Systems Designers (1986); CORE - the manual; Internal Publication, SD-Scicon.

Teichroew D.& Hershey E. (1977); PSL/PSA: a computer-aided technique for structured documentation and analysis of information processing systems; IEEE Trans. on Software. Engineering, SE-3 (1), pp41-48.

Zave P. (1982); An operational approach to requirements specification for embedded systems; IEEE Trans. on Soft. Eng., SE-8 (3) , pp250-269.

Zave P. (1989); A Compositional Approach to Multi-Paradigm Programming; IEEE Software, September 1989.

Acknowledgements

This research has been supported by the Rome Air Development Centre, Griffiss Air Force Base, under contract number F-49620-85-C-0132. The views and conclusions contained in this paper are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Rome Air Development Center or the U.S. Government.

We would like to acknowledge the contribution made by our colleagues Colin Potts and Keng Ng. Thanks also to our partners SD-Scicon and particularly to Ken Whitehead.

References

Boehm B. (1982); Software Engineering Economics; Prentice Hall.

Finkelstein A. (1987); Reuse of formatted specifications; IEE Software Engineering Journal, September, pp186-197.

Finkelstein A., Kramer J. & Goedicke M. (1990); ViewPoint Oriented Software Development; Proc. of 3rd International Workshop Software Engineering & its Applications; Cigref EC2 V1, pp337-351.

Jackson M. (1975); Principles of Program Design; Academic Press.

Jackson M. (1990); Some Complexities in Computer-Based Systems and their implications for System Development; Proc. of IEEE Int. Conf. on Computer Systems and Software Engineering (CompEuro 90), 344-351.

Klausner A. & Konchan T. (1982); Rapid prototyping and requirements specification using PDS; ACM SIGSOFT Software Engineering Notes, 7(5), pp96-105.

Kramer J. & Ng K. (1988); Animation of Requirements Specifications; Software - Practice and Experience, 18(8), pp749-774.

Kramer J., Finkelstein A., Ng K., Potts C. & Whitehead K. (1987);"Tool Assisted Requirements Analysis: TARA final report"; Imperial College, Dept. of Computing, Technical Report 87/18.

Kramer J., Ng K., Potts C. & Whitehead, K. (1988); Tool support for Requirements Analysis; IEE Software Engineering Journal, 3(3), pp86-96.

Maibaum T., Khosla S. & Jeremaes P. (1986); A modal [action] logic for requirements specification; Software Engineering '86; (Eds) Brown P. & Barnes D.; Peter Peregrinus.

Mullery G. (1979); CORE - a method for controlled requirements specification; Proc. 4th Int. Conf. Software Engineering; pp126-135; IEEE Comp. Soc. Press.

Ross D. (1977); Structured Analysis (SA): a language for communicating ideas; IEEE Trans. Soft. Eng., SE-3 (1), pp16-34.

Sheil B. (1984); Power Tools for Programmers; (In) Interactive Programming Environments; (Eds) Barstow D., Shrobe H. & Sandewell E.; McGraw Hill.

Stephens M. & Whitehead K. (1985); The Analyst - a workstation for analysis and design; Proc. 8th Int. Conf. Software Engineering; pp364-369; IEEE Comp. Soc. Press.

most sensible strategy in this setting is to provide a variety of powerful means of viewing and understanding such specifications. Animation is one such technique. Other effective validation techniques also aid reuse.

As mentioned, TARA provided us with considerable experience of and respect for CORE and for CORE viewpoints as a means of domain decomposition. The CORE viewpoint, essentially an agent or role, combines a domain structure with the distribution of authority for making decisions about the specification. As such it provides a powerful means of structuring requirements specification and organising requirements elicitation.

It is clear form the above comments that TARA exerted a substantial influence on our current work on ViewPoints. We believe that our notion of ViewPoints provides a sound and systematic basis for constructing and presenting methods, for managing and guiding method use, and also for the provision of tool assistance.

An additional benefit which seems to follow from the identification and encapsulation of style (representation) and work plan (specification method) in a single ViewPoint Template is the opportunity for CASE tool support. Individual support could be designed for each template in a particular method, thereby simplifying the complexity of the tool in much the same way as one expects to simplify the steps and expression of that particular ViewPoint specification. We can then envisage method tool support as comprising a configuration of template support tools, configured to suit the particular method adopted.

The work on ViewPoints reported in this article is in its early stages (Finkelstein et al 1990) and requires considerable further work. A major objective is to complement our intuitive use of ViewPoints with a comprehensive formal description. We are investigating the use of modal action logic as a suitable base for such a description.

We believe that ViewPoints provide a systematic basis for constructing and presenting methods. ViewPoints would be particularly useful in the description of mixed approaches such as those described as “multiparadigm programming” (Zave 89). The ViewPoint approach is also strongly related to Jackson’s recent work on views and implementations (Jackson 90) in which he describes “complexity in terms of separation and composition of concerns”, and focuses on the problems of coping with the relationships between concerns.

Our short term goal includes developing descriptions, in the ViewPoint style, of a repertoire of standard information systems development methods such as SSADM and JSD. This would act as a means of refining the ViewPoint concept and of illustrating the utility of the approach. In the longer term we intend to develop a ViewPoint based method for developing reconfigurable and extensible distributed systems.

7 Conclusions

The insights and experience derived from working on the TARA project have been considerable. This is a result both of the specific contributions of the work and of the increased respect we have developed for CORE as a method. In particular it has lead us to favour support for software development by methods consisting of many, relatively simple, representations tightly coupled to each other by large numbers of consistency checks. In this setting an explicit and enactable work plan provides a means for both managing the enforcement of the consistency checks and managing the consequences of redundancy.

Given a method with a work plan and with a rich collection of heuristics the method advice must be delivered to the point at which the work - the construction of the specification - is actually being carried out. The granularity of this method advice must be appropriate to the tasks being performed.

TARA has also given us a much better understanding of how people work with CASE tools. In particular we have come to realise that work is often left incomplete and inconsistent, that users move rapidly between different representations changing their minds frequently, and that analysis and validation are tightly interleaved with the construction of the specification. We believe that CASE should support this mode of work rather than attempt to constrain it.

A mundane but nonetheless significant consequence of this is the importance of efficiency and performance in CASE implementation. Raw speed in navigating around the specification, performing analysis and constructing diagrams is extremely important. This militates against CASE architectures which are centred on large (and slow) databases.

Support for reuse needs to be engineered into the representation schemes underlying a method from the start. Without such support taking advantage of existing specifications will always be difficult. The

a *domain* defines which part of the "world" delineated in the style (given that the style defines a structured representation) can be seen by the ViewPoint (for example, a lift-control system would include domains such as user, lift and controller);

a *specification*, the statements expressed in the ViewPoint's style describing its particular domain;

a *work plan*, how and in what circumstances the contents of the specification can be formulated and changed;

a *work record*, an account of the current state of the development.

As can be seen, the ViewPoint encapsulates knowledge in the form of various slots e.g. a style and a specification. The slots style and work plan represent general knowledge, in the sense that it can be applied to a wide range of problems. In contrast to this the knowledge encapsulated in the slots domain, specification and work record of a ViewPoint represent specific knowledge related to one particular problem. The specification is given in a single consistent style and describes an identified domain of the problem area. The work record describes the current state of the specification with respect to the development activities and concerns of the ViewPoint. This would include interaction between ViewPoints to transfer information and perform activities such as consistency checks.

Since a ViewPoint is also a means to express a certain perspective on a problem or system, one would like to have the ability to see or express different parts of a problem or system from the same perspective. Thus a kind of "ViewPoint type" is required which can be used as the template from which to create ViewPoints instances of that type. A *ViewPoint template* consists of the general slots of a ViewPoint, in which only the style and the work plan have been defined. A method in this setting is a set of ViewPoint templates and their relationships, together with actions governing their construction and consistency.

The "architecture" of the development process is thus a number of ViewPoints expressing partial knowledge of a system from a particular domain point of view, concentrating on a particular aspect of concern (responsibility) and at a particular stage in the development process. This "ViewPoint space" can be considered as a configuration of ViewPoints, with the relations between them expressed as interconnections. The notion of structure is fundamental. Both the internal information and the interrelationships seem to be best expressed in some organised, structural form.

The collection of all ViewPoints for a particular stage, such as design, can be considered to provide all relevant information for the design specification. This could perhaps be considered a 'horizontal cut' in the ViewPoint space.

Furthermore, we suggest that the domain inspired ViewPoints which originate at the initial requirements elicitation and specification stage are actually "stable", and that they provide a comprehensible and sound basis for viewing the information created in later stages of the process. This information may well be dispersed in many ViewPoints at these later stages. Hence, there is also a 'vertical cut' in the ViewPoint space which expresses all the stages in the process but from a single domain point of view.

6.3 Status & Experience

We believe that ViewPoints provide a basis for unifying models of the information systems development process and models of software structure. The partitioning of knowledge exemplified in the ViewPoints approach facilitates distributed development, the use of multiple representation schemes and scalability. Furthermore, the approach is general, covering all phases of the information systems development process from requirements to evolution.

not suitable for design. What appears an appropriate structure for carrying out design is not suitable for construction and reuse and so on. Because there is no single structuring approach which is wholly appropriate to all the activities in information systems development some important aspects of the development process, notably requirements engineering and system management, have been neglected.

Multiple Formal Representation Schemes: Much effort has been devoted to developing ever richer and more sophisticated formal representation schemes. On the surface this appears to be a worthwhile enterprise - if a representation scheme is made more expressive the task of elicitation and specification should, in theory, become easier. This has however not proved to be the case:

the learning overhead in the use of these schemes is significant;

the development of such schemes is extremely difficult, in particular developing sound and adequate verification or proof schemes;

such schemes are often very different from the conventional (and reasonably well understood schemes) used in information systems engineering practice and consequently pose difficulties for technology transfer;

the richer the representation scheme the easier it is to write baroque and unreadable descriptions;

although a more expressive representation scheme may still theoretically permit validation of complex properties of a description (for example, generation of consequences using formal reasoning) it generally makes simple validation by inspection more difficult, and automated aids less likely;

no single person may want, or be able to, use the full expressive power of the representation scheme.

There is an alternative to the “big language” approach. The “multiple representation” approach in which each participant is allowed to use simple "bespoke" representations for eliciting, presenting and determining properties of relevant parts of the specification world. The problem that arises from adopting this approach is that, if many different representations are used, how can potential inconsistencies and conflicts between them be detected and resolved?

6.2 Approach

We propose the use of ViewPoints as both an organising and a structuring principle in information systems development. In outline, a ViewPoint captures a particular role and responsibility performed by a participant at a particular stage of the development process. The ViewPoint must encapsulate only that aspect of the application domain relevant to the particular role, and utilise a *single* appropriate scheme to represent that knowledge.

A ViewPoint is a loosely coupled, locally managed object which encapsulates partial knowledge about the application domain, specified in a particular, suitable formal representation, and partial knowledge of the process of information systems development.

A ViewPoint is a combination of the following parts or slots:

a *style*, the representation scheme in which the ViewPoint expresses its role or view (examples of styles are data flow analysis, entity-relationship-attribute modelling, Petri nets, equational logic, and so on);

domain into viewpoints is of considerable benefit. We now describe our current work.

6.1 Objectives

Our current work is aimed at developing a new approach to information systems development. The approach, which we call the "ViewPoint approach", explicitly avoids the use of a single representation scheme or common schema. Instead, multiple ViewPoints are utilised to partition the domain information, the development method and the formal representations used to express information systems specifications. System specifications and methods are then described as configurations of related ViewPoints. This partitioning of knowledge facilitates distributed development, the use of multiple representation schemes and scalability. Furthermore, the approach is general, covering all phases of the information systems development process from requirements to evolution.

The concept of a ViewPoint is a synthesis of the concepts of "view" (partial specification) and "viewpoint" which were successfully exploited in other research projects. The TARA project provided us with considerable experience of and respect for CORE and for CORE viewpoints as a means of domain decomposition. The CORE viewpoint is closely related to agents or roles in that it takes into account the way in which authority for making decisions about the specification is distributed.

Information systems development is a complex combination of activities. It requires a knowledge of the application domain, of specification schemes and of ways that these schemes are used. The key to managing this knowledge is to structure it so as to provide a partitioned, distributable organisation for the information systems development process, and a partitioned, distributable structure for the software specification. We believe that a common partitioning and structuring is both possible and desirable.

This presents three particular challenges: finding a common structure that accommodates both software structure and the development process; finding a means of handling the different structuring approaches required at the various stages of development; finding a means of working with many representation schemes. We discuss these briefly below.

Common structure for software and the development process: Developing software-in-the-large involves many participants, with experts in various aspects of information systems development and in various aspects of the application area. In addition, each participant may have different roles, responsibilities and concerns which may change and shift as the information system develops and evolves. Participants have knowledge which they want to bring to bear on the development of the specifications. This knowledge will generally complement that of the other participants but may also overlap, interlock and conflict.

This presents us with two groups of closely related problems:

With all these participants how can we guide and organise the process of information systems development? How do we assign and maintain responsibilities?

How can we allow each participant to see only that aspect or part of the "specification world" which is relevant to that participants interests and responsibilities while preserving consistency between them

Despite the obvious relation between these groups of problems they are commonly treated separately - the first in so-called software process modelling languages, the second in specification languages. The structuring schemes employed are generally mutually incompatible.

Structuring at different stages of development: A well known difficulty, which arises with all approaches to structuring in information systems development, is that of "structural transformation". What appears an appropriate structure for carrying out requirements analysis is

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.

5.2 Approach

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 retrofitted 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 manageable in size, sufficiently information rich to offer a return over and above the cost of use and management of a reuse library, 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.

5.3 Status & Experience

A prototype tool - TRUE - to support Transaction Reuse has been developed. This tool is based on a model of reuse partly derived from artificial intelligence research on analogy. The tool is integrated with The Analyst by means of transformation and note passing tools similar to those of the Animator.

The tool contains implementations of contextual views - means of looking at the reuseable transactions through the filter of steps in the CORE analysis - and a set of the global views including class inheritance classification and browsers for synonyms and annotations. Pattern matching strategies have also been implemented these include means of using weighting and combination of weightings on pattern matches. Strategies drawn from analogical reasoning have been implemented (in a relatively simple minded way) these include generalisation based, causal chain and purpose matching. Method guidance, which controls and ties together views and strategies by prescriptive guidance on their use, is in the form of a “help” system. Although not implemented, heuristics could be integrated into the larger method guidance scheme of TARA. Allocation and restructuring, the process by which a reuseable transaction is placed in its new setting and the functionality distributed across the new viewpoint structure, has proved difficult to support. Instead the tool allows free editing and flags outstanding inconsistencies in the emerging new transaction.

A full description of TRUE appears in Finkelstein (1987).

6 ViewPoint Approach

Experience in the TARA project has exerted a substantial influence on our research work, not only from the specific details of the TARA research but also as a result of the use of the CORE method and its application to a substantial case study (Kramer et al 1987). In particular it has lead to favour support for software development by methods consisting of many, relatively simple, representations tightly coupled to each other by large numbers of consistency checks. Furthermore, the partitioning of the

the user with some basic operators to perform data processing. 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. An alternative is where *part of an action* is to be left for user decision. A simple extension to functions available provides this facility.

4.3 Status and Experience

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. Figure 11 shows a snapshot of the Animator in use, with data values on the arcs.

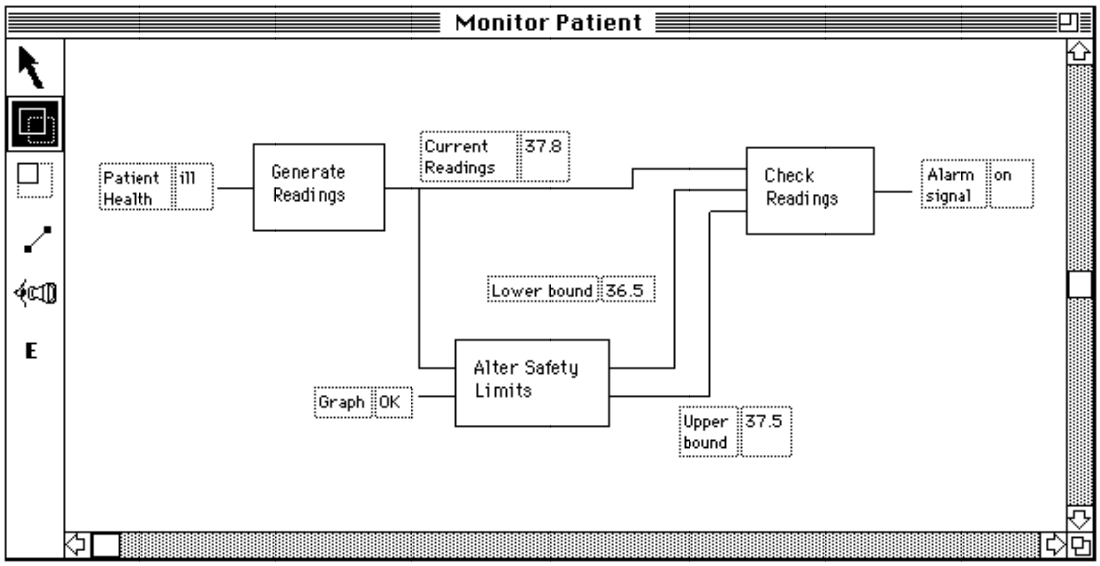

Figure 11: Animator interface.

It was intended that the animator should be integrated with The Analyst 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 implementation of The Analyst available at the time. The integration has therefore taken a weaker form by providing a transformation tool to transform The Analyst specifications into a form used by the Animator, and a mechanism for posting notes from the Animator back to The Analyst to indicate comments or changes required as a result of animation. Although the animator could be used for data flow diagrams in general, it has been designed to take into account some of the specifics of CORE, such as channel (stream) and pool (store) data flows.

A full description of the Animator appears in (Kramer & Ng 1988).

5 Reuse

5.1 Objectives

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.

4.2 Approach

Many requirements analysis methods involve, at some stage, the identification of actions and data flows within the proposed system (e.g. SADT (Ross 1977) and PSL/PSA (Teichroew & Hershey 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. The prime use of an animator is 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. 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. The Analyst specifications are transformed 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. The note passing mechanism, discussed above, provides a convenient means for transmitting the required modifications back into The Analyst.

There are several ways in which a scenario can be animated from a static action/data flow description. Our approach, which we refer to as transaction animation, is one in which 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. We also support a strategy in which 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.

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, we believe that 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

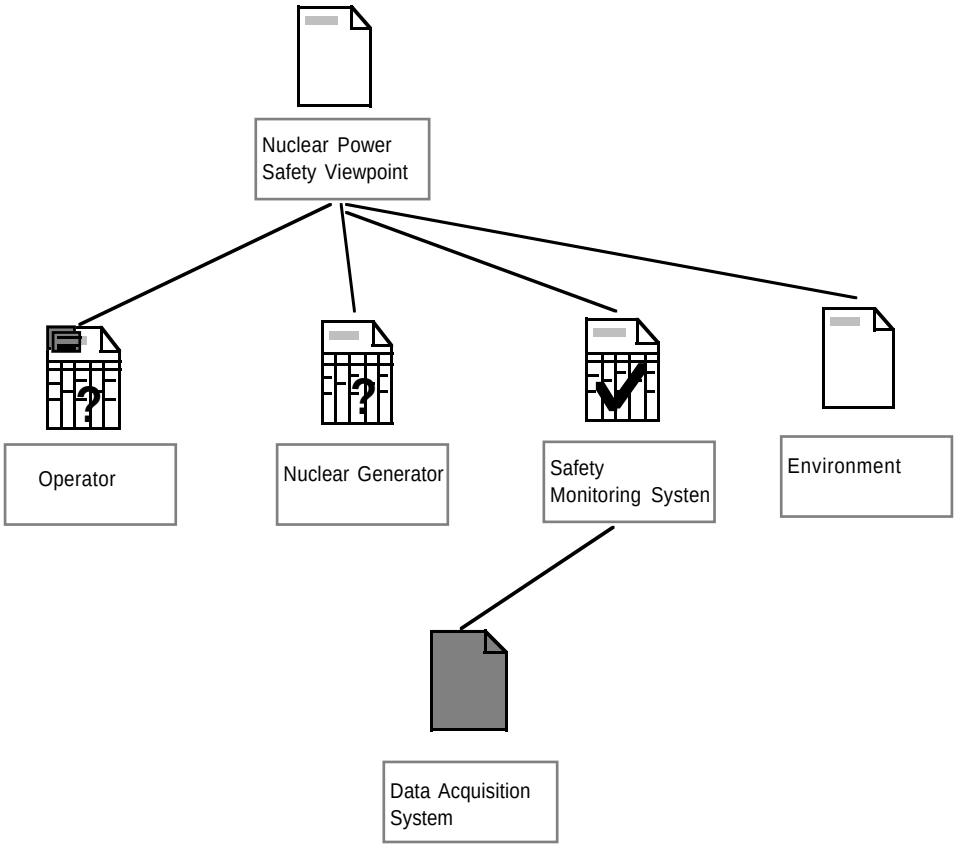

Figure 10: Current state of the Analysis.

4 Animation

4.1 Objectives

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 & Konchan 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

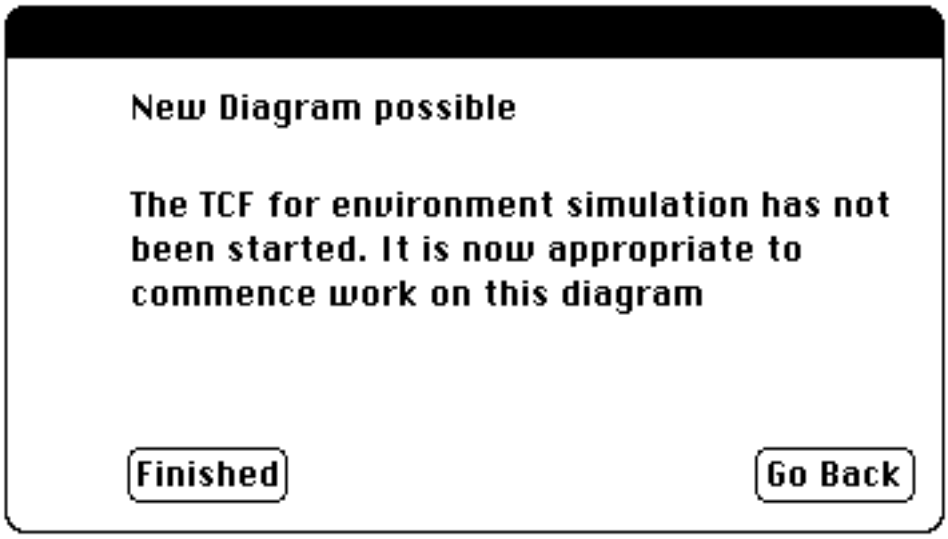

Figure 8: Example of advice

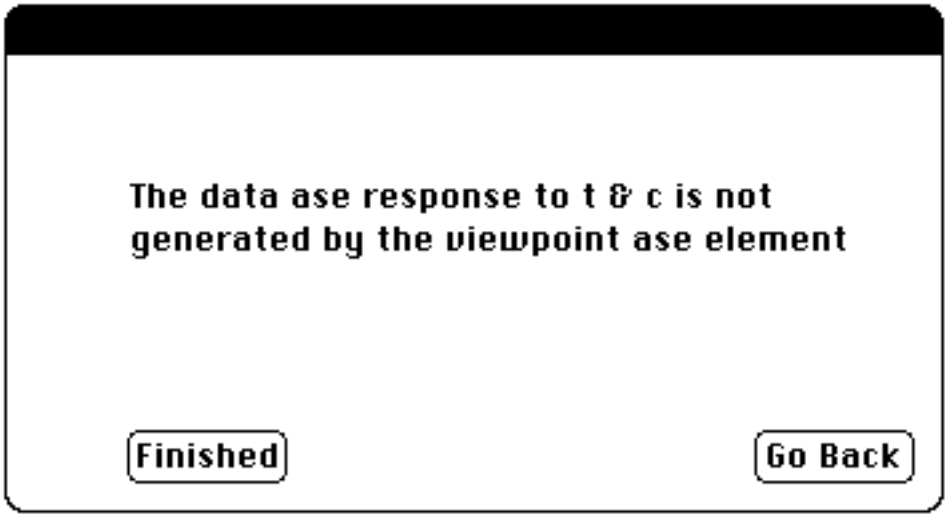

Figure 9: Example of advice

set of heuristics.

Active Guidance is generated in three stages: collection of all plausible actions generated by the normative model and the notes generated by The Analyst according to the remedial model; ordering and possibly filtering these actions by prioritisation heuristics; presentation of advice, status information and explanations to the tool-user on demand.

In order to derive suitable prioritisation heuristics several attributes have been identified on which to judge each piece of advice. A priority is assigned to each piece of advice to be presented to the user of The 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.

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). Figure 6 shows a summary of the notes outstanding, Figure 7 shows a list of the advice according to their weighting, Figures 8 and 9 show further explanation of the notes obtained on request. The user may request that a representation of the current state of the analysis is shown in graphical form (Figure 10) In this summary the type (tabular collection form, data structure diagram, SVM etc.) and status (empty, started, finished, notes attached etc.) of each diagram is given.

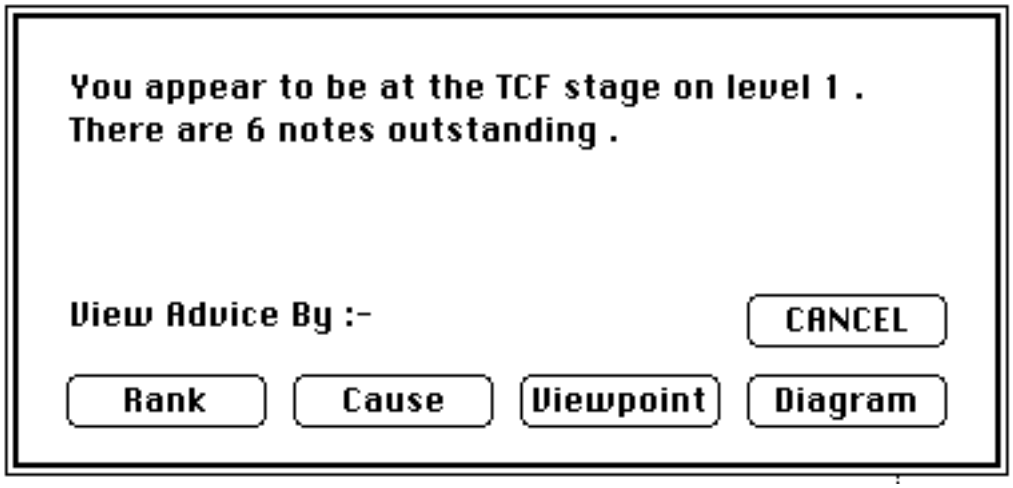

Figure 6: Summary of Notes outstanding.

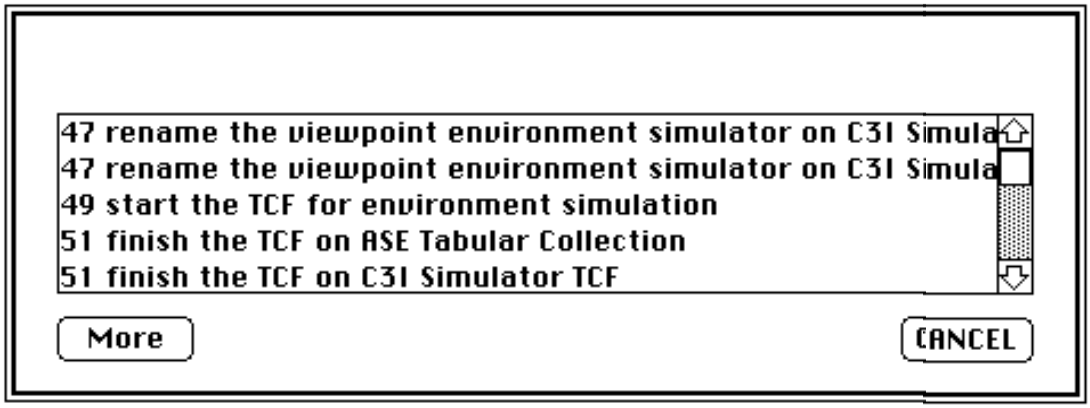

Figure 7: Prioritised list of the advice

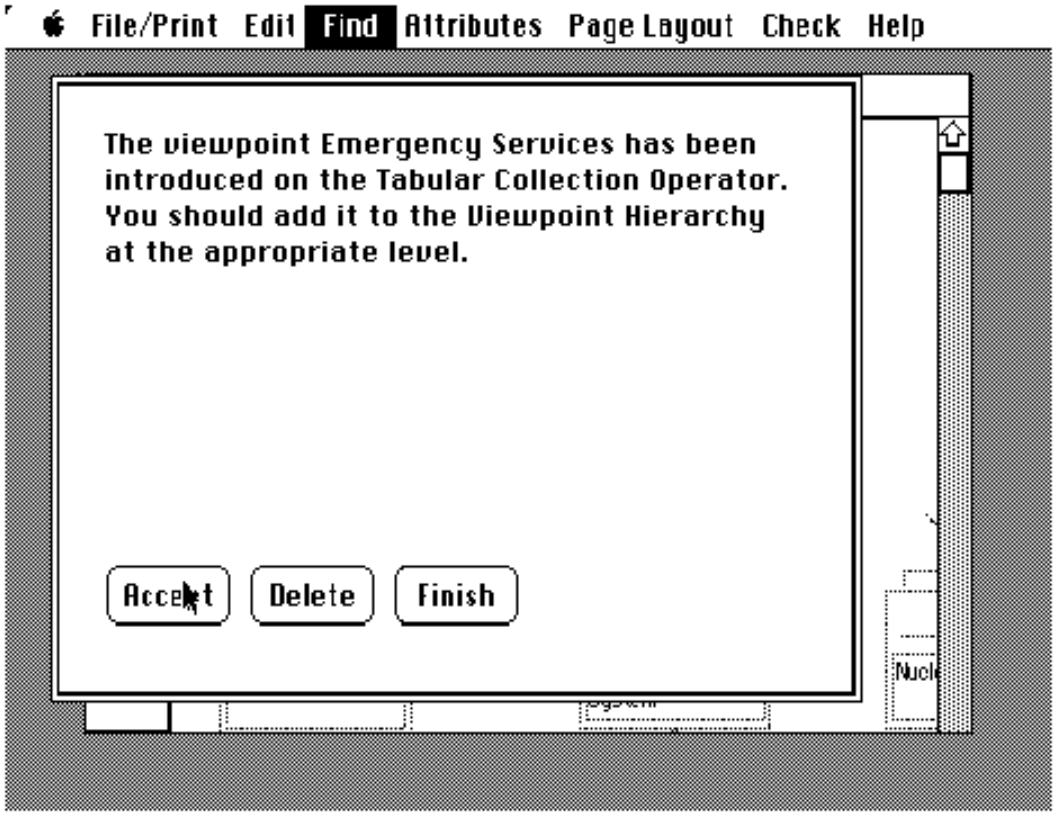

Figure 5: Typical Note contents.

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. The general classes of anomaly 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.

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 analysed 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

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 5 shows the result of opening the note.

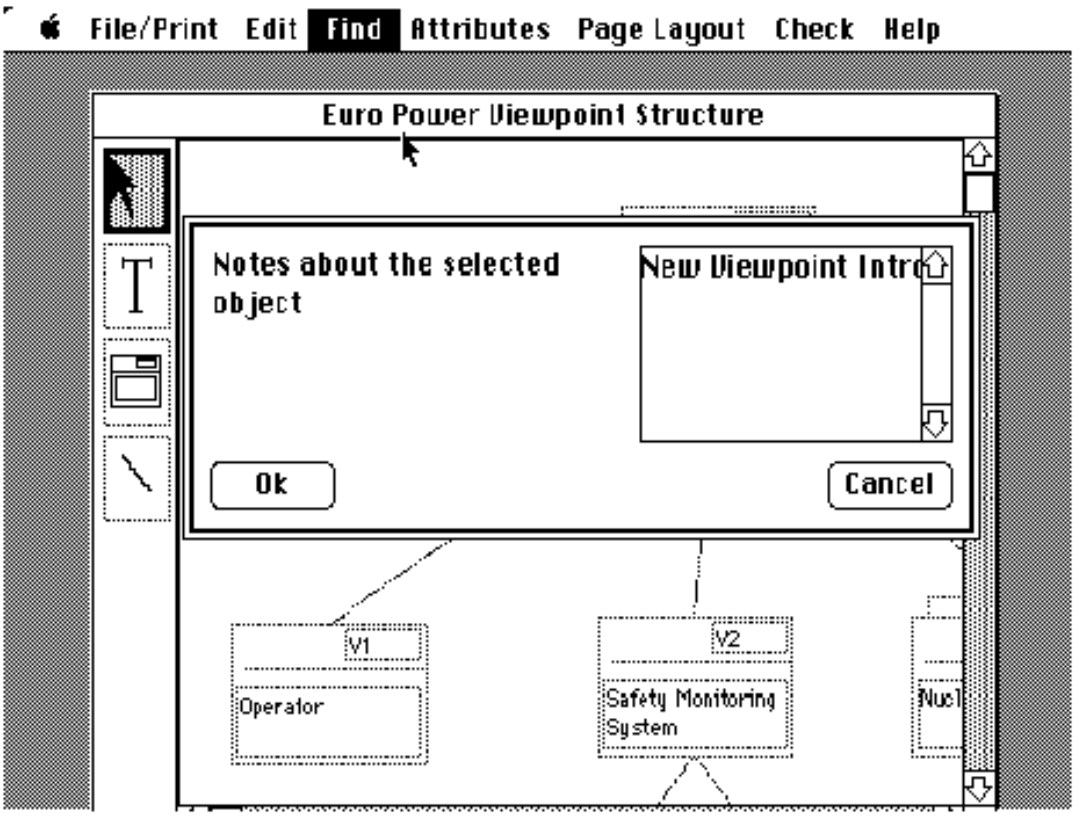

Figure 4: Note attached to a diagram

reference is attached to each of these diagrams which indicates the source of the 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 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 occured	Ammend	- 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 componants
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 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.

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. 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 4 we see what happens when the practitioner opens the viewpoint

associated with Goals.

Goals are created (activated) by events which transform the specification database into a certain state. The characteristics of this state we 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 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.

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. A prototype using such a model was constructed. Unfortunately, the rules of CORE are too context-dependent for this approach to support more than the simplest form of guidance.

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 approach, which appears a promising basis for further work, is not discussed here.

In the prototype, a more pragmatic approach was adopted: the normative model is encoded as a set of Prolog clauses. This includes the context-free grammar rules, integrity constraints such as those defining the binding of parameters, and the termination conditions for iterative steps. 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. This is discussed below.

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 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-

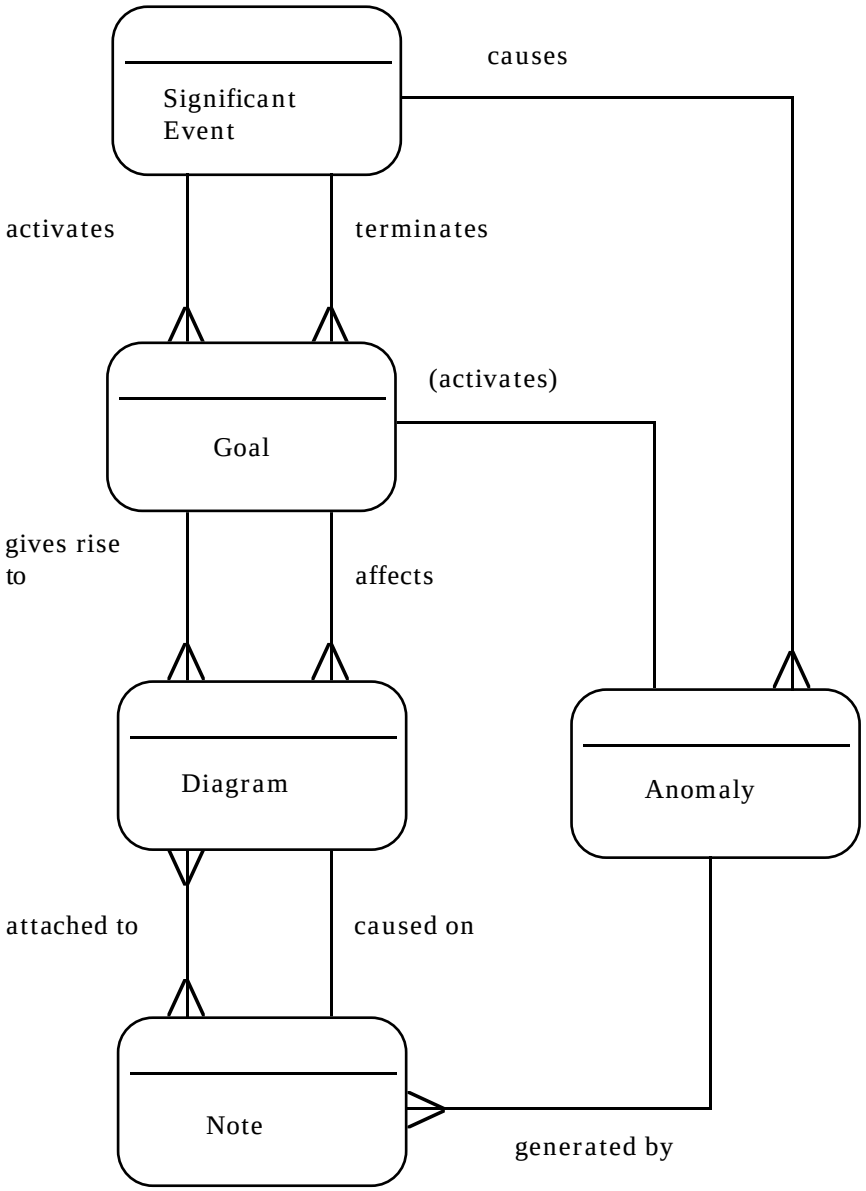

Figure 3: Data model of active guidance

Significant Event

A significant event is an action or sequence of actions by the user which invokes the method rules within The Analyst. For example, the user actions of selecting an area of text, modifying the text and deselecting the area. A Significant Event causes Anomalies to be detected when the event would result in an inconsistent state of the specification; activates a Goal when the event results in the specification satisfying the pre-conditions of the Goal; 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

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 standardised 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 organisation 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.

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, with 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.

3.3 Status and Experience

An active guidance system for CORE has been implemented in Prolog within The Analyst. This is tightly coupled to The Analyst so the user sees the combination as one system. We briefly describe the architecture of the guidance system in terms of a simple data model (Figure 3) and describe the implementation of the active guidance system.

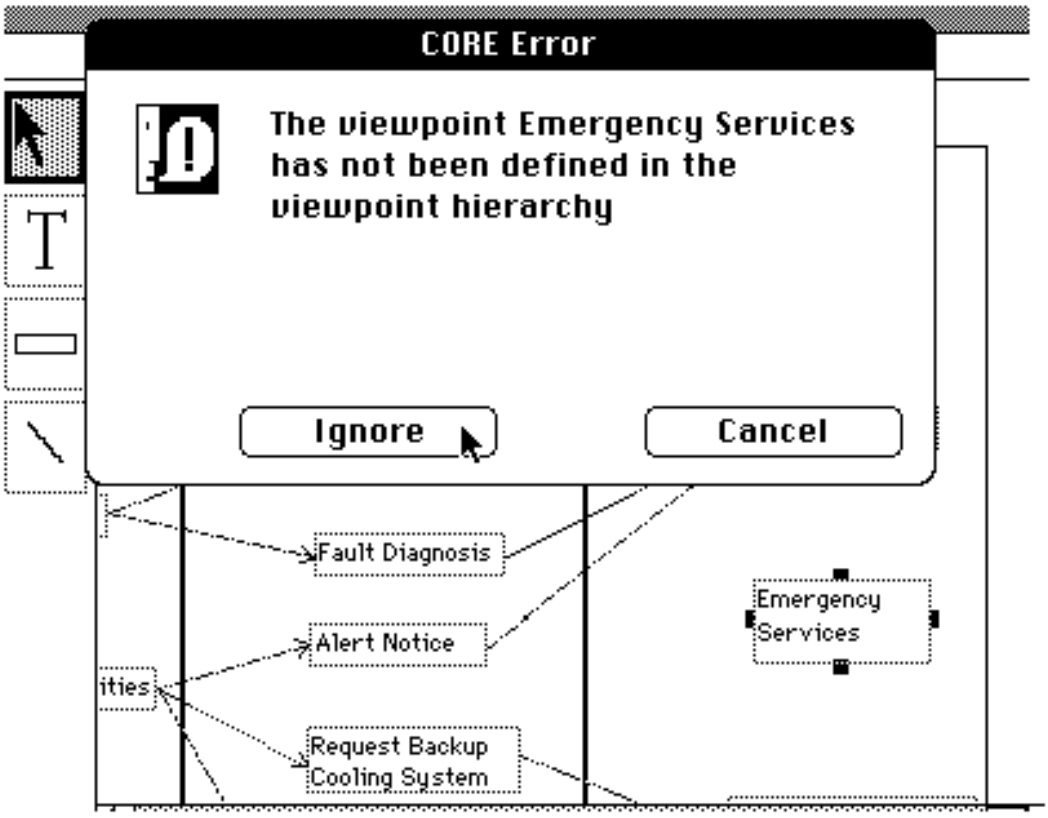

Figure 2: The Analyst in use

3 Method Guidance

3.1 Objectives

Many requirements analysis methods are in use in industry, and most practising systems analysts are familiar with one or two. However, the range of expertise 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 analysis 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 prioritisation of advice between alternative actions, such as corrective actions before method steps.

3.2 Approach

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,

Tabular collection represents a viewpoint's responsibilities 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 (other viewpoints) of the data. In data structuring, the output of each viewpoint is analysed. A diagram is produced resembling a Jackson structure diagram 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) for that specific level. 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 just 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.

2.2 The Analyst

The Analyst, developed by SD-Scicon (Stephens & 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 consistency 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 be done either continuously or on demand.

Figure 2 shows a typical snapshot of The Analyst in use. During construction of a tabular collection form a consistency check has been invoked and an error signposted.

CORE is one of the few truly prescriptive methods available. It consists of a series of steps which elucidate the user view of the services to be provided by the envisaged system and the constraints imposed by 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/data flow form of specification.

In CORE the constituent steps should be performed in a well-defined order. Figure 1 illustrates the grammar of a perfect use of the CORE method 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).

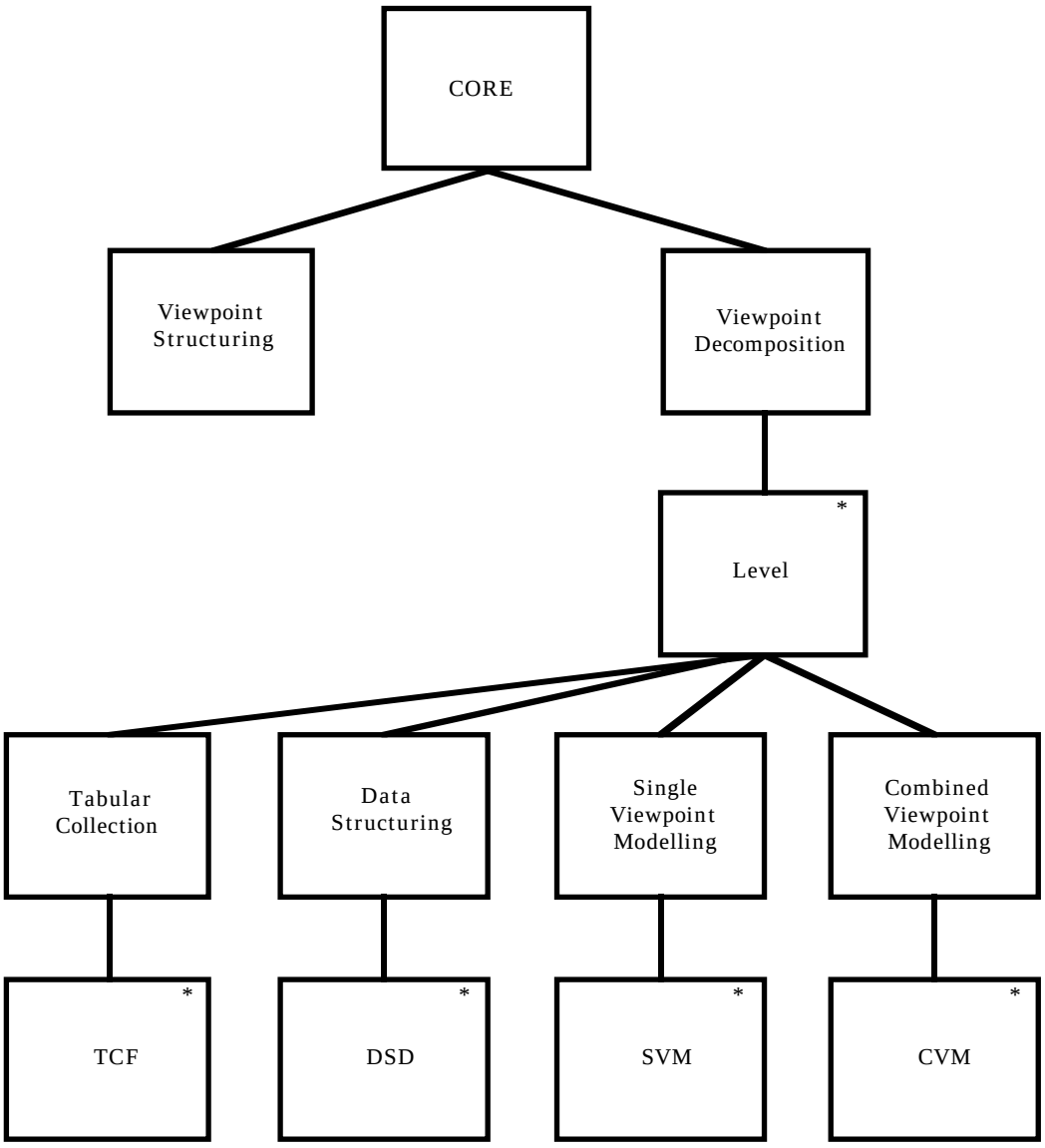

Figure 1: DSD showing the steps of CORE

In the first step, the domain of discourse is partitioned into disjoint viewpoints. These entities are organisational, human, software or hardware components of the system and its environment. Some of these are designated as indirect viewpoints, and are of interest only as sources or destinations of data. The rest are direct viewpoints, and are to be subject to further analysis. To aid the understanding of complex systems, direct viewpoints are decomposed into sub-viewpoints recursively until they represent a sufficiently simple role. The remaining steps of the method are repeated at each level of the viewpoint hierarchy.

systems development, Loucopoulos, P. & Zicari, R. (John Wiley) 1991, 413-432.

TARA: Tool Assisted Requirements Analysis

Anthony Finkelstein & Jeff Kramer
Imperial College, Department of Computing, UK

0 Abstract

The TARA Project conducted research into the provision of tool assistance for requirements analysis techniques. In particular it concentrated on automated support for three specific areas: active method guidance, requirements animation and the reuse of specification fragments. In this article we discuss the aims and status of TARA and the application of CASE technology within a method framework. In addition, we outline work on specification and method integration which is based on some of the approaches developed within TARA.

1 Introduction

Requirements analysis is one of the most critical tasks in information systems development. Unrecognised 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. The main focus of our work was the large class of systems which can be classed as "real-time information systems"; that is systems which must satisfy temporal constraints and are also data rich. Examples of such systems are: military command, control and intelligence systems; trading and financial information systems; hospital patient monitoring systems.

The particular objective of the TARA (Tool Assisted Requirements Analysis) project was to examine three important extensions to current CASE technology in the area of requirements analysis.We were interested in the role of automatically provided *method guidance* to support the use of requirements analysis methods, the ability to use software tools to help clients and analysts visualize the behaviour of the specified system by *animation* of the specification, and the possibility of supporting the *reuse* of specification fragments or parts of existing specifications in the composition of a new specification.

The intention was not to construct another diagram editor and requirements specification technique. Hence we adopted as a base for this work an existing, widely used, requirements analysis method - CORE - and a CASE tool for diagram construction and consistency checking - The Analyst.

An earlier paper (Kramer et al 1988) provided an incomplete and preliminary view of the TARA project. This article provides the first comprehensive description and discussion of the TARA work and its contribution. We first outline CORE and The Analyst, and then present and discuss the work in each of the three areas of concern. In a concluding section we outline a new approach to information systems development, based in large part on insights gained from the TARA project, which explicitly avoids the use of a single representation scheme or common schema. Instead, multiple "ViewPoints" are utilised to partition the domain information, the development method and the formal representations used to express information systems specifications.

2 Background

2.1 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). We assume that the reader is familiar with the spirit of formatted requirements analysis methods. The following brief summary of CORE specifics should be sufficient for the purposes of this article.