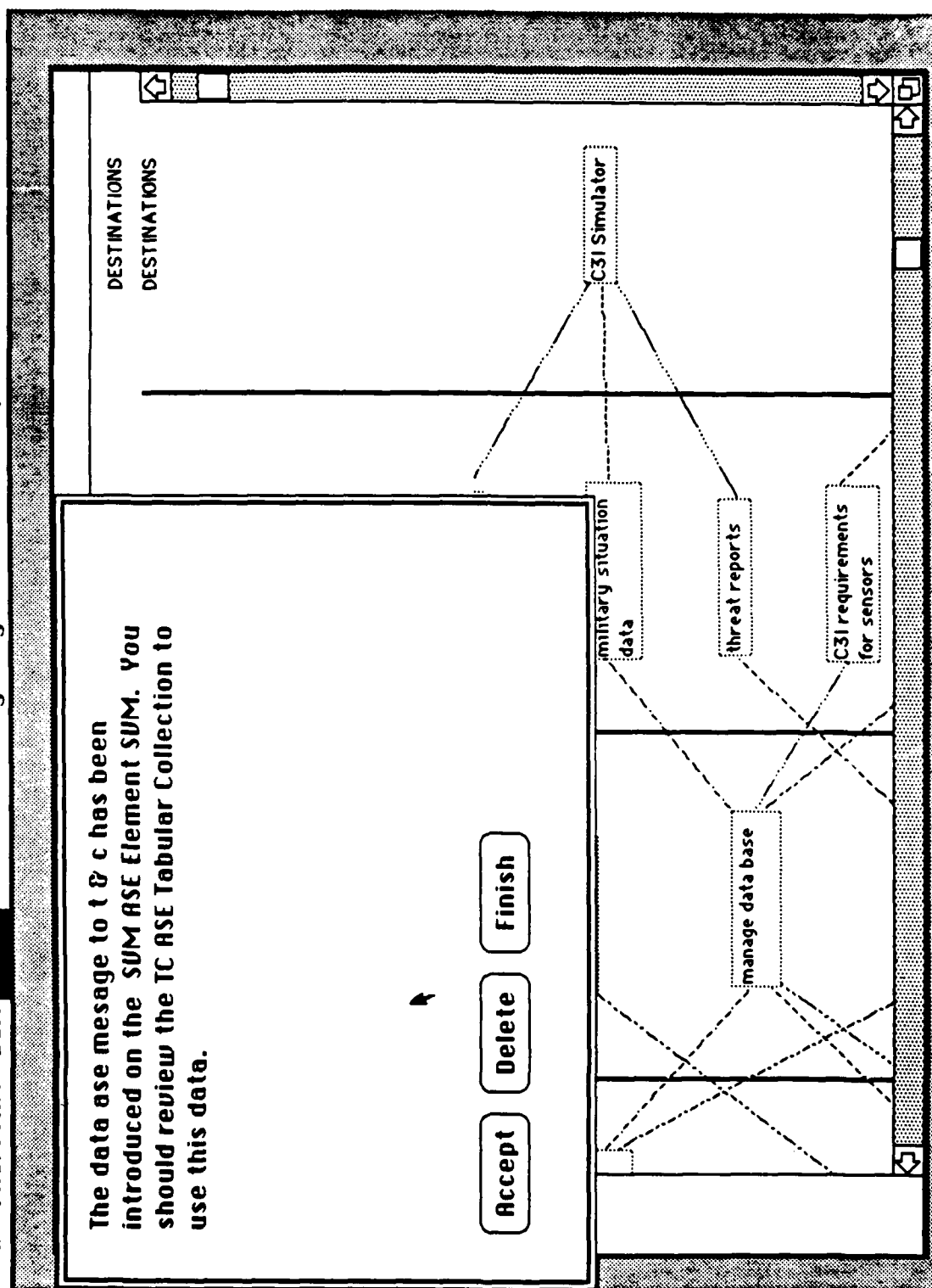


Figure 14

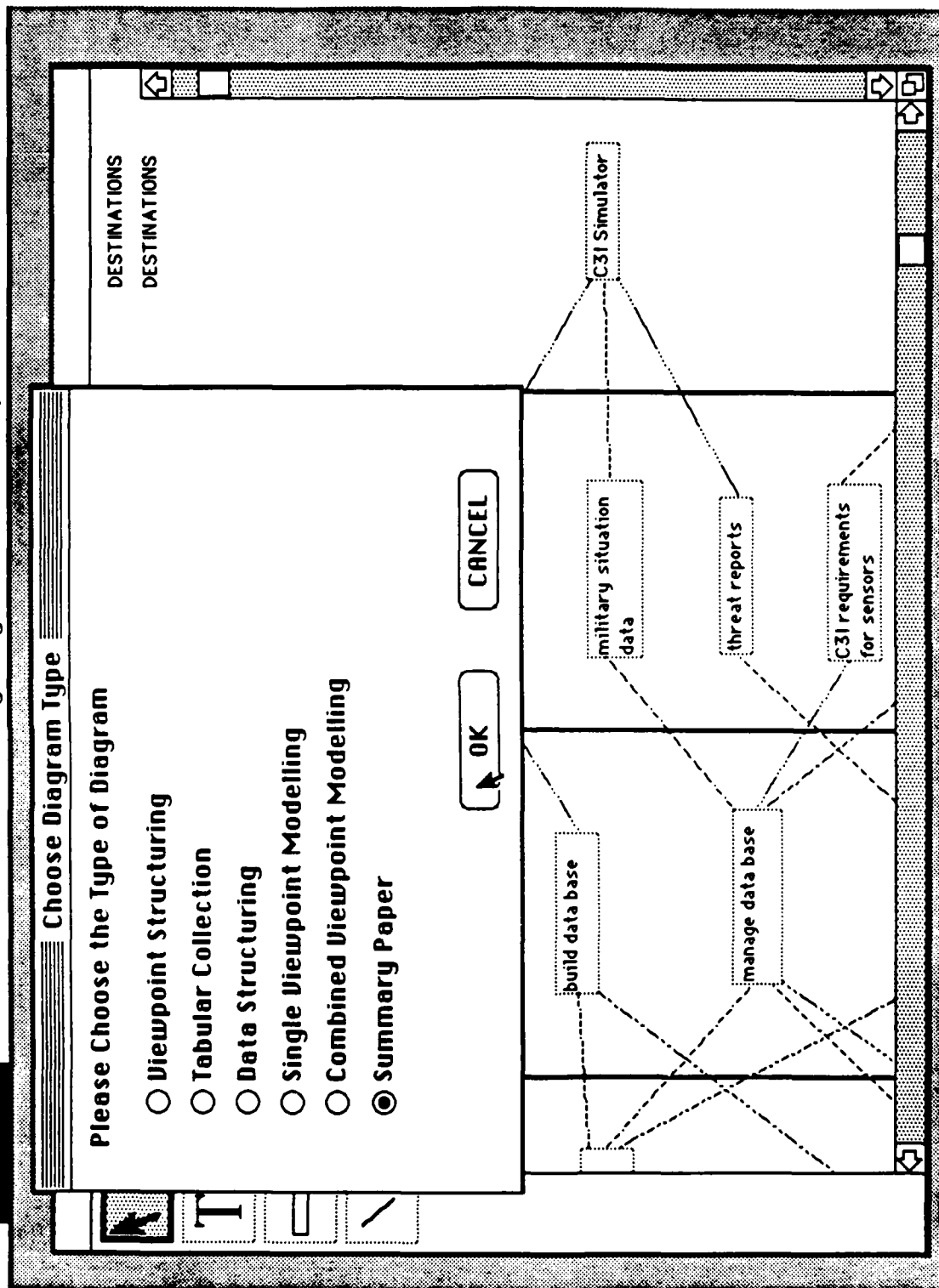


Figure 15

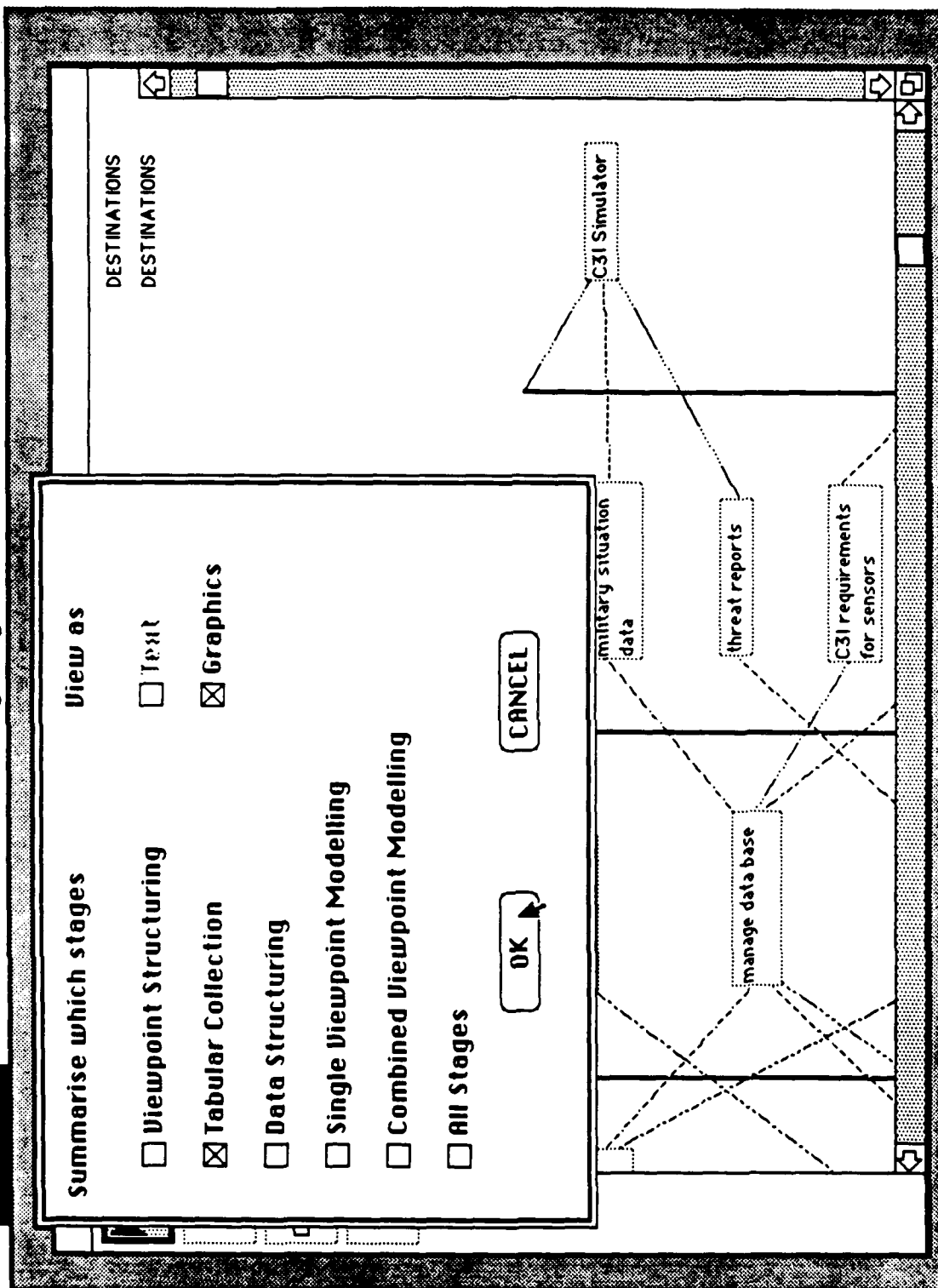


Figure 16

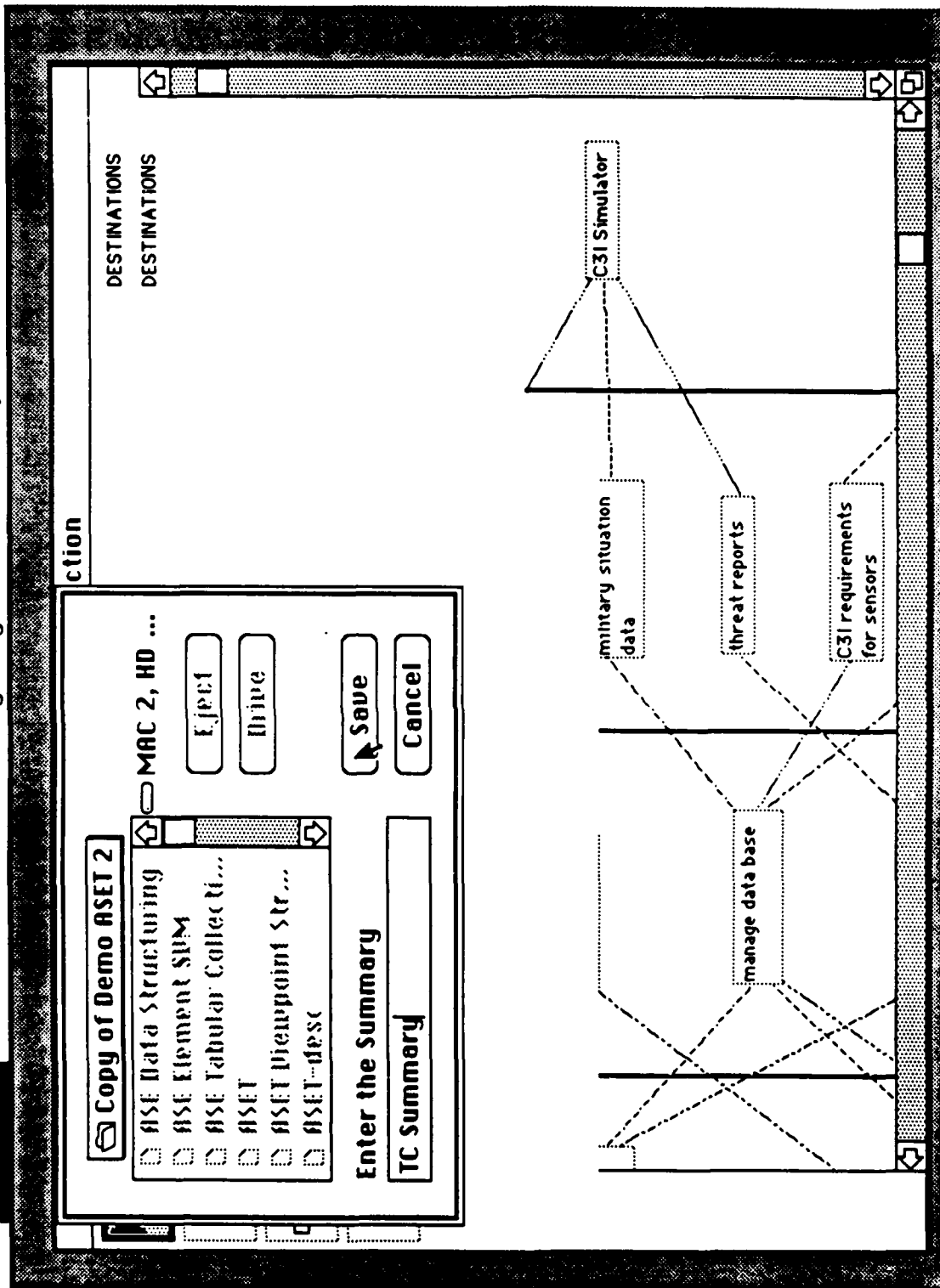


Figure 17

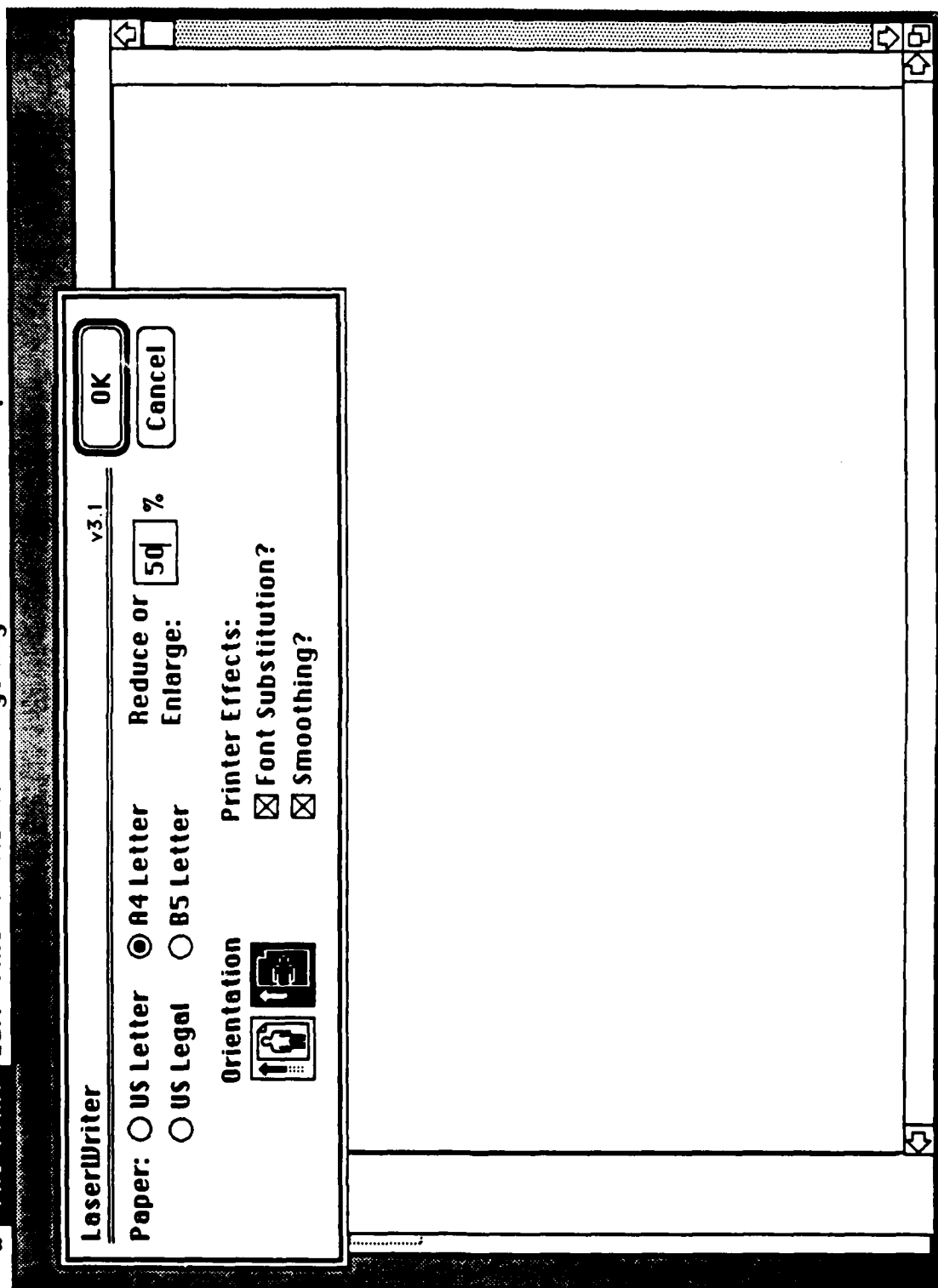


Figure 18

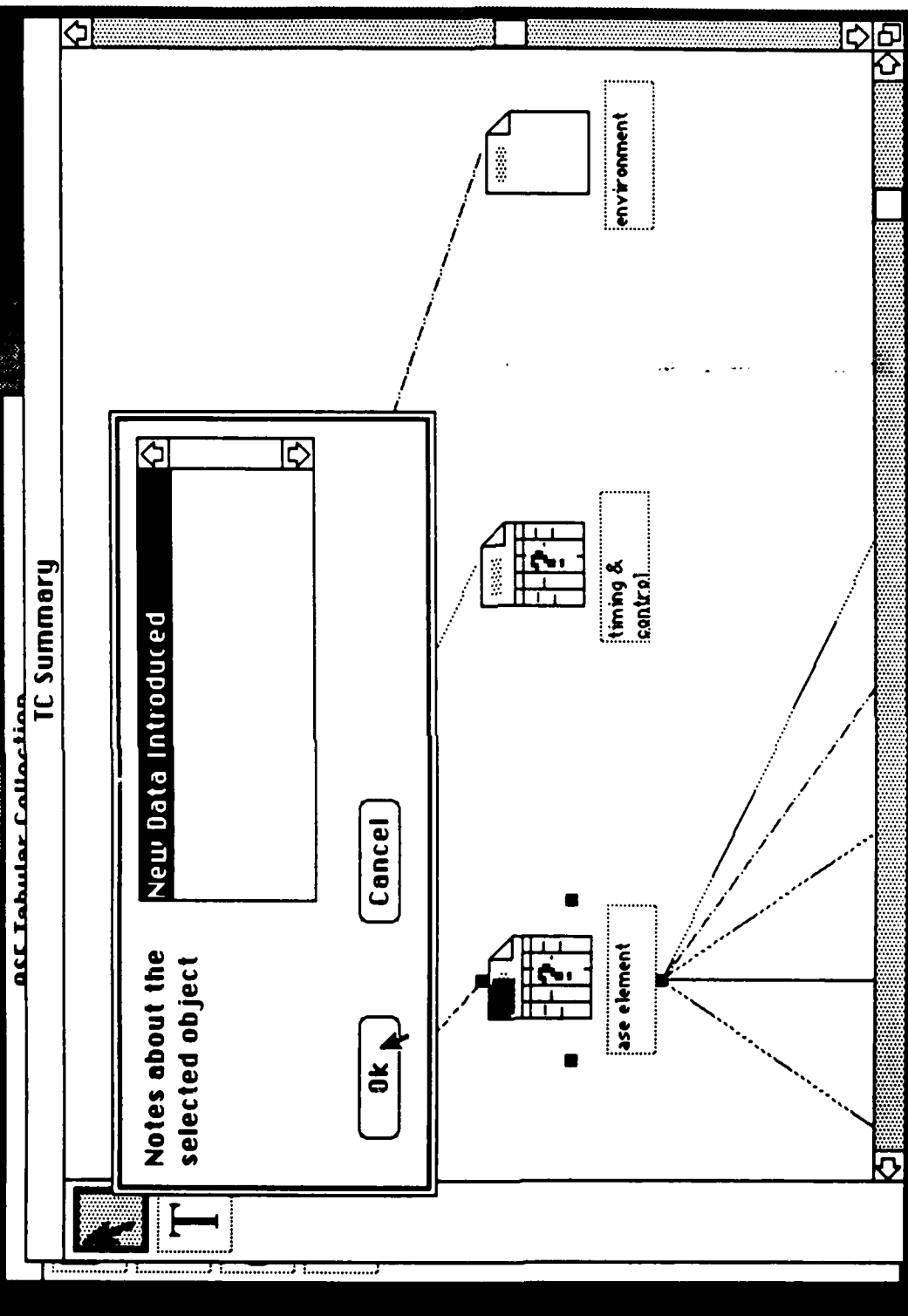


Figure 19

The data ase message to t & c has been introduced on the SUM ASE Element SUM. You should review the TC ASE Tabular Collection to use this data.

Accept

Delete

Finish

ase element

timing & control

environment

Figure 20

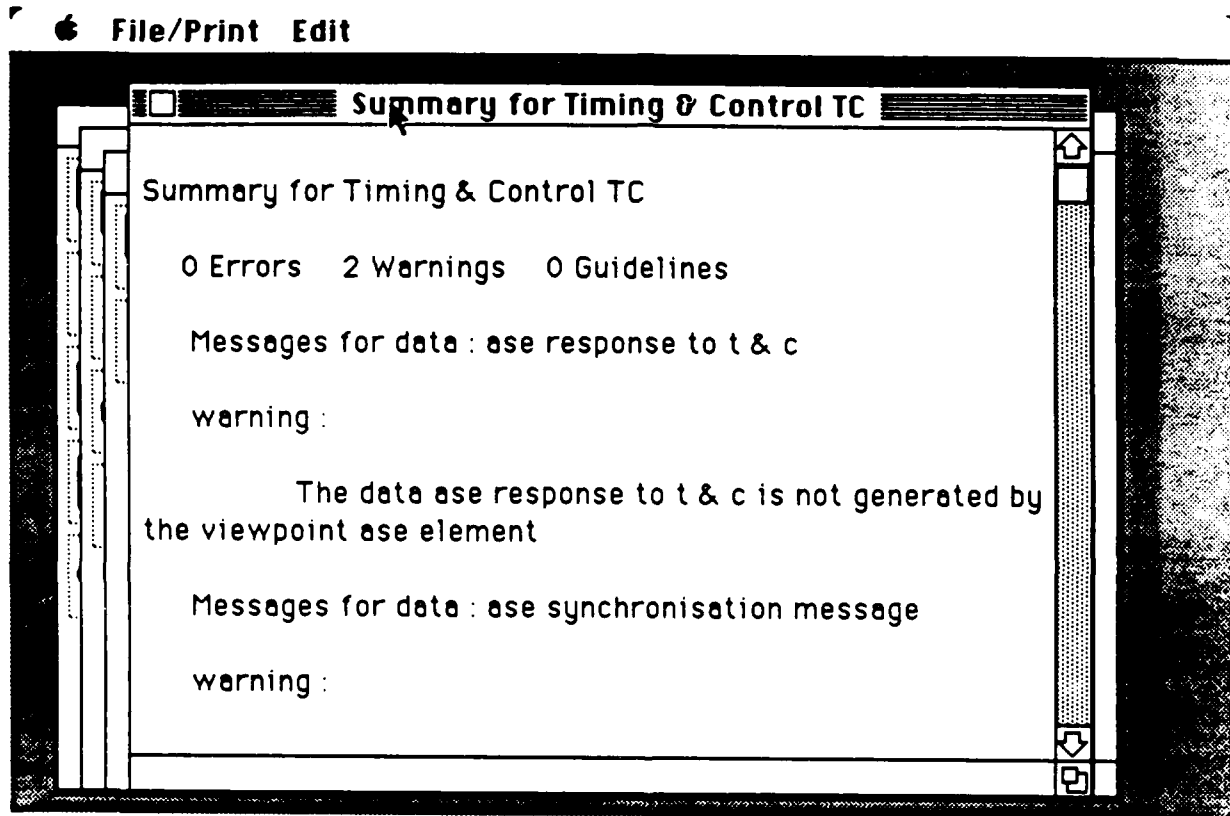


Figure 21

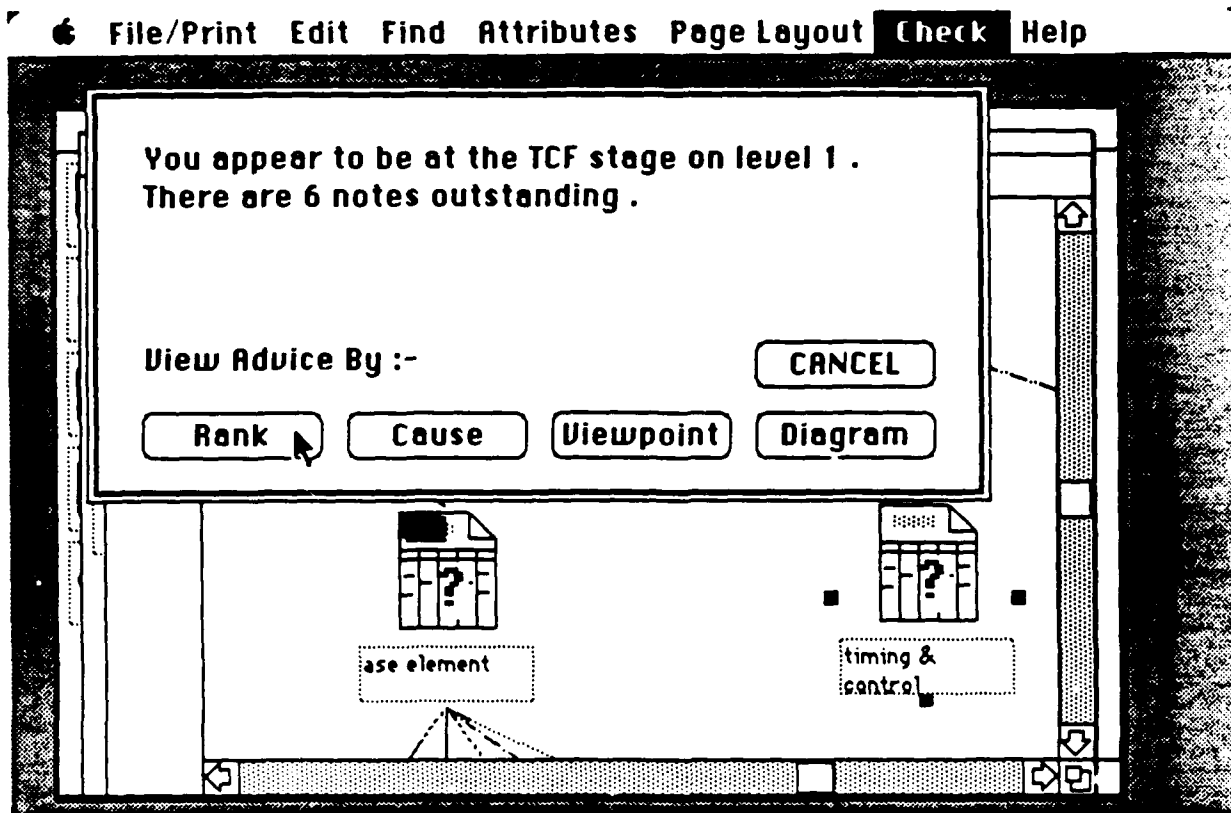


Figure 22

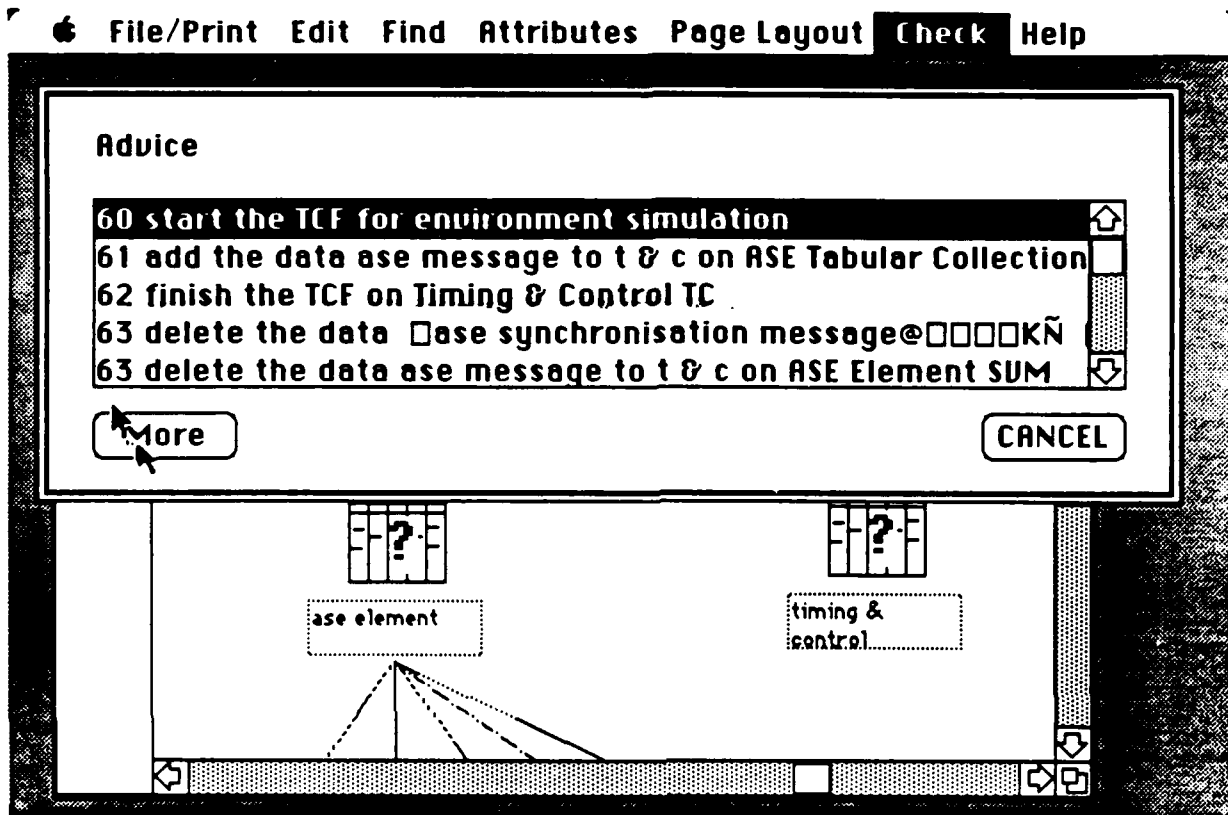


Figure 23

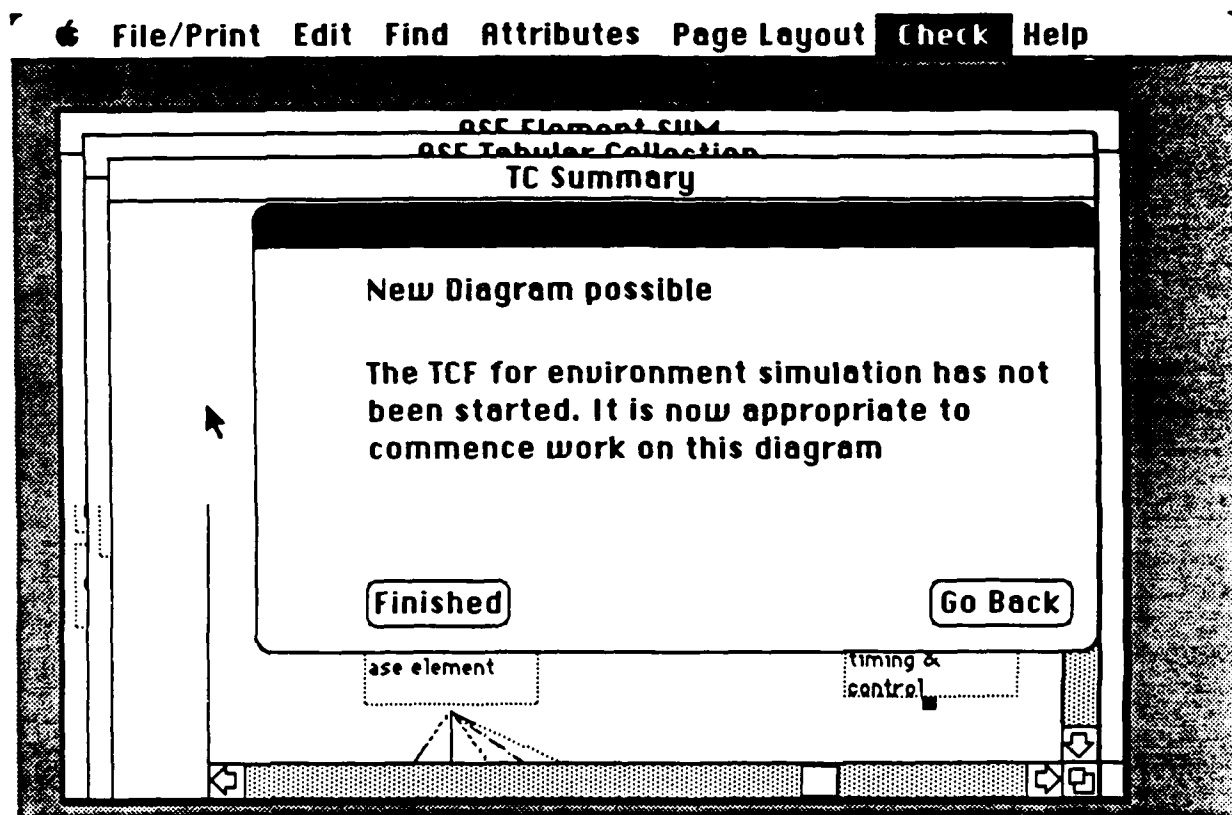


Figure 24

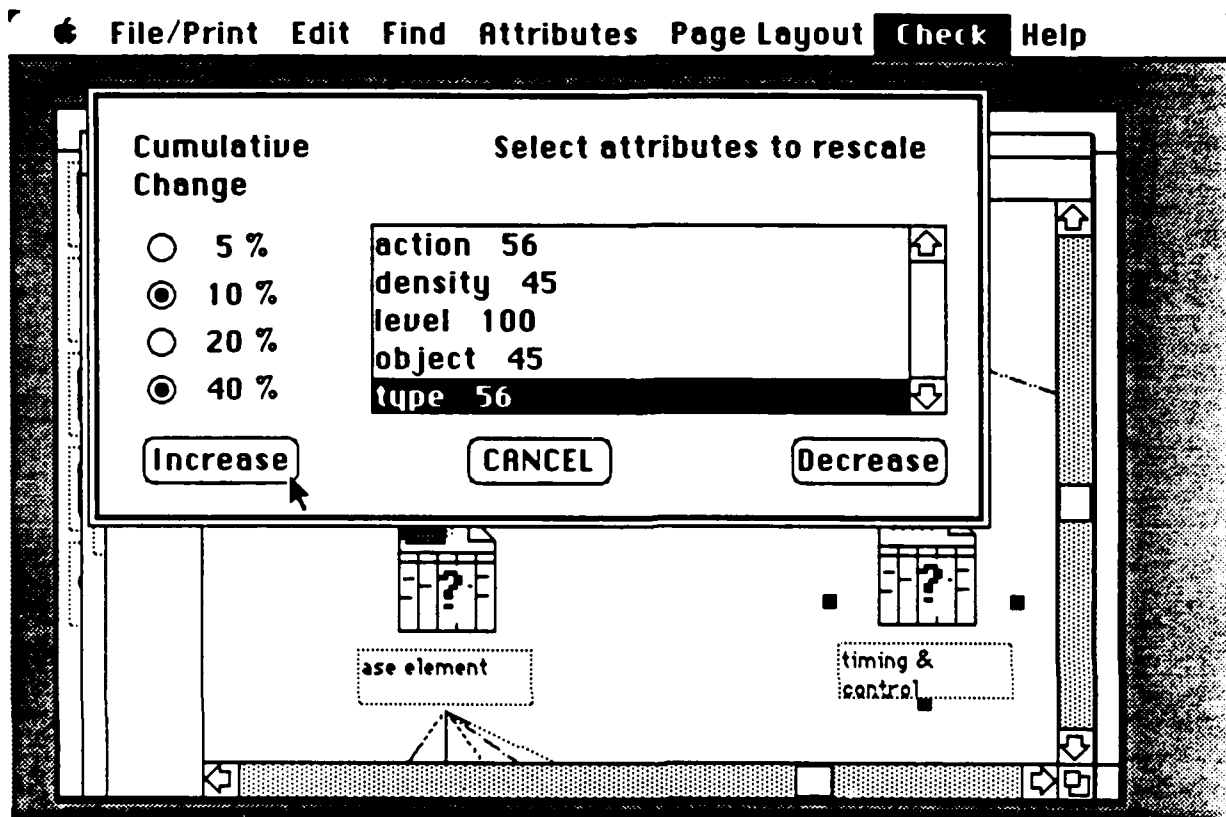


Figure 25

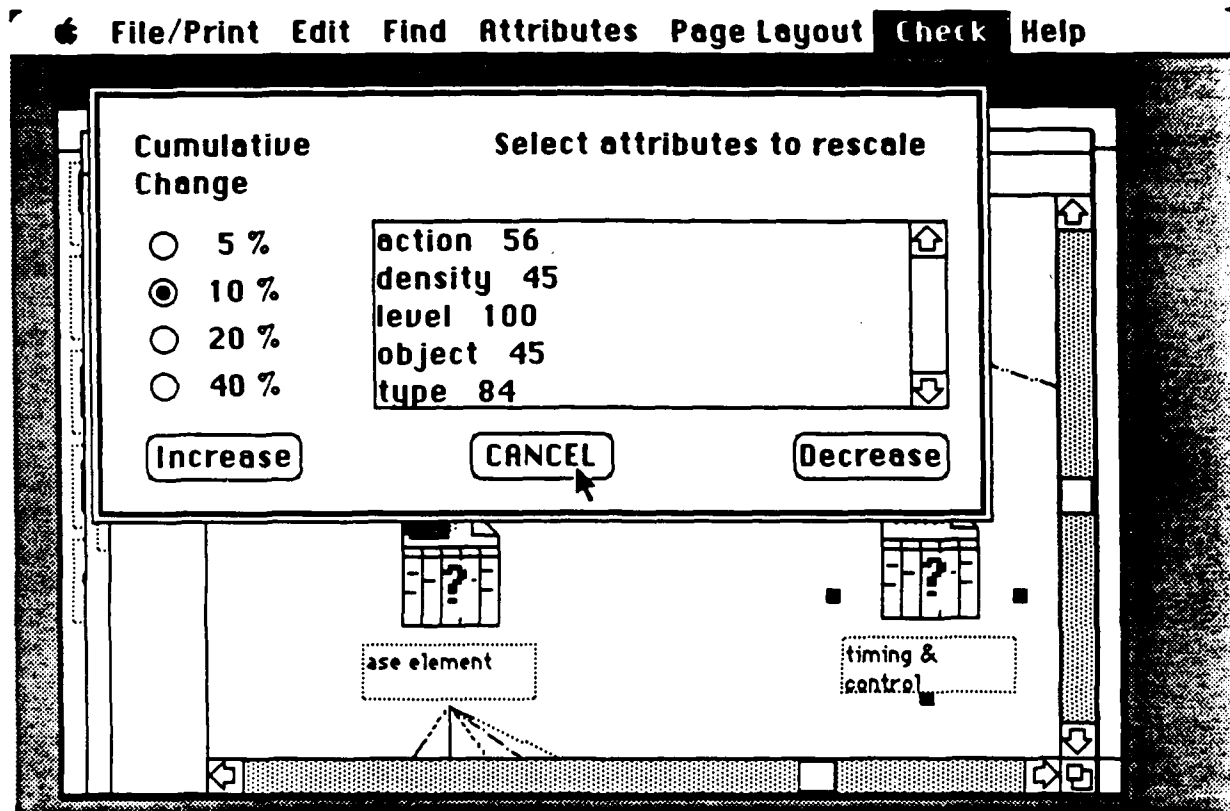


Figure 26

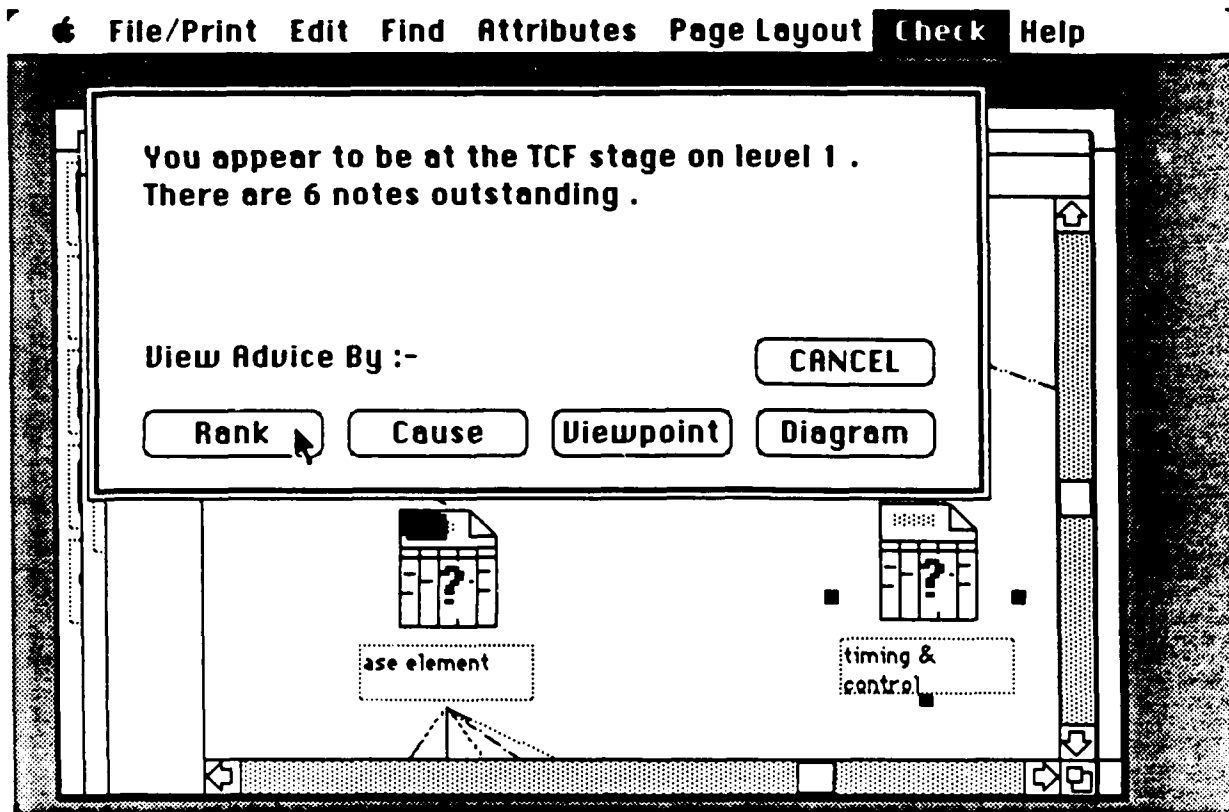


Figure 27

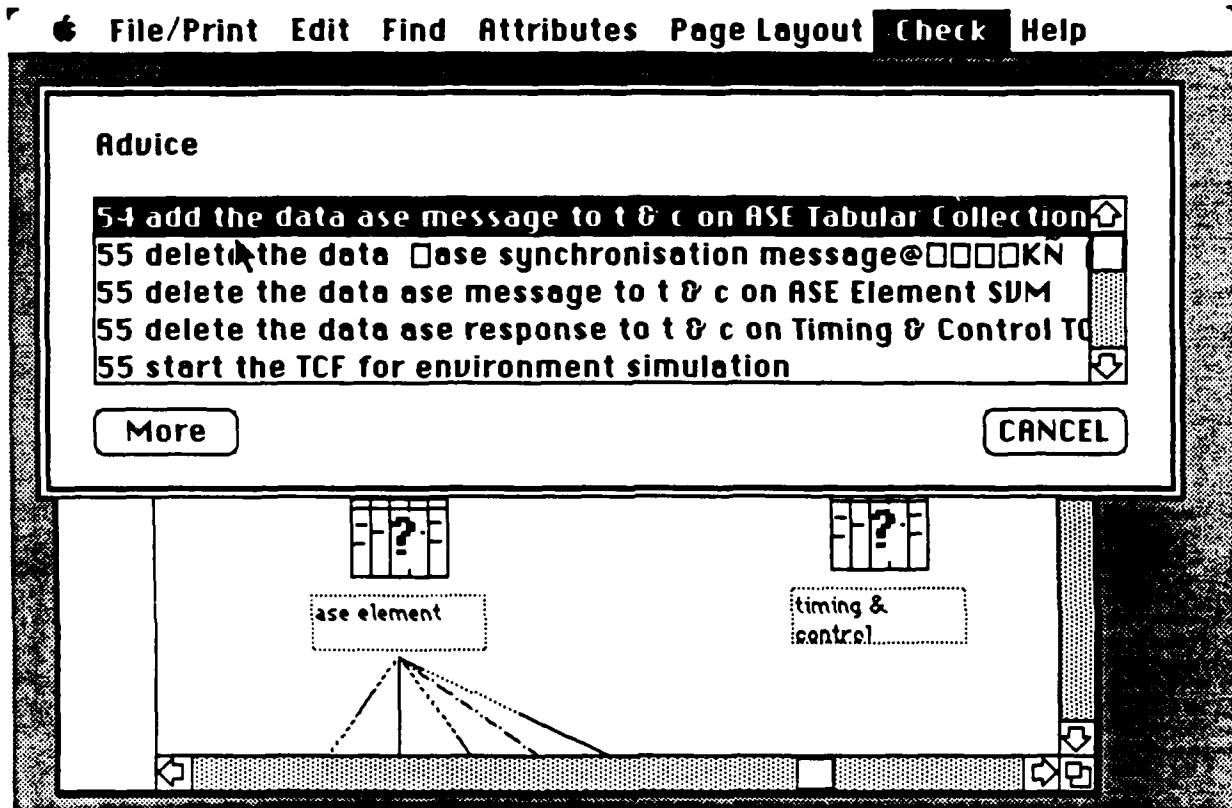


Figure 28

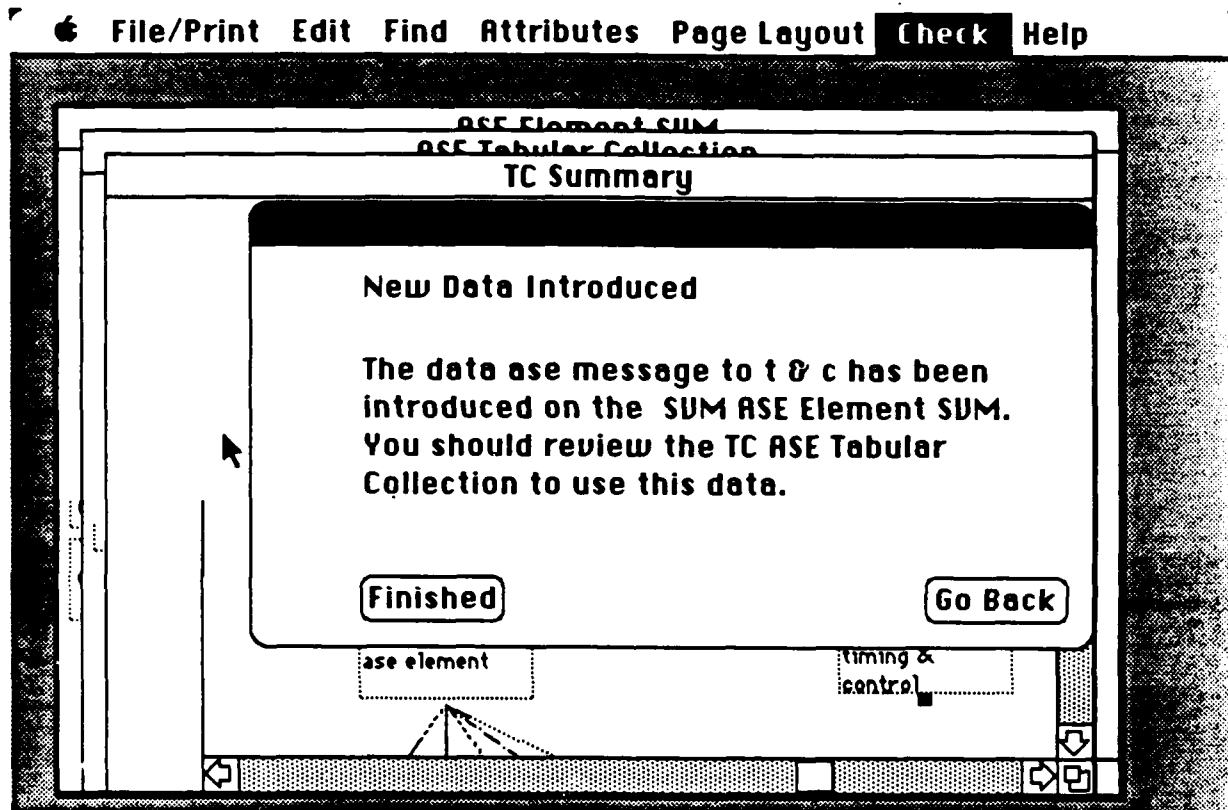


Figure 29