DESIGN METHOD AND MANAGEMENT UTILITY ENABLING THE CONCURRENT EXERCISE OF DISTRIBUTED EXPERTISE

by

Andreas Stylianos Tsiotsias

James Watt Engineering Laboratories
Department of Mechanical Engineering
University of Glasgow
Scotland

Submitted to the University of Glasgow
for the degree of Doctor of Philosophy

September 1994

Thesis
10090
Copy 2

Dedicated to Audrey and the remembrance of Stelios.

"The elements of design are the elements"

Abstract

Concurrency of engineering activities requires a utility allowing designers, working at all phases of design to: communicate the design requirements to specialists and external technologists, elicit responses and integrate the resulting actions with the design solution; acquire resources which are functionally and geographically distributed; communicate a formally agreed product description to the collaborating agents. The creation of such a utility is presented here which employs techniques of knowledge engineering to represent the entities and methods used in design. The utility manages representations within existing standards and methods, including communication at interfaces, resolves constraint conflict during design by referring dependency relationships, is unitary and can be made recursive in its operation.

The Glasgow Utility for the Integration of Design (GUIDE) employs the methods of knowledge engineering to secure a basis for design by a multidisciplinary team, the membership of which may be distributed and will vary as the product emerges through successive design phases. GUIDE offers designers a range of design functions which may be applied to the task performed through a single interface and without operational prescription. GUIDE maintains a single product description, which includes integrally a record of the entire design activity. It also provides distributed data base access and communications facilities.

GUIDE employs a representation scheme which involves *structures*, *atoms* and *methods* as its elements. Additional characteristics have been invested in these elements to provide for their manipulation and control. With GUIDE and the tools it provides designers can create graphical, data and information related working entities and involve active processes. Process entities may invoke proprietary tools, provide translation at their interfaces and sustain the required communication with various engineering and product centred data bases. Operations on design entities and information generation processes are managed by control functions which can also cause data transformations. GUIDE has the capacity to aggregate generic, modularly defined knowledge representations to create higher level, formally constructed unique design solutions or part solutions and to manage associations between design entities and the constraints affecting them.

GUIDE's design record - the route taken and that structured information generated during design - provides a mechanism for the accumulation of expertise which can be used in future designs. In addition to the actual outputs of a design, such as the part description in its various forms, a designer could obtain information concerning the design tasks undertaken and their sequence. The design record enables design traceability and audit of the design process, sustains status evaluations and provides for regression.

The concepts, design and implementation of GUIDE are described. Three examples are used to illustrate GUIDE's capacity to support the operations of design teams, the constant availability of a multidimensional product model which exposes tasks more quickly and precisely and the ability logically to collocate design teams through product model coincidence.

GUIDE provides an extension to knowledge representation using frames through the characteristics of the elements it employs and by the way its control mechanism manages operations upon and communication between them. Links formed between elements and between elements and methods can be described in a structured way. Constraints are represented as methods which can evolve over time and may influence the use of other GUIDE elements. Relational data bases are used to hold the knowledge representations employed and GUIDE exploits the relational architecture to physically distribute the representations and maintain their integrity. The design record contains comprehensive meta-knowledge and supports the abstraction of formal generic representations from specific instances.

Acknowledgements

My greatest debt is to Professor Brian F. Scott, for his continuous encouragement, scientific advice, discussion of the topics contained and for making available the facilities of the Department of Mechanical Engineering where the work was conducted.

Appreciation is expressed to Messrs Tony Munton and Alan Crombie for much helpful discussion on the industrial relevance of the topics described in the work, for arranging funding from IBM, on three occasions, in support of the author's research ideas and for obtaining permission from their organisation for the inclusion in this work of IBM materials, descriptions of procedures and registered product information.

Thanks are also due to Miss Isabelle Lawson for her help with the illustrations and in preparing the manuscript and to my team, Messrs Douglas Muir, Karl Scriba, Alexander Skandalakis and, particularly, Walter Robinson for suffering my dictatorial terms and delivering, to the highest professional standard, the frequent 'final' versions of GUIDE code.

I have dedicated this work to my nearest and dearest. Audrey, my partner, for her support, tolerating a hectic work schedule and providing the impetus for concluding this work. Stelios, my late father, for teaching me to persevere and be professional and for his encouragement, to his last days, properly to conduct and conclude this work. Audrey and Stelios, this work is as much your achievement as it is mine.

Gilmorehill, Glasgow A.S.T.
August 1994

Contents

Figures

Tables

Plates

Design Data And Methods

1.1 Introduction

The following quotations, taken from reports by captains of international engineering bodies, provide a clear statement on the issues which influence operations within their organisations:

"At first sight, the idea of any rules or principles being superimposed on the creative mind seems more likely to hinder than to help, but this is quite untrue in practice. Disciplined thinking focuses inspiration rather than blinkers it" **[Glegg, 1969]**

"The project was large enough and communication poor enough to prompt many members of the team to see themselves as contestants, rather than as builders making products; this breakdown in orientation and communication is a major hazard in large projects" **[Brooks, 1975]**

"...the management often lacks understanding of the field (of information management), which leads to a lack of results and thus to a situation where the field remains a hobby for the interested designer, rather than a tactical management tool" **[Brooks, 1975]**

"We must exchange information freely so that mistakes are not made twice and the best ideas can be retained and built upon" **[IMechE, 1988]**

"...our (company's) wealth of expertise, know-how and experience with products and markets lies no further than the Corporate database; we must employ new ways to manage and communicate this information amongst our engineers and designers if we are to retain a competitive edge" **[Arnold, 1993]**

"...the (software) tools are becoming increasingly sophisticated, able to generate vast amounts of accurate, vital product development information..." **[Anonymous]**

"Engineering data banks must be created for access to all functions and stages of plant life. Computer aided design/computer aided engineering can then draw on vast, proven experience in creating new products" **[IMechE, 1988]**

The force of these statements is clear. The information stored by companies comprises more than merely data: it is a reflection of the expertise which the company possesses; it represents the results of the processes engaged in the development of products; it describes the technologies employed in records of the constraints or enabling functions which were introduced; it includes models of products and of company operations which generate them. Mechanisms must be employed to secure the information's structure, communication, management and maintenance, whilst ensuring its timely delivery, appropriately to the requesting authority without prescription.

1.2 Engineering information

Engineering information is generated whenever the agents of an enterprise engage a task. Manifestations of it are encountered in paper-based and electronic forms. The electronic collection comprises data which are stored as operating system files, proprietary-access indexed file systems and, sometimes, in relational data bases as sets of tuples in one or more tables.

The last form has been employed[1] by engineering companies to structure their peculiar information base by exploiting the capacity of database management systems to represent relationships between items of data. Relationships link business functions such as marketing, finance, design, the technology base and manufacturing. Emphasis has been placed on structuring data for the purposes of its storage and on the definition of schemata for its efficient retrieval and, in some instances, simultaneous access by a group of agents or applications. These efforts have culminated in, *inter alia*, the application to Computer Integrated Manufacture (CIM).

1.2.1 Constitution and structure

Stored information frequently remains as a polymorphous collection of diverse[2] data sets. Several factors contribute and stem from the 'historical baggage' associated with the operations of a company and the external or internal constraints imposed. Figure 1 provides an example of information structuring in a multi-national, multi-product company.

Data collection is necessary whenever specific tools are exercised. The data format depends upon either the facilities provided by the tools themselves, in which case their structure and content is beyond the control of the company, or constraining guidelines introduced by the company to adapt the data structure to its peculiar requirements. The latter usually involves transformation of data during its storage and retrieval. Each 'system' may handle its respective functions and data forms with its own management facilities efficiently, but due to

[1]A recent report indicates that, in terms of world-wide corporate data, a mere 20% is stored in some form of either relational or object-oriented databases. The remaining data are in either flat or indexed files which require, respectively, operating system specific or proprietary access mechanisms.**[Arnold, 1993]**

[2]Diverse in character; composed of diverse elements; heterogeneous.

incompatibilities in the individual data models, only a limited flow occurs between them. **[Sharad, 1991]**

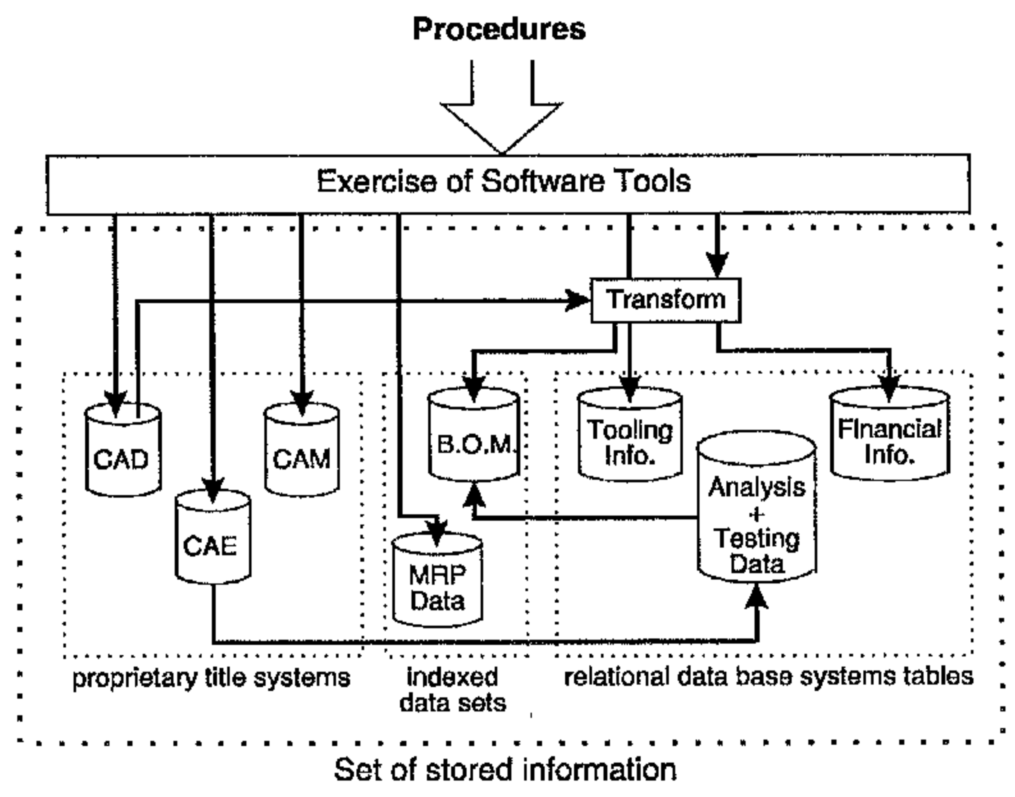

Procedures
Exercise of Software Tools
Transform
CAD
CAM
CAE
B.O.M.
MRP Data
Tooling Info.
Analysis + Testing Data
Financial Info.
proprietary title systems
indexed data sets
relational data base systems tables
Set of stored information

Figure 1 : Heterogenous nature of an enterprise's information base

Data is frequently duplicated in heterogeneous environments. The information is stored in a 'raw' form which does not lend itself to communication in a standard format; communication between and comprehension of the information by the collaborating centres is, therefore, inhibited. The consistency of the shared information cannot be maintained in the presence of change.

The engineering information base includes data which describe the physical product. Additionally, a record of the constraints which were applied during the product design is kept which may include records of the inputs to each of the design steps or pointers to alternative product data. This record identifies the contributors of product related technologies and of the methods associated with them; furthermore, it contains knowledge of past and current practices and information on standards and public-domain data employed.

1.2.2 Information stereotypes

The utility of information to the enterprise for the purposes of design and manufacture, including communication with external contractors, depends upon the effectiveness with which it is represented by formal structures and descriptions. In most cases, a company's data collection predates modelling activities it might engage. Frequently, the data are held in autonomous information islands.

Stereotypes[3] - structured pieces of knowledge about entities, processes[4] and functions - which are widely recognised and understood can be a powerful aid in creating order and for modelling. They can be used to sort and classify the knowledge that can be found in data by providing a description of information generation processes defining their input and output data flows.

Manifestations of stereotypes implicit to the information base of enterprises may not be recognised by the company or be made explicit by the methods they use to aggregate data. Descriptions of entities and processes must be constructed from the instantiated data and their expression through stereotypes is a powerful option capable of relating processes and entities they use by identifying common data values, following data transformation. These relationships provide a basis for modelling any constraints affecting processes.

Stereotypes which can describe a hole in a component are shown in figure 2 by way of illustration.

Aggregations of mutually constraining or otherwise related processes and entities can, when they are expressed as stereotypes, form higher level structures. Such aggregations can represent enabling technologies incorporating the expertise[5] of the company. They may serve as templates for the generation of new descriptions

[3]The Oxford Concise Dictionary defines **stereotypes** as: *"printing plate cast from a papier-mâché or other mould of a piece of printing composed in movable type"* and the term **to stereotype** as: *"to make unchangeable; to impart monotonous regularity to; to fix in all details; to formalise"*.

[4]A **process** is the course of actions to be taken in any particular stage of product development. A **procedure** is a process 'cross-section' and denotes the series of actions to be conducted in a specific order or manner. A **method** is a formalisation or implementation of a procedure.

[5]The company's 'know-how'; 'beaten tracks' in prosecuting a task; engineering practice; an appropriate method for combining and using a set of processes and managing the information which they generate.

and allow for retrospective analysis by a company of its operations, practices and products.

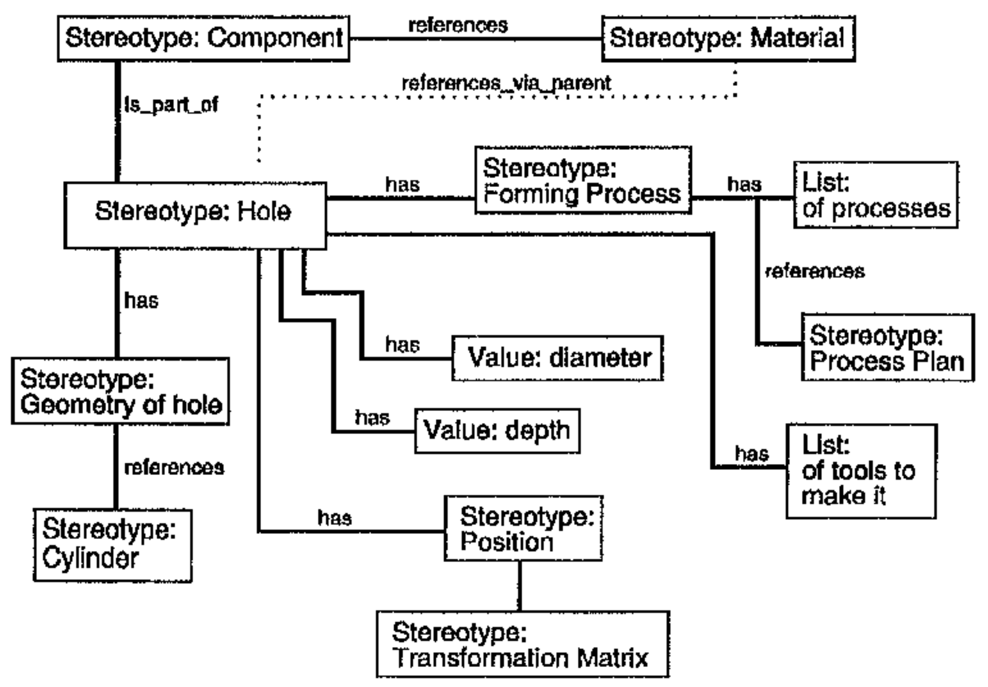


Figure 2: Stereotypes used to describe a hole in a component

Stereotypes can be used to structure the company information base and describe operations upon it because they are modular and can be made semantically contiguous. Stereotypes can be made available to agents of the company for use without additional prescription or knowledge of the processes involved. The challenge lies in the abstraction of accurate, generic definitions from instances of data and the data generating tools.

1.2.3 Product descriptions

Products are represented as structured collections of physical entities related through interfaces defining, for example, component/assembly combinations. Frequently product descriptions are expressions of the enterprise's needs determined by practice or reflect the structure of a family of products. In many instances the peculiar facilities offered by the software tools employed in design provide the data base for product descriptions by default.

These internal classifications are structured in an *ad hoc*[6] fashion. The resulting models are weak because they:

- do not allow for a retrospective evaluation of the products or of the technologies used to generate them[7] and do not support design iterations well;
- are structurally rigid and an inadequate basis for product descriptions;
- cannot readily be communicated within the organisation or between the company and its external contractors because of the disparate data structures employed;
- adapt to change in the product composition poorly because they rely upon customised interfaces being maintained with the data.

Fundamental weaknesses of these *ad hoc* methods are the static nature of representations and the dependency of data models on the peculiar tools or facilities which generate them. The implicit assumption that the representations/tools will neither be changed nor extended results in outdated product descriptions, inaccessibility of information and product models having historical rather than active design relevance[8].

Stereotypes, through their capacity to model entities, their relationships and the processes which generate them, are robust representations which accommodate change readily. Comprehensive and unambiguous product descriptions can be constituted which include knowledge of the technologies contributing to their development. Product descriptions so enriched can be used, in whole or in part, in

[6]The term *ad hoc* here refers to the origin of the product information classification method: this can be historical, product specific, tool specific, or a combination of these. The term is used to emphasise the short term representational stability which, it is the author's opinion, is enjoyed by a company before a reorganisation is required.

[7]The product data are seen as being derivatives of the generating technologies: product data modifications can be easily performed only when the technology is retained; use of a different technology implies a consequent data transformation. This is in contrast to the idea where a technological method is chosen out of a set of possible options which were compiled out of the requirement to generate or modify product information: the technological set of options can be recompiled by direct reference to the need to generate information; the data are thereby made technology independent.

[8]The representational difficulties encountered by enterprises are exemplified in the frequency with which they change their product model and the associated 'data transfer' activities and the frequency with which past solutions or techniques are employed in the design of new products.

the composition of modified or new products. High level descriptions of varying complexity and which are capable of representing the whole or a part of a product can be constructed from combinations of stereotypes with great economy. The resulting descriptions constitute a Product Model[9] capable of sustaining multiple views of the product needed by the enterprise. The capacity of stereotypes to change in time without loss of earlier definitions ensures that product models are dynamic. In communication, it is necessary only to devise a transfer medium suitable for the stereotypes rather than for the data from which they are instantiated. The communication mechanisms can thereby be decoupled from the form of the product description. Product descriptions built upon stereotypical definitions are more likely to be rich in content and easily transmitted.

1.3 Design process

The essence of design is the specification and analysis of a complex problem into a number of simpler ones which individuals or teams can tackle. This involves the transformation of an ill defined requirement into a set of tasks which add detail and raise the finesse of the product description.

Figure 3 depicts this concept. The design requirement is expressed as a set of objectives which the solution must meet. These initial objectives may be in conflict and incompletely or inconsistently expressed. As a result, a large part of the design process is devoted to discovering the nature and scope of the task set by the initial specification. Particular elements of the specification may suggest certain features of solutions, but these solutions in turn create new problems. This leads to the process becoming a set of decomposed problem solving activities. As a result, design problems are full of uncertainties about objectives and their relative priorities; objectives and priorities are likely to change as solutions to the individual problems emerge.

It is rare for any part of the design solution to serve a single purpose; it is frequently necessary to devise a solution which satisfies a range of different requirements and the designer cannot optimise around one requirement without constraining the span of the solution **[Lawson, 1980]**. The fundamental objective is to understand the structure of the problem and analyse the relationships between the design decisions which define the solution **[Logan et al, 1991]**.

[9]The **Product Model** is considered to be an accurate description of a set of interacting components with relationships between them directly referencing physical entities or systems.

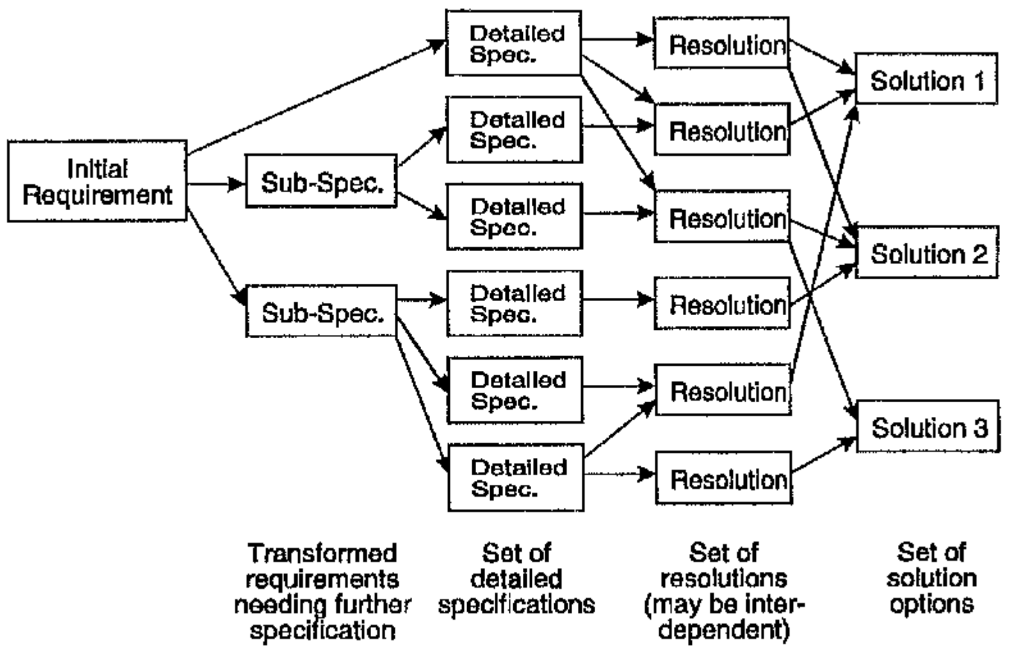


Figure 3 : Specification and analysis for problem solving

The formulation of the design problem at any stage is not final; rather, it reflects the designer's current understanding of the problem **[Inui et al, 1991]**. The analysis of problems and the synthesis of solutions are seen as merging together rather than as being distinct, one following from the other. The design is explored through a series of attempts to create solutions and to understand their implications in terms of the design **[Simons, 1973]**. The major effort in design is directed towards the understanding and structuring of problems. Only a fraction of it is devoted to solving problems, once they have been structured **[Simons, 1970]**. The design process involves the discovery of a framework to underpin information about problem structures which will ultimately be valuable in developing possible solutions.

1.3.1 A framework for design

The division of a large task into smaller tasks is a fundamental problem solving paradigm. The performance of this division requires an insightful model **[Eppinger, 1991]** of the design process. Several descriptions of the design process have been proposed which are not easily comparable, except in the widest sense.

Analyses of the design operations, the knowledge employed and the data generation methods can however be used to formulate frameworks for design.

Traditionally, operations carried out within the process of design have been prosecuted in a sequential fashion. A series of discrete steps, inter-related via constraints for process control is followed. Simple feedback mechanisms provide for control and cause iterations which increase the detail of the solution **[Hayes-Roth, 1983]**. This method creates a serial 'checklist' for design operations and while useful for routine[10] or parametric[11] design, it does not lend itself to situations where a design plan for the solution of the problem does not exist[12] *a priori* or where the design comprises a unique combination[13] of existing components.

A more flexible view of design activity is to be found in the framework **[Scott, 1988]** shown in figure 4. This is adapted here as a reference model to describe processes and identify their characteristics. In this model design is initiated by the communication of the 'intent' by the designer to engage a particular task through the Design Engine module.

The chief designer exercises all of the primary management functions, including team construction, during the process of addressing the design requirement. The process of problem solving is engaged in the central feedback control loop and is achieved by contributions, including evaluation and auditing functions, made by the elements (blocks) on this loop. These blocks contain people and physical resources exercising peculiar software and computer facilities concurrently and, possibly, not at the same physical location.

The ability to control, organise and manage distributed data bases and knowledge representations, manifested in the process models as global engineering and manufacturing standards, integrated technology descriptions, sourced component descriptions and past design product models is a vital component of the framework.

[10]A plan for the solution exists *a priori*. The subparts and alternatives are known in advance, perhaps as the result of an earlier creative or innovative design process.

[11]Implying, sometimes, redesign; an existing design plan is used with modified parameters to reflect changes in the original requirements.

[12]Creative design

[13]Innovative design.

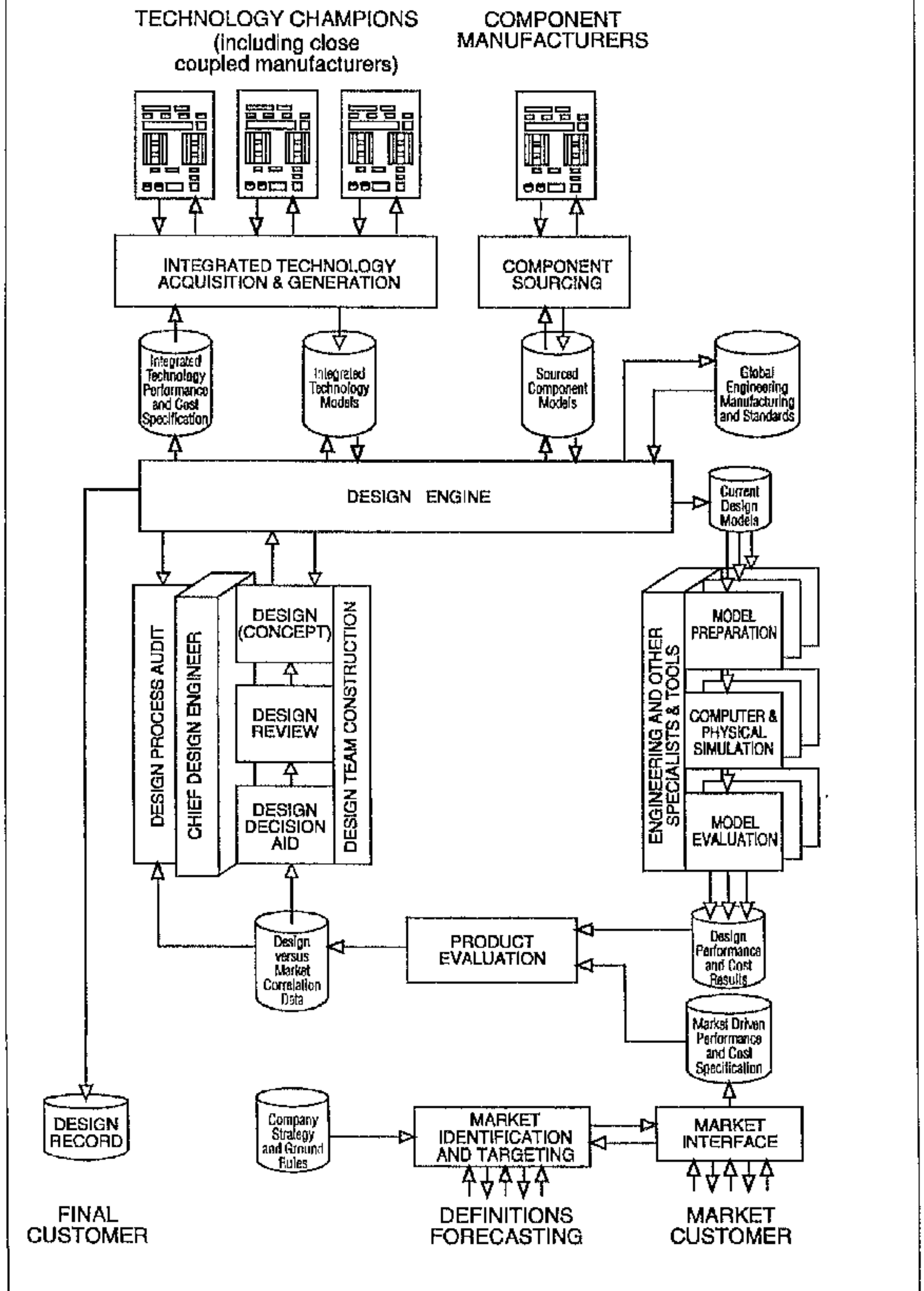

TECHNOLOGY CHAMPIONS (including close coupled manufacturers)
COMPONENT MANUFACTURERS
INTEGRATED TECHNOLOGY ACQUISITION & GENERATION
COMPONENT SOURCING
Integrated Technology Performance and Cost Specification
Integrated Technology Models
Sourced Component Models
Global Engineering Manufacturing and Standards
DESIGN ENGINE
Current Design Models
DESIGN PROCESS AUDIT
CHIEF DESIGN ENGINEER
DESIGN (CONCEPT)
DESIGN REVIEW
DESIGN DECISION AID
DESIGN TEAM CONSTRUCTION
ENGINEERING AND OTHER SPECIALISTS & TOOLS
MODEL PREPARATION
COMPUTER & PHYSICAL SIMULATION
MODEL EVALUATION
Design versus Market Correlation Data
PRODUCT EVALUATION
Design Performance and Cost Results
Market Driven Performance and Cost Specification
DESIGN RECORD
Company Strategy and Ground Rules
MARKET IDENTIFICATION AND TARGETING
MARKET INTERFACE
FINAL CUSTOMER
DEFINITIONS FORECASTING
MARKET CUSTOMER

Figure 4 : Framework for design

Implicit in the framework is the belief that the problem specification and analysis is not retained exclusively by the design authority. Contributions are made by several agents. Some of these are teams of technological champions, tasked with the acquisition of knowledge about methods and formalisation of solutions to specific problems. In 'Extended Enterprises'[14] the contributions may be made by organisations which specialise in the generation of complete product solutions, offered externally as components. The contributions made will usually lead to task segmentation and, therefore, the framework will be used in an essentially recursive manner, leading to its simultaneous exercise by several authorities having different functions and objectives and occupying different levels in the company.

1.3.2 Distributed problem solving

Explicit in the framework is the idea of distributed problem solving. This is the cooperative solution of problems by a decentralised collection of knowledgeable agents situated in a number of distinct locations **[Smith et al, 1981]**. The agents cooperate in the sense that no one of them has sufficient information or the capacity to solve the entire problem; mutual sharing of information is necessary to allow the group as a whole to produce an answer. Cooperation[15] of the individual agents in a substantive manner requires:

- that the knowledgeable agents must employ uniform[16] representations of the information which is being generated; if common descriptions of the processes are employed, the information which is generated can be communicated without the need for expensive transformations;

- the conversion of the processes engaged to an elemental or, unitary, form.

Figure 5 shows a schematic of such a process. Unitary processes perform a completely contained function, under external control, and have the ability to become process controllers in their own right.

[14]The term **Extended Enterprise** refers to the company and its external contractors sharing product design and development responsibility.

[15]This interaction is the basis of concurrent operations.

[16]Through the provision of a framework and a discipline to the agents so that teams collaborate in a natural way rather than by forcing the agents to create their own discipline and methods.

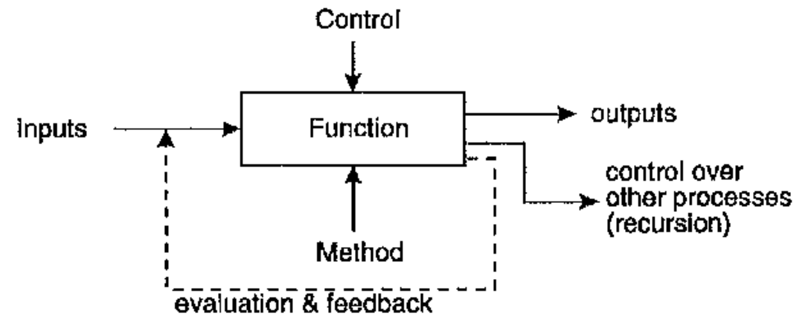


Figure 5 : Unitary process operation

- the situation when multiple agents compete for the same resource in order to achieve similar goals must be identified and controlled by an overall authority, perhaps the agent initiating the process in the first instance;
- result sharing; this situation arises when sub-problems cannot be solved by independent experts working alone; the agents should be able to report to each other the partial results they have obtained during execution of individual tasks.

In the framework, problem partitioning and, consequently, task sharing is determined by reference to the design data being generated and records of previous design activity. The constant availability of a multi-dimensional product model exposes tasks more quickly and precisely. Stereotypes may be employed to hold information and process descriptions. These last two items require a mechanism for process control and management, including the communication of data and active filtering which might be required, between the processes. These functions are provided by the 'Design Engine' in the framework.

1.3.3 Design engine

The Design Engine is responsible for the communication of the intent to act to designers and the company more generally and carries the results of actions to the requesting authority. Its primary function is to aid designers by providing communication and the facilities which will enable the analysis of a problem into component parts and the synthesis of contributing solutions into a set of design options. The former includes the ability to create sets of sub-tasks and communicate them to other engineering or design authorities for resolution.

Figure 6 presents the Design Engine function as a unitary process. To enable its exercise the issues of:

- control,
- constraint representation,
- information sourcing and communication, and
- method selection and invocation,

must be resolved and managed appropriately.

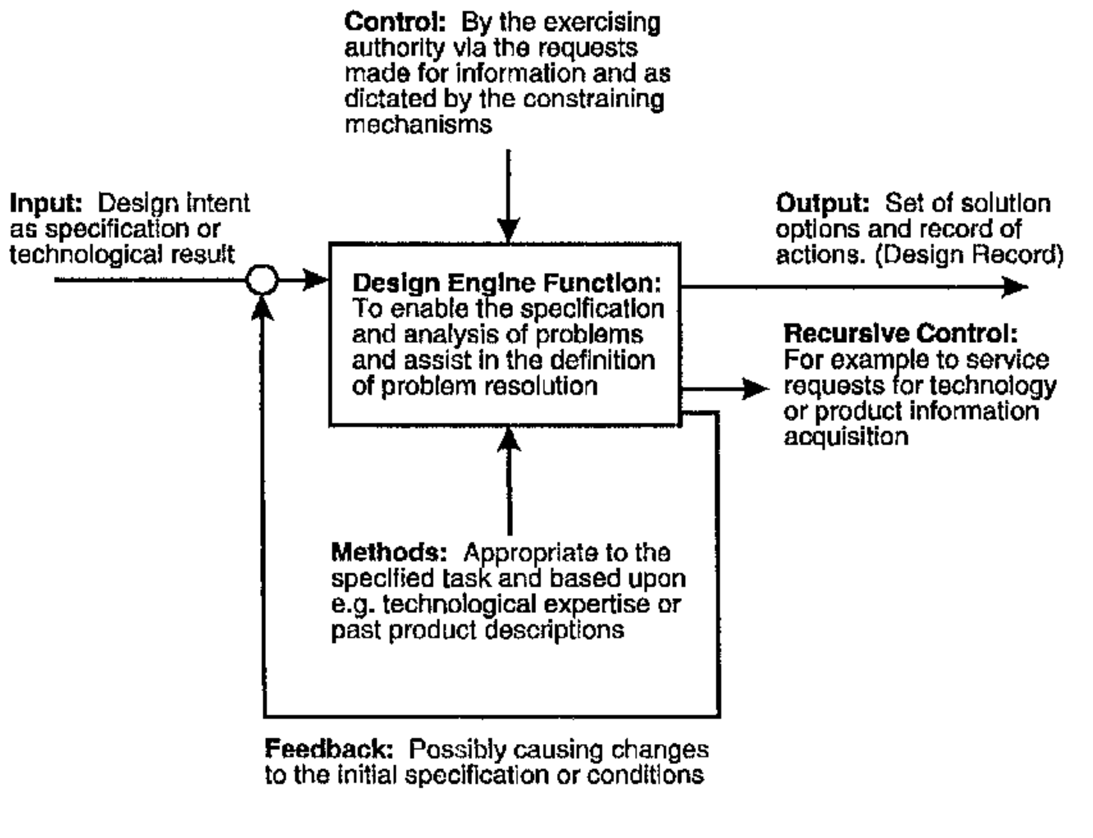

Control: By the exercising authority via the requests made for information and as dictated by the constraining mechanisms
Input: Design intent as specification or technological result
Output: Set of solution options and record of actions. (Design Record)
Design Engine Function: To enable the specification and analysis of problems and assist in the definition of problem resolution
Recursive Control: For example to service requests for technology or product information acquisition
Methods: Appropriate to the specified task and based upon e.g. technological expertise or past product descriptions
Feedback: Possibly causing changes to the initial specification or conditions

Figure 6 : Design Engine as a unitary process

1.3.3.1 Constraints

Constraints in design limit the range of variables which may be attached to processes associated with design. Constraints may be expressed qualitatively or quantitatively and may be continuous or discrete. Furthermore, constraints are employed in design to ascertain the set of methods which might be used in the prosecution of a task or to determine the appropriateness of processes engaged by designers at different stages of the design.

Designers employ constraints to help them generate concepts and to refine these concepts into a completely defined, detailed solution. The primary role of constraints in design is to provide control over the process and facilitate the generation of valid solutions.

Constraints should not be considered as being exclusively restraining or, limiting. Their evaluation does not always lead to a logically 'true' or 'false' situation. Frequently they can identify the options available and guide subsequent actions to be taken constructively. Table 1 provides one taxonomy for constraints as they can be employed in design.

Type	Function	Applies To	Causes	Depends Upon
parameter	validation	value(s)	true or false	specification or design analysis or technological limit or availability or concept
default	suggestion or sourcing of information	value(s)	valid result(s)	current design state or technology or availability of particular resource or prescribed value or set
process	control	tools, methods or tasks	valid or invalid	design process stage or solution state
technological	evaluate and suggest technology to be used	tools, methodologies or concepts	suggestion of a set of technologies suitable to the task in hand; allows development of simultaneous or sequential methodologies	understood concepts, past solutions and beaten tracks
compatibility	evaluation of combinatorial possibilities and compilation of a set of options	design entities and methods	relationships, dependencies, associations (properties)	entity state if in the context of the design solution or entity type if in the technological domain

Table 1: A taxonomy for constraints in design

The establishment and representation of constraints constitute a definitive mechanism for encapsulating expertise, whether in the form of standards or past practice. Knowledge representation in design concentrates therefore upon the definition of constraints, their association with design entities and the mechanisms controlling their use, application and interaction with the designer.

Constraints can be either local or global. Local constraints are associated with the design task at hand. Global constraints are associated with generic[17] classes of problems in design. The designer's repertoire of global constraints increases with experience. This is manifested in the application of a particular technique (sometimes perceived as a *beaten track*) to a variety of different problems. Mechanisms supporting design environments ought, therefore, to be able to incorporate new, practice-oriented constraints and make them available appropriately to designers in all areas and at all levels of design.

Constraints may originate from **[Stauffer et al, 1990]** the initial statement of intent or be derived during analysis of the design specification. The former frequently describe the desirable performance criteria; they represent the design functions and sometimes depend upon the designer's initiative and expertise. They may be listed as standards or derived from the designer's past experience. The rate of occurrence of constraints derived during design increases as the design evolves: they constitute the majority of constraints affecting the designer **[McGinnis et al, 1990]**. In the Design Engine, facilities must be provided to enable the introduction of constraints by designers at any stage of design.

Constraints may be in conflict. A possible means of resolving these conflicts is by further classifying them as being fixed, relaxable[18] or according to their relative importance **[Stauffer et al, 1990]**. Fixed constraints typically represent performance criteria, boundaries on values or arise from considerations of compatibility. The integrity of the design solution is at risk when these constraints are not satisfied. Relaxable constraints are susceptible to control by the designers and are an aid in the resolution of conflicts. Control over constraints can influence the flow of design operations and aid the management of activities within the Design Engine.

Constraints may be applied in several ways. Bounding values may be implemented within the storage mechanisms of databases. In knowledge

[17]For example, the determination of the material type for a component which is to be assembled with another component made out of a given material relies upon the provision of a compatibility check which can be enabled via a global constraint.

[18]Or retractable, when infinitely relaxable.

engineering, constraints are often implemented as ranges of default values and condition-action rules. Adding constraints to the rule base of a knowledge based system (KBS)[19] has the advantage of integrating them into the overall conceptual schema **[MacKellar et al, 1991]**. However, the domain of application of the KBS may be narrow and restricted and, therefore, the utility of the constraints may not be maximised or exercised in associated domains. If constraints evolve in time, then rule base maintenance must be accommodated[20]. Several researchers have proposed that data integrity constraints be applied in databases to specify restrictions on possible designs **[Cammarata et al, 1986; Dittrich et al, 1986; Dayal et al, 1986]**. This is particularly suitable for decision aids.

In the Design Engine, management functions controlling the operation of the engine can be provided by:

- data model constraints,
- procedural constraints,
- domain constraints,
- specification constraints.

Designers will be able to use these management functions to:

- evaluate the relevance of given technologies to the design task in hand;
- determine the applicability or suitability of tools, methods and techniques available at a particular design phase;
- ascertain the conformity of solutions to particular design specifications currently in effect;
- impose compatibility restrictions which determine the possible relationships between entities;
- introduce limits on values.

[19]A **Knowledge Based System** is a computer system that embodies an explicit representation of knowledge on an application domain. The Knowledge Base is a separate component of the system, the other parts of which are domain independent. In essence, they are computer systems that can lend assistance with technical and professional tasks.

[20]To allow the KBS to function effectively, the presence of conflicting rules, within its Knowledge Base, must be avoided; the introduction of new rules or modifications to existing ones must be preceded by checks on how existing rules and, subsequently, the operation of the KBS might be affected. In design situations, where the number of interrelated parameters and rules is large, this rule integrity check becomes a significant task.

Table 2 provides the outline of a scheme to incorporate design constraints into the function of the Engine.

Nature	Incorporation	Verification
data model	ab initio, associated with data values, entities or output from specific methods and tools	check performed on the result
procedural	ab initio, associated with tasks, processes or specific tools	check performed prior to invocation and also following from it to cause possible concomitant actions
domain	ab initio, associated with data, entities, procedures, tasks and specific technologies; modified and/or enriched, retrospectively, based on practice and technological changes	checks as above but final acceptance by the designing authority; validation override used to modify current domain issues if appropriate
specification	introduced during the process of design; incorporated into the system following retrospective analysis by the designing authority, thereby causing transformation of 'beaten tracks' into domain, procedural or data model assertions	entirely under designer control

Table 2: Constraint incorporation in the Design Engine

Specification constraints are highly diverse and complex and formulation of a flexible representation may not be possible. Consequently, they should be left open to the designers to control, interpret and, if appropriate, introduce.

1.3.3.2 Information sourcing and communication

Data manipulation during design requires of the designers the ability to:

- deal with peculiar interfaces and languages,
- be aware of any data associations and, therefore, of the repercussions of their actions,
- anticipate information transformations as part of the task which they engage,
- have detailed knowledge about the processes and the information which is generated, including the differences in data generated by different versions of the software tools engaged.

One function of the Design Engine is to make data available to the designer. These will, in most instances, be part of the company's information base and be referred to by a number of past products. If stereotypes were to be used to structure the product descriptions which refer to the data, then use of the latter would allow for complete combinations of data and products, including references to their generating technologies, to be imported into the current problem domain. These could then be used by the designer to alter the input state and, consequently, influence the process of problem solving.

Information systems have been developed **[Eastman, 1992]** to manage access to the data by designers. Advanced implementations have employed knowledge-based tools to link distributed data sources: figure 7 depicts a representative example. A number of communicating KBS's are brought together and managed by a supervisory KBS which is responsible for external communication **[Eastman et al, 1991; Dilts et al, 1991]**. The necessity for user-driven data transformations is eliminated by incorporating the necessary methods **[Baral et al, 1991]** in the individual KBS's rule bases.

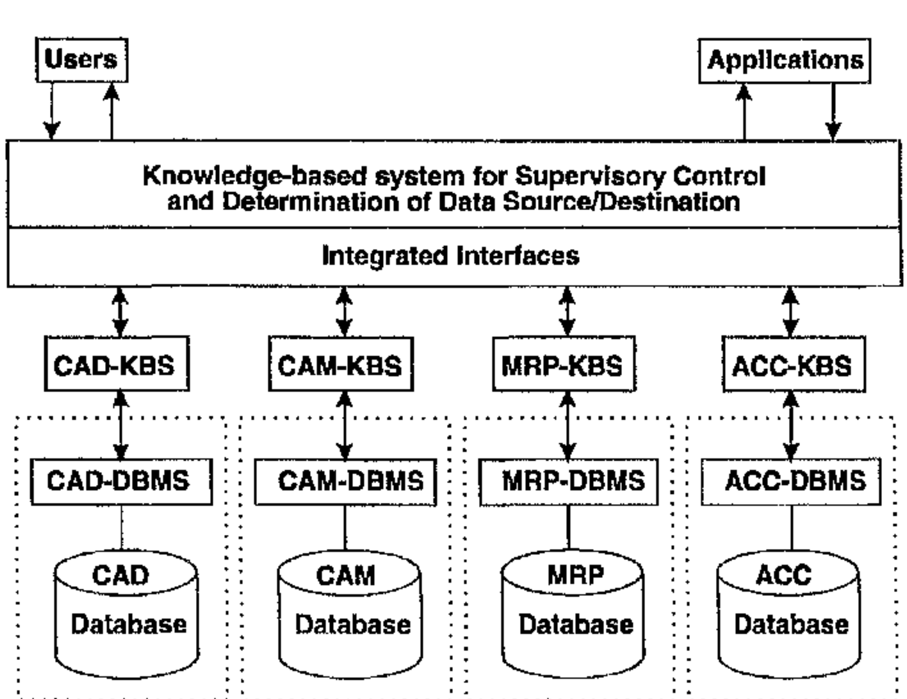


Figure 7 : A Knowledge Based System for data retrieval

One disadvantage of employing information systems for data retrieval and communication is that the flexibility in structuring and storing the data is lost through the exercise of strict control over the individual data bases to maintain the logical integrity of the whole system. An additional complication is that such schemes require detailed knowledge by the system administrator of the processes engaged by the designers and the data which are generated: this knowledge may not always be either available or easily expressible.

Stereotypes could be used for data manipulation without invalidating the pre-existing information systems, should they exist. Information system functions could be attached to stereotypes for the purpose of data acquisition, thereby causing them to be driven by design rather than as simple retrieval operations.

1.3.3.3 Method selection and invocation

In a unitary process (figure 5), input is transformed into output by a method exercised under external control. In the Design Engine the span of problems is such that several methods might be applied. This set of methods is constituted in the software tools and facilities employed by the enterprise and its agents. It includes routine software tools - e.g. a solid modeller or a manufacturing simulation package - or company-specific facilities, such as an internally developed structural analysis system.

These methods are, traditionally, invoked intuitively[21] under the control of designers. Designers match a need to generate data with their particular knowledge of the methods which are capable of providing them. Familiarity with the methods usually extends to knowledge of and adherence to the constraints applying to them. Furthermore, the designers are aware of relationships and mutual dependencies of methods and are capable of communicating their requirements to other experts.

Intuitive invocation of methods is acceptable in situations where problems are engaged exclusively by agents who possess peculiar expertise. Design concurrency[22] requires the experts in one field to generate information in

[21]The term refers to the familiarity of the designers with particular techniques, through past experience, training or, simply, availability; it is, in this instance, taken as synonymous to: 'performed as routine'.

[22]It is the author's opinion that design is by nature concurrent and this is evident in the analysis of a task and the synthesis of a set of possible solutions. However, rigid practices within engineering enterprises can prevent operations from taking place concurrently.

associated areas, where their knowledge is small. For example, a designer wishing to determine the optimum (or safe) position for a hole in a component may need to invoke a method to perform a structural analysis; furthermore, the designer must provide the requisite inputs and be able to interpret the results.

Another challenge is the identification of which method out of a set of possible methods should most appropriately be applied. This selection can be accomplished by employing a representation which closely associates input and output data with methods and includes multiple references to the data, if these exist, as options. Designers must be informed of the characteristics of the methods they use[23].

An advantage of employing stereotypes to describe methods is that the design record can be made directly to refer to data generated by the methods. Whenever a design activity is undertaken, a track is recorded which relates it to the activity and the methods employed. The methods used in performing a task, together with their input and output data as well as the constraints applying is the basic output of the Design Engine. This collection of information is the record of the design activity and forms the basis for the company's product descriptions. Product descriptions can be used to define models of the design practices of the company.

Product descriptions may be employed under the designing authority's control to synthesise further solutions[24] and provide a basis for concurrent operations. Incomplete descriptions - those activities which did not culminate in the development of a product - are also of utility. They may contain 'dead-end' solutions not appropriate to the current design, but which might be made appropriate by a relaxation of constraints or by changes in the technologies employed.

[23]Knowledge in a particular domain of interest consists of descriptions, relationships, procedures and methods. Descriptions consist of rules and procedures for applying and interpreting information in a specific context. Relationships express dependencies and associations between items. Procedures specify operations to be performed when attempting to solve a problem. Methods are engaged to carry out the operations specified in the procedures.

[24]These take the form of composite method and relationship descriptions and are usually referred to as: *technological models.*

1.4 Product models

A product model comprises a set of entities and a structure relating them. Use of product information by designers through the Design Engine suggests that the product model should embrace:

- information on the methods (software tools) employed in its generation, together with a record of the information which was referenced in the application of these methods;
- data as inputs to and outputs from the engaged methods;
- an account of all the constraints which were in force at the time of data generation together with a description of any assumptions made by the generating authority;
- references to technology models or expertise.

Several definitions of the nature of a product model exist among the engineering disciplines and are often domain dependent. Two representative definitions are given below:

"The term product model denotes the totality of data elements which completely define the product for all applications over its expected life cycle. Product data includes the geometry, topology, relationships, tolerances and features necessary to completely a component part or an assembly of parts for the purposes of design, analysis, manufacture, test and inspection." **[Smith, 1986]**

"A product model contains 'complete' and non-ambiguous information about a product. Complete in the sense that all information required for a specific application, if available, will be specified as an integral part of the product model or can be derived from it." **[Gielingh, 1990]**

These definitions are stringent on the requirements for completeness. They are most easily applied to established products or 'closed systems'. Such products are rigidly defined and the records are rarely amenable to change without loss of information. Furthermore, it is difficult, in most instances, to incorporate into the product model information which relates to products made by the company's subcontractors if this is differently structured. The product information structure cannot be modified easily and its primary perceived function is to service requests for data rather than be used to abstract definitions relating to practices which would be appropriate to its future use by multiple agents.

The emphasis in the product model definition for environments sustaining concurrent operations is on the incorporation, in addition to the routine data, of manifestations of a product's evolution. Ideally, this should be based upon the record of design of the product and comprise information which represents information and associated actions. This record should allow for an evaluation of the product and provide the opportunity for 'beaten tracks' to be deduced from it under the influence and explicit control of the designer or engineering authority for general use, leading to the multiple use of product descriptions in product design.

The use of stereotypes in product descriptions is appropriate and bears the advantage that the definitions derived can be organically related to the methods and data models employed by the organisation and its external subcontractors. Furthermore, a product model based upon stereotypes inherits the dynamic aspects of them and evolves over time while also being insensitive to changes in company practices.

A Method To Support Concurrency In Design

2.1 Architecture of the Design Engine

The Design Engine aids designers who must specify and analyse the design tasks. To sustain concurrent operations the engine must enable the communication of intermediate results to other designers as well as the need for other tasks. Past practice provides the basis for the analysis of tasks and the operation of the engine must respect what has been done before. This involves the abstraction from past records of the design process, which include instances of designed entities and their relationships, of generic structures and their presentation to the designer as available options in the analysis of the design.

The Design Engine operation and function is dependent upon a set of internal elements which enable it to interact with associated external facilities[25]. Figure 8 depicts a possible architecture for a Design Engine and indicates its internal elements and their connections. The engine comprises four elements: the Activity, Record, Execution and Acquisition Monitors.

The primary element is the Activity Monitor which manages and controls interactions of the engine and external facilities as well as those occurring between elements of the engine. Its operation is similar to that of a 'blackboard': it provides the designer with a communication surface; it has tools which assist the designer to segment the design task by formulating requests for information acquisition or the concurrent execution of associated sub tasks; it displays the responses to requests and enables the designer to manipulate them. It is the primary mechanism for building a model of the solution and managing any options which this includes.

Requests for data and information are managed by the Acquisition Monitor. It attempts to service a request by engaging a 'consultation' process which may involve: searching the available methods to determine possible matches; communicating requests to knowledge sources to elicit suggestions for possible solutions; invoking the product model navigator[26] facility to match the data request to values relating to existing products and subsequently, by abstraction, determining the methods or technologies which had been employed in the product

[25]Software tools, company methods, external standards and data base systems are representative examples.

[26]This is a sub-element of the Acquisition Monitor which is used to trace links between data items in the product model and, subsequently, enable the designer to navigate across related data items by following the links.

data generation: this allows multiple use to be made of models and of formal product descriptions. By incorporating a product description into the current design solution, the associated technologies are also imported. These can be used by the designer, through the facilities available in the Activity Monitor, to cause the concurrent exercise of several methods and to involve the appropriate disciplines in the synthesis of possible design solutions.

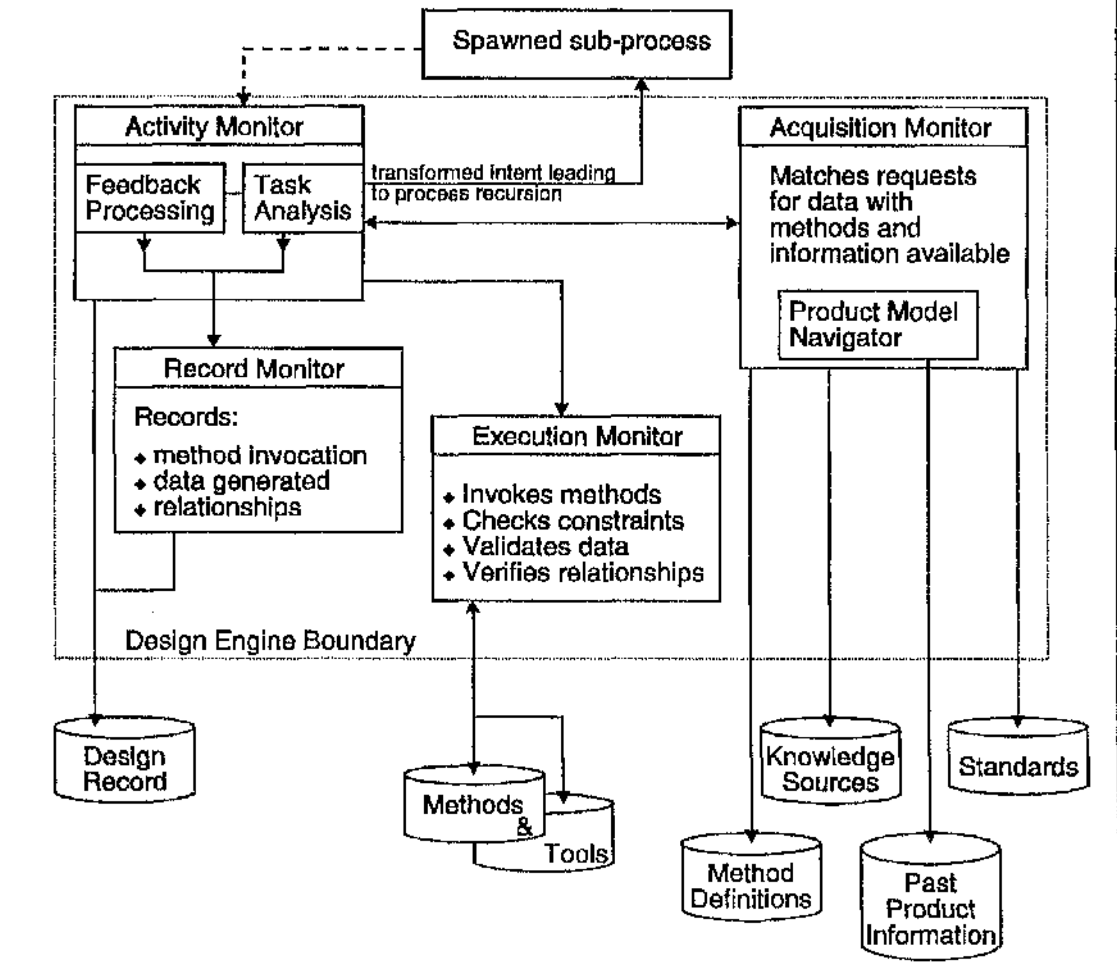


Figure 8 : Design Engine architecture

The results of data and technology requests are communicated back to the Activity Monitor as one or more options. Methods are invoked exclusively by the Execution Monitor which, in addition to executing the methods, is responsible for determining their appropriateness to the current design state. This it achieves by executing the constraints, if any, which relate to method control and by determining the input data to the methods and verifying their existence or their availability. Constraint failures or further requests for data are communicated to the designer via the activity monitor for resolution. Upon successful completion of the requisite method, the Execution Monitor validates the data (values) produced and any relationships generated before returning control to the Activity Monitor.

The Activity Monitor records actions taken by the designer and the state of the current design solution using the facilities provided in the Record Monitor. This element is responsible for recording the methods, data and data relationships associated with the current task. Information stored by the record monitor complements the record produced by the Activity Monitor. The two sets taken together constitute the complete product description and provide the principal output of the Design Engine.

2.2 Implementation approach

The operational characteristics of the Design Engine elements being well defined, their communication requirements are easily predicted and interfaces between them easily constructed. Constraints in the construction of the engine stem from a consideration of the nature of the external facilities with which the engine interacts during design: these are a collection of diverse methods and disparate data forms operating over a variety of distributed computational platforms.

The order by which external facilities will be accessed or invoked by the engine is very hard to predict as it depends upon the way in which the design task develops. It would be possible to predict the communication requirements if a single facility was to be engaged at any given time. However, the need to operate concurrently raises the requirement to plan for the exercise of several facilities simultaneously and introduces complex interactions and dependencies which cannot be predicted *ab initio*. In the proposed system this is not a presumption.

Communication interfaces which deal with specific data sets or methods could be combined to form composite linkage mechanisms. To deal with all possible interactions would require that a large number of specialist interfaces be built. Furthermore, since the data and the tools which generate them evolve constantly, high levels of maintenance would be required. An alternative is to generate descriptions of the data, the methods employed and their associations which are independent of the tools and applications used. Generic interfaces might then be built corresponding to the data descriptions and a control system employed to manage communications appropriately and construct the required interfaces using combinations of the generic ones.

These issues of representation and control constrain the implementation of the Design Engine architecture and the construction of its elements. The development of extensions to existing facilities by attaching knowledge-based systems to CAD software as shown in figure 9, have resulted in some possible implementations **[Held et al, 1991]**.

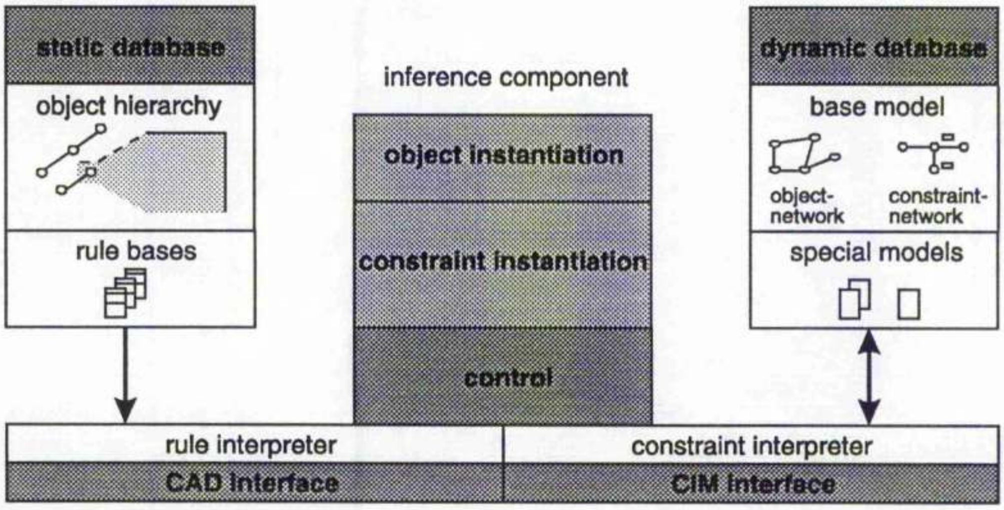


Figure 9 : A CAD - AI system combination

However, these combinations do not escape the problems of information description independence and interface maintenance; rather they compound them by introducing additional, proprietary interfaces related to the internal structures of the software involved. Generic descriptions employing stereotypes to represent knowledge associated with design entities can produce a solution. The Design Engine could then be built upon the facilities available within this generic framework and be unaffected by removal or replacement of the software facilities which it employs.

2.3 Representations

The incremental nature of the design process involves the integration of existing parts and the design of new components that will eventually form the required objects. New artefacts may be derived by modifying previous designs. Evolution, modification, adaptation and reuse of earlier designs is common practice and designs may share sub-parts. Design entities are the basic building block for the description of complex objects and designs **[Nguyen et al, 1991]**.

Design entities are structurally and semantically complex. Structural complexity arises from the need to represent hierarchies of entities (e.g. parts and assemblies) describing designs and products. Semantic complexity arises from the need to represent dependencies between entities. These dependencies arise from design rules or constraint verification procedures and frequently cannot be adequately described using routine semantic relationships[27] or predicate logic constructions.

Dependencies between objects arise in various forms during the process of design. The deletion of a parent object could, for instance, imply the deletion of all of its components. However, the deletion of the parent object could allow its dependants to be associated with several different objects. In some cases dependency arises from the acquisition of properties by association. Deletion of one side of a relationship (e.g. when a material type is associated with a component) may have no repercussions or result in design inconsistencies, especially if the related information is being used by other dependants[28] of the parent object.

Relationships during design are constrained by:

- the type of objects associated,
- the nature of the designed entities involved in the objects' description, and
- the requirements of the design task.

During the process of design it is likely that the designer will:

- introduce new constraints between design entities which define new relationships;
- relax the constraints between objects to achieve a required goal in a design task and change the existing relationships.

[27]For example: *belongs_to, is_a.*

[28]A typical example arises when an entity describing the properties of a given material is associated with a component: the material properties thus associated with the component can be used to constraint the assignment of a manufacturing process and, consequently, the necessary tools to form a feature associated with the component such as a hole. Deletion of the material to component association would invalidate the choice of manufacturing technique and cause a chain of inconsistencies subsequent to the process.

The accommodation of relationship definitions within object descriptions requires a very sophisticated control mechanism to manage, maintain and distinguish between all versions of objects. Furthermore, the differences in object descriptions would require a large number of object types to be defined and prevent the abstraction of generic object classifications. Dependency relationships ought, therefore, to be decoupled from object hierarchies. This suggests that a mechanism must be provided to represent dependencies as entities in their own right.

To satisfy the requirements for hierarchical representation it is necessary to:

- create entities to describe objects;
- generate definitions of relationships which allow for the imposition of constraints appropriate to the particular design contexts as well as for the type and number of entities which can be attached to either side of the relationship;
- provide a mechanism which allows aggregations of entities to be built, using the relationships defined as the linking elements.

Existing designs can be altered to produce new objects or more satisfactory results. New customised components might be designed by trial and error. This suggests that object descriptions are not static but may undergo several modifications during the design process. Design entities which describe objects must, therefore, be easily modified and combined with others to aid the evolutionary process. It is implicit that design entities may be left inconsistent or incomplete until all the information has been generated. The nature of the design process requires a more sophisticated consistency control on the design entities **[Borning, 1987]** to accommodate change.

The use of stereotypes to describe design entities and their relationships does not necessarily require that traditional definitions must be superseded or replaced to the detriment of existing product descriptions. Rather, the representation should incorporate existing definitions for the purposes of:

- building apposite combinations of them, the better to describe a design entity or activity; e.g. the imposition of constraints on the attributes of a composite definition which were unconstrained;

- communication of a set of entities and related information to various disciplines in forms which can be understood in the discipline;

- enhancing the function of tools, methods and software which operate upon data or relationships: e.g. to manage data storage or generate requisite geometric representations in a CAD system, by invoking these facilities organically within the design process.

Examples of stereotypical definitions are shown in figures 10, 11 and 12 taking a simple countersunk hole as the subject.

Figure 10 illustrates the combination of two entities used in solid modelling in a CAD system: the stereotype defining the hole employs[29] the attributes of the solid primitives sometimes in a transformed form; additional constraints can be imposed on the hole stereotype without the need to change the application software. The methods employed by the software to create and display the individual entities are retained, but they are invoked subserviently to the design task.

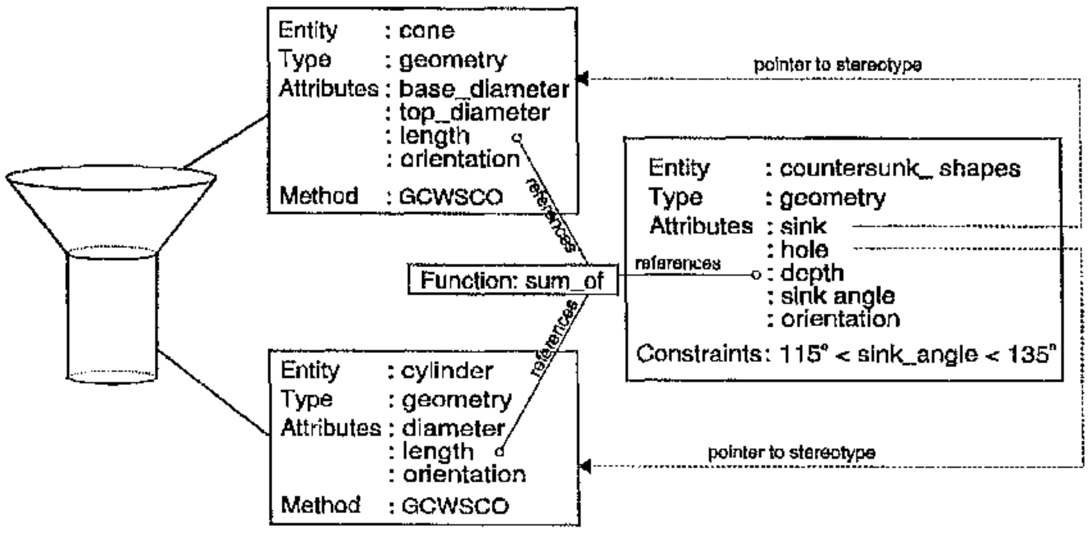


Figure 10 : Countersunk hole shape definition using stereotypes

[29]This is often referred to as: *attribute propagation.*

Figure 11 demonstrates the use of a set of information in the communication of a design concept to two engineering disciplines: the design of a countersunk hole places different requirements on manufacturing and structural engineers. The dimensions of it dictate the set of tools and forming processes required for its manufacture and the shape of the finite element mesh to analyse its structural integrity.

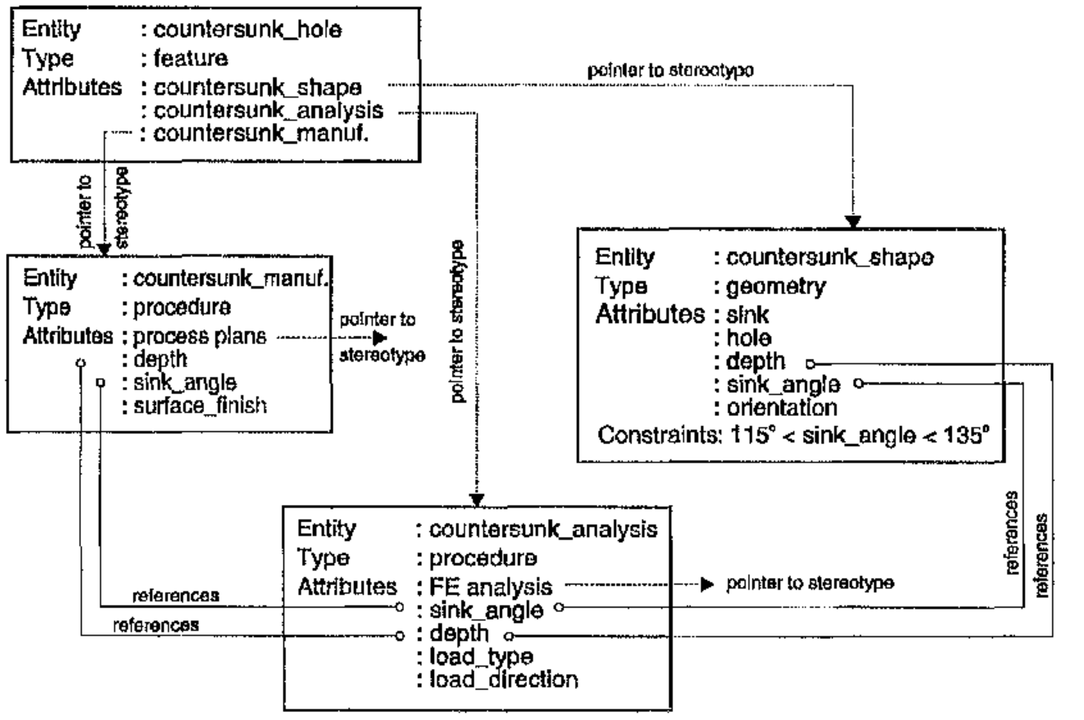


Figure 11 : Using stereotypes to define design entities and for discipline communication

Otherwise, the disciplines will view the hole information differently accounting for surface finish on the one hand and the magnitude of loads on the other. The description of the hole must satisfy all such needs coherently and consistently.

As an illustrative example, figure 11 depicts a structure for defining a hole: this collection of design entities shares common information. A designer generating a hole accounts for its entire definition including the appropriate technologies. Communications and interactions are determined by the description of the designed entity.

Designers often refer to data associated with products designed in the past. These data are, usually, accessible only via mechanisms designed to operate on particular models implemented by the company. These models serve the purposes of information storage and retrieval (section 1.2.3) and may be structured for CIM or be part of a proprietary data schema. Any representation employed by the Design Engine must respect the structure and maintain the integrity of past product descriptions but without constraining the designer by these structures. The design entity representation should provide access to the data and observe integrity restrictions, or invoke the proprietary mechanisms employed to maintain them. The advantage of using an entity definition is that several data manipulation operations can be combined into a single functional unit. In addition, the responsibility for any transformations of data, including data management, can be removed from the designer.

Stereotypical attributes can be defined to correspond to disparate data forms which may be physically distributed. This redefinition of the data allows them to be collated into alternative descriptions. Figure 12 depicts this concept using the countersunk hole example.

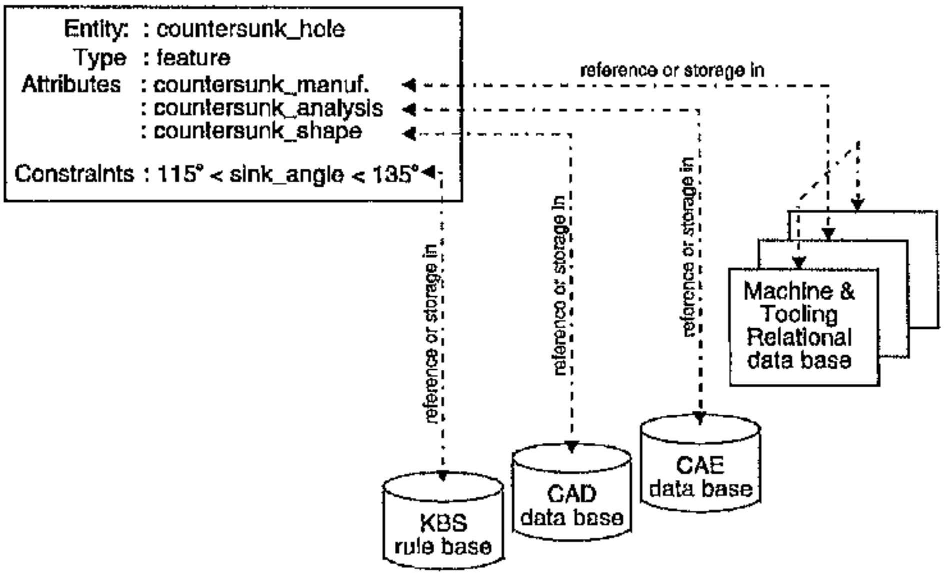


Figure 12 : Mapping stereotype attributes to disparate data forms

Knowledge of the location of the data and the appropriate methods for their retrieval is required to facilitate mapping of data values and stereotype attributes. Once the map is defined, a generic control mechanism can be provided to invoke storage/retrieval methods and transform the data. With this approach, the original data access methods and models can be maintained independently of definitions which the designer chooses to employ. The integrity of procedures and designs built upon the stereotypes is secured while the underlying data models continue to evolve. Additional control can be imposed upon the data by the association of constraints, external to the data model, with the stereotypical attributes which refer to them.

2.4 Employment of external representations

A salient feature of any engineering enterprise is the large number of calculations made, constraints imposed and techniques employed during design. The technological solutions are taken from the public domain or may be peculiar to the company. The availability of such solutions as external representations in the Design Engine safeguards the company's product development ethos and makes past and future designs contiguous.

A technological solution used widely in design employs *features*[30] to represent geometric characteristics of parts. Features represent stereotypical situations which can be applied to the generation of instructions for their manufacture, for assembly purposes or for stress analysis of a part. Figure 13 shows an example of the use of features to describe a gear shaft **[Ehrlenspiel et al, 1991]**. The shaft is represented as an aggregation of basic shapes (e.g. chamfers, fillets, keyways) linked by specific relationships: in the feature tree of the gear shaft, shown in figure 13, solid arrows indicate inheritance, dotted arrows indicate adjacencies and dashed arrows indicate explicit constraints.

Several implementations of features exist: in their vast majority, the feature definitions are incorporated within existing CAD systems **[Giacometti et al, 1990]** to provide facilities for the generation of geometry which can be used by multiple applications.

Dixon **[Dixon et al, 1983]** proposed a knowledge based design system which is not solely dependent upon a geometric modeller. The system consists of two parts: the first part consists of a user interface, a design with features library, an operations library and a monitor, which allows the user to create primary representations of features and in turn a primary representation of objects; the second part of the

[30]Several definitions for the term feature exist; the most representative one defines them thus: *features are sets of information related to part descriptions; they are entities which describe a characteristic of a local area of a part.*

system is used for converting the primary representations of the objects into the secondary representations needed by respective activities such as manufacturing and assembly. This conversion may involve some feature extraction to obtain more abstract features.

In a prototype system, proposed by Mantyla **[Mantyla et al, 1987]** the definition of basic features are based upon a classification of machine tool types. Hence, in principle, each basic feature can be manufactured with a single tool or with tools of the same type. In contrast, compound features must be processed with tools of several types. A pocket with fillets on the bottom is a compound feature, for instance, because its production will require several kinds of milling tools.

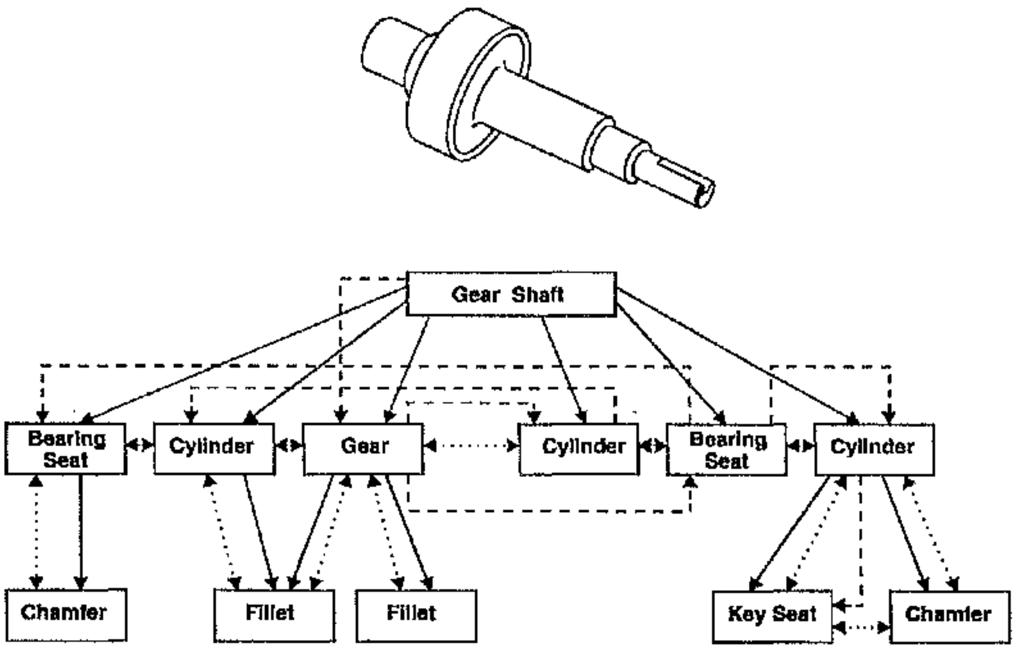


Figure 13 : Feature description of a gear shaft

Faux [Faux, 1986] summarises the design by features methodologies with an emphasis on the dimensions and tolerance models associated with feature models. He states that all features belong to some generic feature class in the sense that they obey the rules of the class, so that application software operations can always and only be applied to features whose classes lie in the domain of the operation concerned. At the highest level, features are defined as implicit 'specific features' of a particular generic class. A 'generic feature' is held in implicit form as a specific feature with default parameter values and a set of generic features from the feature library. A feature is a set of faces of a component, structured into one or more primitives and sub-features and whose faces, primitives and sub-features obey the rules of its declared generic feature class. A primitive is a set of one or more faces which are treated as a single indivisible unit for the purposes of defining tolerances on its shape and location. Four standard primitives are defined: plane, cylindrical, plane pair and swept profile primitives. A generic feature class is characterised by a number of structuring, selection, combination and sizing rules. A generic feature can be created by a generic modeller, the purpose of which is to allow companies to design their own generic features in a user-friendly way.

Features offer levels of abstraction from the low-level entities which the CAD, CAM or analysis software operate upon. They provide representation structures which are sufficiently open to access from other applications or agents without the need for elaborate interfaces, facilitate better data generation and allow for validation of their attributes using constraints. The emphasis in their definition is in the communication of information across engineering disciplines.

The design by features approach allows the designer to model a part in terms of generic shapes whilst maintaining freedom of design **[Butterfield et al, 1985]**. However, the available systems impose limitations on designers: the design by features library is finite and the feature operations, such as add, delete, edit, etc. are frequently limited. The flexibility and freedom of designing the geometry of an arbitrary part in conventional CAD systems have been lost to some degree in the design by feature systems **[Sackett et al, 1992]**.

A strength in the design by features method is its ability to incorporate the semantics of features. Parameters or properties of the parent feature can be inherited by the dependent feature. The definition of features can be employed to check the validity of other features.

Features provide an initial framework for the representation of design entities and objects which can be used to build descriptions of parts and products. However, in their present form, features are more appropriate to the generation of detailed part geometry rather than to the process of generic problem solving which is encountered in the initial stages of the design process. Furthermore, the peculiarities of feature implementations narrow the scope of application to specific areas of engineering design. The scope of application of features can be expanded by making them accessible to the Design Engine as external representations.

The availability of technological solutions, such as features, through the Design Engine provides the opportunity to associate them with methods and constraints making them available as generic or specific enabling technologies. The Design Engine must employ rigorous management methods based upon generic knowledge representation techniques to combine the internal end external representations used.

2.5 Control

The representation of entities, processes, methods and constraints is central to the implementation of the Design Engine. The pre-existence of design entities appropriate to a task must be communicated to the designer. The applicability of methods must be determined from their definition and the data they produce or by abstraction from past records of design or product descriptions. These methods and associated constraints and definitions must be invoked transparently whenever the designer needs them. Design agents or specialist teams must be called appropriately.

In essence, a controlling mechanism is required which enables combinations of the representations to operate and interact together. Control must be flexibly imposed and leave the designer as free as possible. This may require that methods be invoked recursively by the representations. Control should provide options to designers requesting data or acquiring technologies, facilitate the flow of information and secure distributed operation.

The evolution of methodologies for performing design tasks and the associated abstraction of technological models from specific solutions are dependent upon the combination of a record of the designer's decisions and interactions and the capacity for retrospective analysis. The Design Engine, must manage and analyse the design trail unobtrusively: this should make the designer's decisions traceable and provide a basis for auditing the process of design.

2.6 Design Engine support: Requirements

Concurrency in design requires linked, generic representations and control elements. In particular, the Design Engine must contain knowledge representations manifested as:

- descriptions of generic design entities and their attributes and the methods suitable to invest them with values;
- descriptions of technology including formal methods and software packages and procedures;
- rules and constraints applicable at different levels of abstraction;
- properties, relationships and inheritances;
- aggregations of entities and processes (product descriptions) and their relationships;
- diverse data forms, e.g.: graphical and geometric elements, alphanumeric and textual information and documents or multiply-indexed[31] information sets.

Assimilation of design methods used by enterprises, at different levels of abstraction, should be facilitated. The ability to define and support highly structured (sequential) methodologies as well as more open (simultaneous or concurrent) application is essential. The Design Engine must ensure that tasks are performed by individuals at appropriate times, control the sequence of operations within a project or task and, thereby, reduce the risk involved in product development. The Design Engine must also support distributed operations:

- by controlling its own communication needs and
- through its capacity to store and retrieve instances of the entities it operates upon using standard data bases and proprietary file systems.

Design traceability and audit of the design process require that the process route and the structured information which were generated during design should be recorded. During the conduct of a particular project the record would sustain status evaluations and provide for regression. From the record of experience, preferred methodologies could be abstracted to aid and service future product designs.

[31]Such as those used by hyper-text and hyper-graphics facilities.

More generally, the provision of a method which possesses the above characteristics will facilitate the building of a framework for managing Engineering.

The Framework For Concurrency

3.1 Concurrency: Issues

Concurrency in design depends upon the communication of the design requirements to and between specialists, the elicitation of responses from specialists, a harmony of actions with the emerging design solution and the acquisition of resources which may be functionally and geographically distributed. These dependencies are the focus for the development of the system for concurrent design presented herein.

Yeh **[Yeh, 1992]** has suggested that a concurrent design support system should provide for the scheduling of design activities and contain a model of the design process. A prescription of the design model not accounting for the product domain could inhibit designers by constraining their application of a design support system. Andreasen **[Andreasen, 1991]** suggests that concurrent engineering and '***design for***' approaches demand general procedures and specific methods. The methodology employed must not be limited, but should be expanded to involve management of design and recognise areas which might be adversely affected by design decisions. Facilities to model design technologies, company methods and procedures and to incorporate them **[Trousse, 1993]** into a design support system should avoid prescription if they are to contribute towards the provision of a generic design environment. This approach is adopted here and is an implicit quality of the proposed framework for concurrency.

Central to concurrency in design is the maintenance of multi-dimensional product descriptions. Design specialists should be free to create descriptions appropriate to their needs; and disparities in this set of descriptions must be managed within the framework. There are proposals that the product description must be standardised and validated for use and be communicable in a fixed form **[Bond, 1992]**. Erens **[Erens et al, 1993]** recognises the difficulty of maintaining different but consistent views of a product as a major shortcoming of current frameworks and identifies the designer's need to work with an arbitrary number of levels of abstraction. Product descriptions which employ representations of design entities, processes and their interrelationships are deemed appropriate to design concurrency **[Bauert et al, 1993]**. Thus, the description of products, the management of the design contributions and the development of design solutions are interdependent; they cannot be examined in isolation.

The issues considered here in the development of the framework for concurrency concern:

- the management of the design contributions made by engineering specialists and knowledgeable sources;
- the creation of the evolving design solution, including the recording and maintenance of the design decisions;
- the creation and communication of a formally agreed product description to the designers and associated engineering teams.

3.2 Managing the design contributions

Concurrent design can be regarded as being a distributed problem solving activity. A generic problem solving method discussed in **[Hayes-Roth, 1983]** employs a blackboard model. The blackboard sustains a number of independent knowledge sources and provides for communication between the sources through a common data structure. A knowledge source may be a human expert, an expert system or an application program. Blackboards have been used in a variety of fields where decision making is based on contributions from several specialist sources; its relevance for design applications is highlighted in **[Raczkowsky et al, 1990]** and **[Reddy et al, 1991]** who suggest that the blackboard model is particularly appropriate for the support of 'opportunistic design'.

Blackboards have been used by researchers for the development of frameworks for concurrent design. Myers **[Myers et al, 1992]**, describes a distributed system for architectural design. It employs several knowledge sources and a blackboard coordination expert to support design aspects spanning the entire life cycle of buildings. An important conclusion is that the provision of a common language to describe design issues is essential to the successful operation of the system. The language is defined primarily through prototype databases and is the basis for communication between human participants and components of the system.

A prototype system specifically for designing electrical power transformers is described in **[Finger et al, 93]**. It comprises a blackboard, a number of dependency reasoning algorithms and a network of conceptual dependencies which are representational of a transformer configuration. The system allows the designers to use prior designs, offers several perspectives to reflect the types of concerns involved in the design of a power transformer and enables 'how-to' and 'what-if' questions to be asked. The criticality of information integration for accumulation and distribution of the product knowledge from different sources is highlighted. The suggestion is made that the development of design support tools must be based upon the requirements of practice.

The need to apply rigorous control strategies in distributed problem solving to produce results to ill-defined problems **[Chandrasekaran, 1981]** has also influenced the development of blackboard systems for design. Constraint network and truth maintenance techniques have been used in conjunction with blackboards to manage communication between the knowledge sources and the constraints which are imposed by particular solutions within the design context. The combination of these facilities provides for design task sequencing by identifying the most suitable piece of knowledge to apply at any given time (usually depending on the state of the design and messages on the blackboard) and creation of a strategy for application of the knowledge (usually expressed in the form of an agenda).

DESTINY, a blackboard system described in **[Sriram, 1986]**, employs different knowledge based systems at different stages of the design process. It uses a high level strategy knowledge source to establish the sequence and execution of tasks (and the choice of which knowledge source to use). DESTINY relies upon an agenda and uses an inference mechanism to execute the knowledge source with the highest priority (the first on the agenda).

Vujosevic **[Vujosevic et al, 1991]** presents a framework for concurrent design which employs an assumption-based truth maintenance system (ATMS) in conjunction with a blackboard problem solver. Here, the ATMS is used to create and maintain relationships between different types of information used concurrently. It operates upon specific classes of information (product descriptions) for design which are manifested as objects representing complex entities.

A software architecture (HOBS) based on the blackboard model constructs a hierarchy of knowledge sources **[Carter et al, 1991]** to coordinate design. The blackboard mechanism - "Executive" in the architecture - formulates the strategy to control the knowledge sources and their communication by soliciting bids from the knowledge sources and selecting which one should next operate. Once a knowledge source has contributed, the control mechanism solicits bids from its children, thereby decomposing the problem. Once the children have appropriately operated (themselves causing decomposition), control reverts to higher levels in the structure and rebidding can commence to identify the next most suitable knowledge source.

The implementations examined above demonstrate the utility of the blackboard model for communication within frameworks for concurrent design. Complex control mechanisms are employed to manage the content of the communication and the coordination of design tasks. Task coordination, particularly, is shown to be successful in well-understood, specific design domains, but there exists still the need **[Bradley et al, 1993]** to support the decision making process in a way which is inherently neutral to the technologies employed. This view is supported by Kroll **[Kroll et al, 1991]** who suggest that the solution of a configurative design problem requires a high-level method to represent the design subject in terms similar to those used to formulate the knowledge. Eppinger **[Eppinger, 1991]** suggests that the concurrency of operations at the generic level of design creates tremendously large and unstructured problems that defy rigorous analysis. To sustain the concurrent approach, a framework which supports the analysis of alternative methods and enables the design team to decide which methods are appropriate is recommended.

Instrumental in coordinating the design activity within a framework for design concurrency are:

- a communication surface, e.g. drawn from the blackboard model;
- a representation scheme to describe entities and the design contributors as methods to invest the entities with values.

The classification of the contributing methods, in terms of their inputs and outputs, transforms a request, made during the manipulation of a design entity, into a search over the whole design space for contributors which can match the request. The responsibility for choosing a particular contributor lies, principally, with the manipulator of the design entity. Control over the validity (which might be in relation to the design context) and the sequencing of contributions can be provided by means of constraints over methods. The representation of constraints as methods enables the extension of their scope and applicability and would allow them to intervene - a desirable characteristic according to Visser **[Visser, 1993]** - in the direction the design develops. Furthermore, the availability of a product data searching method provides the opportunity to link the current design with past designs thus addressing the problem **[Beitz et al, 1991]** of integrating different design solutions.

One advantage of this approach is that the complexity inherent in coordinating the design activity is transferred into the description of the contributing methods and the specification of the constraints on them. The control mechanism is made simple and generic: it is required only to perform searches and execute methods. Its operation does not depend upon knowledge of the design context, which is otherwise a difficult issue to resolve **[Logan et al, 1991]**, generically, using constraint based or truth maintenance approaches. Another advantage of separating the control of the design's evolution from the framework supporting concurrency, is that a trace of unresolved problems, solution proposals and their evaluations can be recorded in a 'neutral' format. This would be useful **[Visser, 1993; Garcia et al, 1992]** to future problem solving organisation and can provide some of the necessary constraints for maintaining **[Nagy et al, 1992]** the design solution.

The delegation of control, within the Design Engine, to the design methods and constraints provides the opportunity to develop highly structured (sequential) as well as more open (concurrent) methodologies for design. As a consequence, the descriptions of the entities, methods and, particularly, the constraints operating upon the methods require careful construction.

3.3 Creation and maintenance of the design solution

The generation of a design solution is based upon the synthesis of an initial model of the artefact and the refinement of this model to achieve the satisfaction of the initial specifications as well as intermediate constraints arising from the specific direction the design has taken. To support the design solution the basic requirements are:

- a set of entities and methods from which to build aggregations suitable to represent the design;
- a mechanism to manage the aggregation of entities and methods by validating relationships created by the designers;

- methods to maintain the integrity of the emerging solution when existing parameters are changed, design constraints are modified (relaxed or enforced) or new entities, parameters and constraints are introduced.

Essential to the creation of the design solution is the choice made of an initial product description. The 'domain theory' presented by Andreasen **[Andreasen, 1992]** emphasises the designers' need to work with many objects of different types, each representing different views of the product and utilising models of past solutions. These can be employed in a 'checklist' like manner and, typically, may be illustrated with sketches to drive the generation of an outline solution. Bauert **[Bauert, 93]** suggests that the indication of 'why' to use, or not to use, a particular solution in the sketches of the designers' notebooks, provides an indication of the design rationale and can be useful in formulating future design solutions. Brown **[Brown, 1989]** and Ullman **[Ullman, 1991]** consider design history as a good basis for building an outline initial solution: old designs developed within the company are an important source of knowledge for designers.

Some approaches to creating an initial design solution are based upon prescriptive product models. The formulation of these models relies upon analyses of components which may be used in design and to which taxonometric considerations have been applied. Several examples exist: Brandenburg **[Brandenburg, 1992]** proposes object-oriented mechanisms to define part structure and behaviour models; Zhu **[Zhu et al, 1993]** proposes the employment of a knowledge-integrated, object-oriented, feature-based product definition model to support design concurrency. Thornton **[Thornton et al, 1993]** suggests that the product model can be divided into components, interfaces and constraints; the structure of each category is predefined and remains fixed. The same work recognises that in order to create a commercially viable system, generic interfaces, generic features and the functionality to allow generic components to be included as extensions must also be provided.

Prescriptive product models are limited in that the span of available solutions is constrained by the extent of the paradigms considered in their development. Models having generic utility must be based upon a consideration of all of the elements which might be encountered in every aspect of design and product development, or else a mechanism must be provided which employs a small set of generic elements to create product descriptions of relevance to the design area being tackled. A generic product description generation mechanism could be applied to the evolution of product descriptions from records of past designs by providing for the integration of the design history with existing representations. In the development of generic design methods presented here, the product description is based upon generic descriptions of entities and methods and facilities to aggregate these elements and, appropriately, to incorporate past designs from a record compiled of designers' actions.

The creation and maintenance of the design solution has been regarded as being equivalent to a constraint satisfaction problem **[Simons, 1970]**. In several examples an initial model of the design is constructed which contains a specification of all the parameters, the constraints applying to them and the relationships between them. Design proceeds through an attempt to satisfy all the constraints set out in the initial design specification and any additional constraints discovered during the search for solutions. The classes of design constraints which are examined are: rule oriented (of type *if.. then..*), relations oriented (applying to design parameters and expressed as equations employing equality or inequality operators) or discrete (for example to restrict a design parameter to a set of values).

The management of constraint satisfaction uses AI tools, of which examples are truth maintenance systems, non-linear optimisation algorithms or graph-based approaches. Banares-Alcantara **[Banares-Alcantara, 1991]** presents a system which employs an ATMS to identify and examine alternatives and maintain the dependencies of equipment and alternative plants during explorations of the design space. Akagi **[Akagi, 1991]** presents a general design system for ship preliminary design and power plant selection which is built as an expert system shell. The system determines the design parameters by following an 'intelligent' searching procedure of the productions. Pham **[Pham et al, 1991]** has examined the utility of ATMS's and suggests that they can support the generation of a design solution and its variants and the resolution of design constraints by temporarily changing the design assumptions. ATMS's perform well if the design requirement is structured as an existing design which is to be modified or developed to meet an extension of the specification: the efficacy of the method depends heavily upon the completeness and detail of the initial design and how closely it can be made to conform with the modified specification.

Young **[Young et al, 1991]** proposes a system (SPARK) which employs constraint networks to detect incompatibilities in the design parameters and advises the designer on improvements which can be made to the design from the perspective of the product's life cycle. The SPARK system is shown to be flexible enough to allow the designer to approach a problem (the design of a printed wiring board) from a variety of viewpoints and with incomplete information. A system described in **[McMahon et al, 1993]** employs Petri nets, with dependencies modelled as constraints along the links between nodes, to propagate changes in the representation of one part of a design property set to dependent models.

Facilities imbedded in CAD systems (e.g. ICAD, Concept Modeller, STONE Rule) attempt to satisfy design constraints by firing sequences of productions which propagate constraints through a network of components. Creation of the sequence of rules and the ordering of them so that they can be taken in sequence is time consuming and reduces the range of design problems which may be tackled. Other systems, e.g. CADET **[Thornton et al, 1993]**, employ prescriptive product models to assemble a model of the constraints applying to the design parameters of a particular solution and employ genetic algorithm techniques **[Dasgupta et al, 1991]** to reduce and optimise the number of effective constraints and enable their satisfaction.

Logan **[Logan et al, 1991]** regards ATMS based constraint maintenance as being restricted to the maintenance of dependencies between data items. ATMS's do not contain knowledge of the context of the design parameters or of the assumptions made in their derivation. A design problem has no inherent structure; rather, it acquires structure through analysis and solution building. The consistency of the solution expressed in terms of constraints does not determine context: a design proposal will be inconsistent with its constituents for much of its history as the designer attempts, with varying success, to reconcile conflicting requirements. Furthermore, Medland **[Medland, 1993]** suggests that for ill-structured, incomplete and evolving problems, the constraint rules and their dependent relationships are difficult to determine *ab initio* and may only emerge during problem investigation. The conclusion offered is that learning processes are required in order to establish the relationships of parameters.

Design systems which maintain constraints on parameters and seek to satisfy them by constraint propagation or through optimisation, depend upon the pre-existence of constraint models of the design solution which encompass all facets of the solution. Such systems can be applied to routine concurrent design **[Moynihan, 1993]**, but they have little utility in generic design. It would be more appropriate to try to constrain the design solution by evaluating constraints at two levels: firstly, at the design parameter level, via value-oriented constraints on the methods determining parameters and on method execution; secondly, by controlling the aggregations of design entities through constraints imposed on the relationships formed between them. Controlling the validity of the design solution when design parameters or constraints are modified, while respecting the design context, can be achieved only by retracing the designer's actions and determining the dependencies of constraints through the record of the design history. This method will be adopted in the framework for concurrency presented here.

3.4 The framework for concurrency: concept to delivery

The following chapters describe a software utility - GUIDE - which provides the enabling technologies required for the development of the Design Engine described in section 2.1. GUIDE's development is based upon the concepts identified above as being fundamental to a framework for concurrency in design. The aim is to demonstrate the validity of the concepts outlined by delivering the software utility GUIDE and testing it through application to industrial design and the business needs of engineering companies.

The Glasgow Utility For The Integration Of Design (GUIDE)

4.1 Concepts

A primary function of the Design Engine is to aid the analysis by designers of a product requirement to a set of manageable tasks. GUIDE supports the Design Engine to analyse the design representation requirements to lower levels of resolution and to match[32] them to stereotypical situations for which solutions exist. GUIDE achieves this through:

- the adoption of a uniform representation structure and
- by providing a control mechanism to match solutions to objectives.

A further requirement, stemming from the nature of the design process, is to conduct this analysis in a unitary manner, provide for regression and facilitate the co-operation of different levels within an enterprise.

Design rarely starts with well-defined goals **[Dixon et al, 1983]**. In most cases, a unique solution may not exist. Different designers may satisfy the requirement in different ways. The designer's decisions ought to be respected by any computational tool supplied to aid in the decision making process: the method to support the design operations must propose resolutions as available options, where these exist, and allow the designer to decide the best way to tackle a problem. Furthermore, it should enable the designer to modify constraints and record these interactions for future reference.

GUIDE must provide a non-prescriptive presentation of options and have the capacity to manage associations between entities and constraints affecting them. GUIDE must also incorporate a recording function in its control mechanism so that the design record can be a part of the product description. Through this function GUIDE can acquire and contribute design knowledge.

4.2 Knowledge and entity representation in the Design Engine

Design solutions often depend upon the aggregation of well-defined, formal entities into unique formations. The solutions are not informal; they are unpredicted, may be unusual and are, at intermediate stages, incomplete. For design, knowledge representation using productions has several drawbacks: the knowledge has to be very detailed, specific and modular because every possible

[32]Or the designer can design stereotypes which match the requirement.

combination of situations has to be considered; it is difficult to organise heuristic rules in practice, even though the basic theory of a particular domain may be well structured; extraction of knowledge from an existing design problem for use in another is not straightforward **[Pham et al, 1991]** and conflict is managed only with difficulty and at a cost in production systems.

Design requires *deep knowledge* representations. These provide:

- explanations of the context of the object domain so that the properties of stereotypes and their relationships are explicitly defined;
- separation of the structure of stereotypes from their function;
- detailed representation of all *cause and effect* relationships;
- symbolic rather than numerical representations.

Frame based and semantic network representations generally satisfy these criteria but, in the design application, have shortcomings in their ability to:

- secure the integrity of the stereotypes they describe in the presence of change;
- safeguard against redundancy of the data which depend upon evolving definitions;
- preserve the operational integrity of constraints on data, properties or definitions which depend upon (hierarchically) lower level, entities which evolve;
- operate in a distributed and dynamic environment;
- interact with conventional information systems such as relational databases.

The current representation mechanisms are the fruits of mature thought and considerable expertise has been invested in them. A solution which combines elements of the existing schemata and employs additional facilities to overcome the shortcomings identified is appropriate. Complex, focused representations frequently require elaborate control mechanisms to manage them and lack the capacity for future extension. Simple, generic representations on the other hand, are more easily controlled but require discipline in their application. Design entities and activities being of a disciplined nature, if kept modular, would be better serviced by a unitary approach.

The Glasgow Utility for the Integration of Design employs a knowledge representation scheme based upon Minsky's **[Minsky, 1975]** frame structures. The basic frame concept is implemented using a conventional relational database system. Relationship management employs network techniques and communication capacities, such as message passing, are included in the representations. Operations on the knowledge[33] are managed by control functions which can also cause data transformations. The capacity to aggregate generic, modularly defined knowledge representations to create higher level, formally constructed unique solutions or part solutions is a salient feature of GUIDE.

4.2.1 Elements

The knowledge representation scheme adopted for GUIDE involves *structures, atoms* and *methods* as its elements. These elements can be loosely compared with the traditional frame elements, namely *frames, slots* and *deamons* respectively. Additional characteristics have been invested into the elements to provide for their manipulation and control. This is the main point of departure from the traditional frame based approach.

4.2.1.1 Structures and atoms

The basic element available within GUIDE for entity representation is the structure, which is similar to a frame. Structures are the basic building blocks for stereotypes relevant to design or other activities. Structures are organised into families. This allows for different classifications of the structures to be built - e.g. to group geometric stereotypes into one family - and enables GUIDE to *filter* their use by designers - e.g. to restrict the span of the available stereotypes at a particular stage of design. In certain instances, GUIDE employs this classification facility to adjust its behaviour: structures belonging to either the ACTIVITY or RELATIONSHIP families are treated differently to all other types; this special treatment is explained in section 4.2.1.5.

Structures contain atoms in the way that frames have slots. Atoms can be associated with a value or set of values relating to physical quantities. Atoms may serve as *pointers* to other stereotypes which can be GUIDE structures or other external aggregations, such as those employed by geometric modellers to describe graphical elements. The range of possible values, GUIDE structures or external aggregations which can be associated with atoms is shown in table 3.

[33]Structured design entities and information generation processes.

Atom Type	Related Entity
value	integer number
value	double precision real number
value	character string
value	array of integer numbers
value	array of double precision real numbers
value	array of character strings
value	array of 3 real numbers corresponding to the normalised components of a direction vector
value	array of 9 real numbers corresponding to the definition of a plane
pointer	geometric element
pointer	general GUIDE structure
pointer	RELATIONSHIP GUIDE structure
pointer	ACTIVITY GUIDE structure
value	matrix of integer numbers
value	matrix of double precision real numbers
value	matrix of character strings

Table 3: Atom classification scheme

Figure 14 provides an illustrative example of a structure and its associated atoms. In contrast with traditional object oriented representations, implying attributes, atoms defined using GUIDE are attached to structures externally through the specification of explicit relationships[34]. This technique allows atoms to be associated with several structures simultaneously and leads to economy in the formulation of representations.

With this technique:

- atoms can be associated with structures more than once;
- atoms can be added to or abstracted from a structure so as to evolve a new structure representation;
- the association of atoms with several structures define relationships between them.

[34]Details of the relationships used in GUIDE to define associations between the elements it employs are given in section 4.2.1.

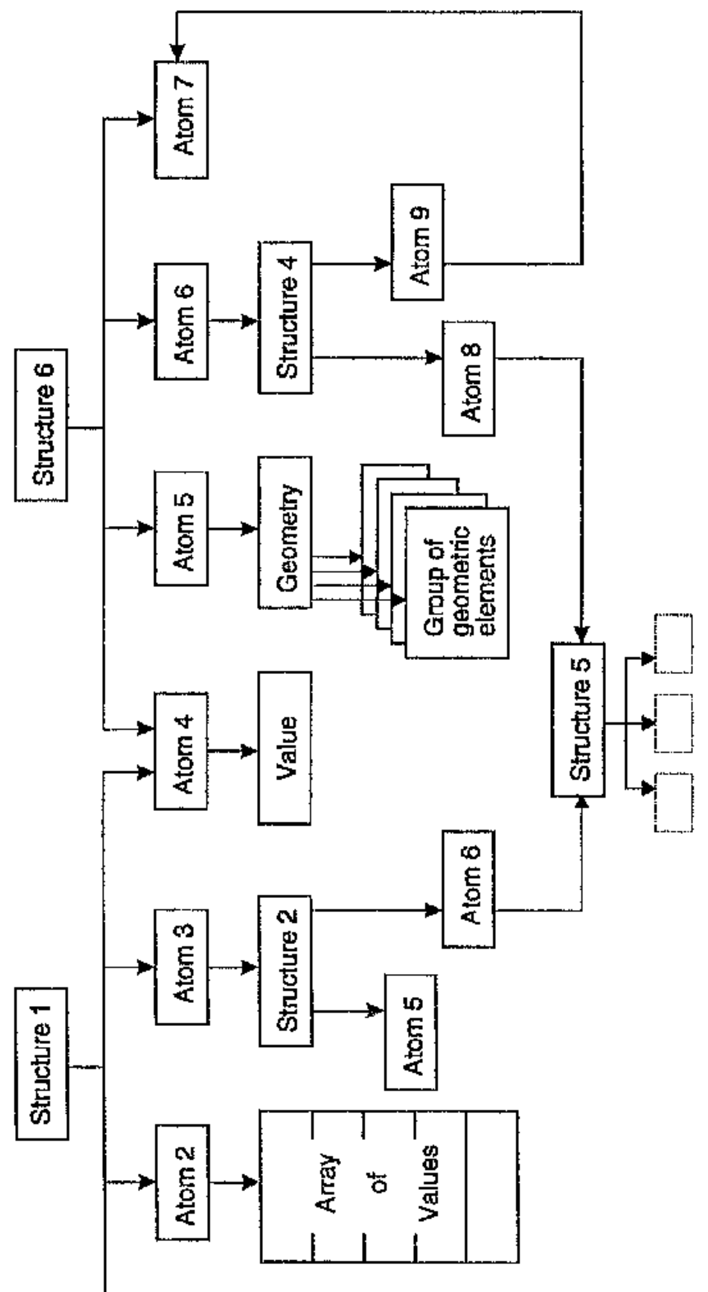


Figure 14 : Layout of a GUIDE structure

4.2.1.2 Characteristics of atoms

The definition of an atom includes information about default values or methods to supply them with initial values and the methods and constraints which may influence their investment with particular values, i.e. their *instantiation*. Figure 15 summarises an atom's characteristics.

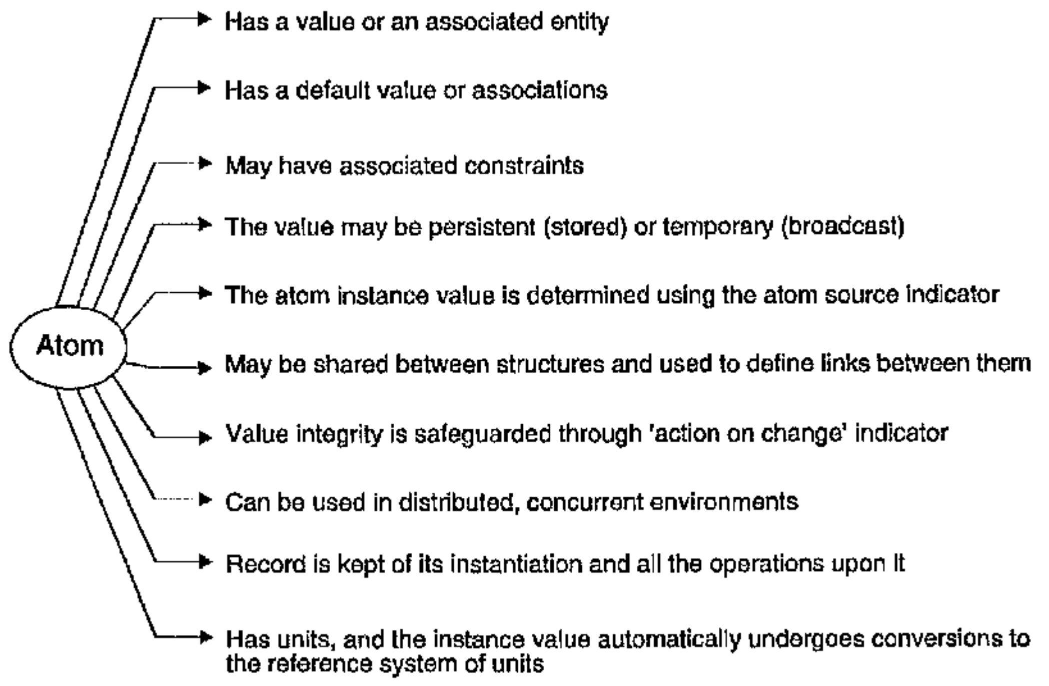

Atom
Has a value or an associated entity
Has a default value or associations
May have associated constraints
The value may be persistent (stored) or temporary (broadcast)
The atom instance value is determined using the atom source indicator
May be shared between structures and used to define links between them
Value integrity is safeguarded through 'action on change' indicator
Can be used in distributed, concurrent environments
Record is kept of its instantiation and all the operations upon it
Has units, and the instance value automatically undergoes conversions to the reference system of units

Figure 15 : Characteristics of atoms

Initial settings can be provided to atoms through:

- a default value,
- a set or list of default values,
- a calculation (part of a method) which yields values, or
- the value instantiating an atom within GUIDE.

The provision of initial settings is not exclusive to atoms taking values; initial associations can be defined for pointer atoms. The search for initial values is performed consistently, irrespectively of the atom type.

Constraints can be specified on the values instantiating atoms, but the assignment of constraints to atoms is not mandatory. The association of constraints with atoms is specified using a GUIDE relationship (section 5.2.1) externally to the atom description. Constraints are GUIDE methods which are invoked every time a change of the stated value of the atom takes place. They can be used to validate the instantiation or cause concomitant actions defined within the constraint. Figure 16 provides two illustrative examples.

The structure in figure 16(a) provides the definition of a line segment. It has three atoms: two relate to the line start and end points and are expressed as structures and one which is the line length. The constraints on this line structure might be as follows:

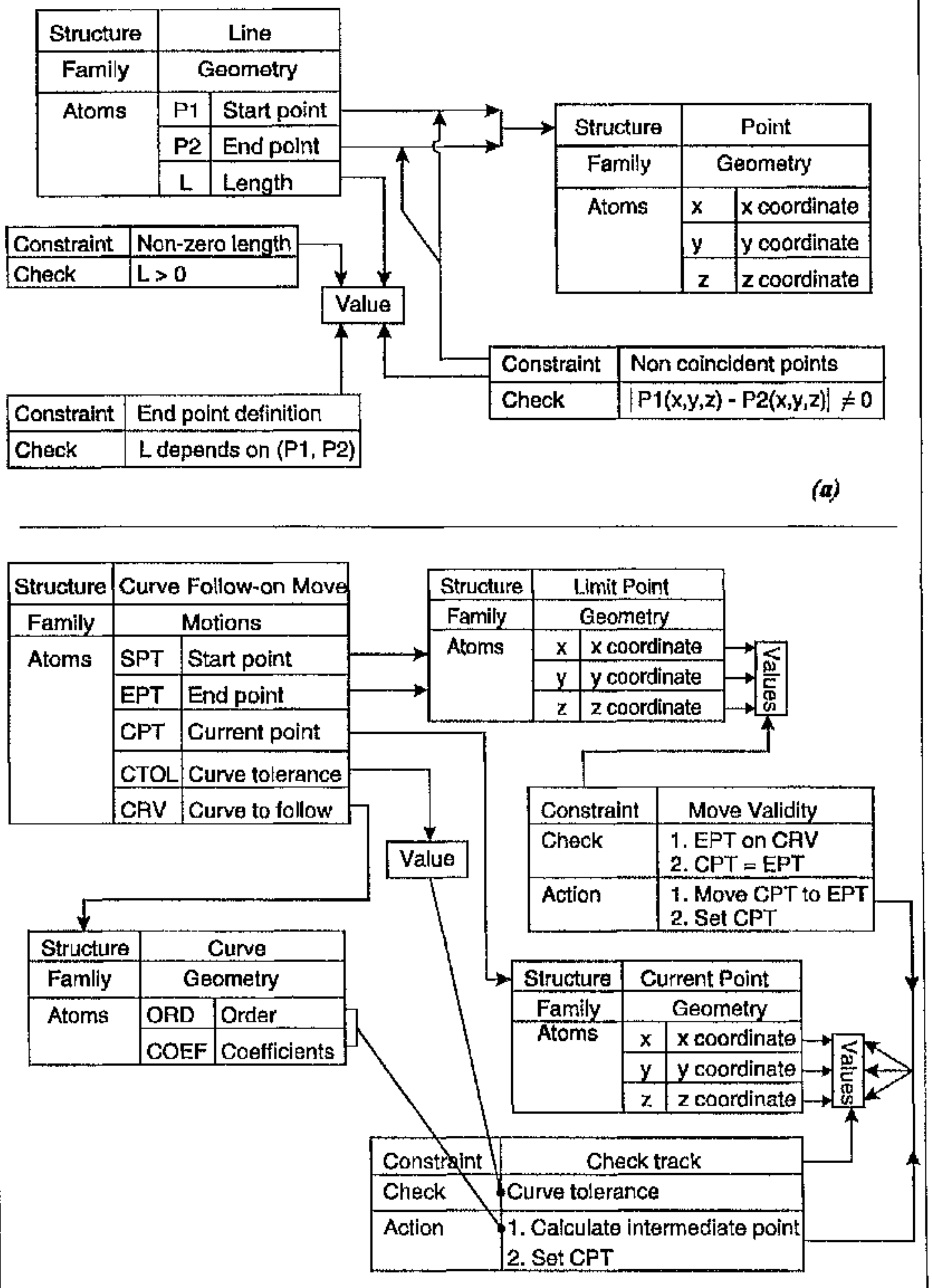
- a constraint on the points to ensure that they are not coincident and, subsequently, to calculate the line length and set the appropriate atom value;

- a constraint on the length value to check that the proposed value has been calculated using the co-ordinates of the end points rather than been input by the user[35].

The structure in figure 16(b) provides the definition of a move on a given curve. It has atoms to define the move limit points[36], the curve to be followed, the location of a point moving along the curve (current point). It is assumed that a number of linear moves will be used to approximate the motion along the curve: a tolerance value is included in the structure definition. The structure is initialised in the state where the limit points and the intermediate location are all coincident. A move is effected by a change in the co-ordinates of either of the limit points.

[35]Alternatively, the position of P2 could have been calculated using P1, L and a direction vector. In this case the constraints should be used to check that L>0 and that P2 has been calculated using P1, L and a vector.

[36]Labelling of the limit points as 'start' or 'end' is intentionally avoided. Motion can take place in either direction between the two limit points.


Structure | Line
Family | Geometry
Atoms | P1 Start point | P2 End point | L Length
Constraint | Non-zero length
Check | L > 0
Value
Constraint | End point definition
Check | L depends on (P1, P2)
Structure | Point
Family | Geometry
Atoms | x x coordinate | y y coordinate | z z coordinate
Constraint | Non coincident points
Check | |P1(x,y,z) - P2(x,y,z)| ≠ 0
(a)
Structure | Curve Follow-on Move
Family | Motions
Atoms | SPT Start point | EPT End point | CPT Current point | CTOL Curve tolerance | CRV Curve to follow
Structure | Limit Point
Family | Geometry
Atoms | x x coordinate | y y coordinate | z z coordinate
Values
Constraint | Move Validity
Check | 1. EPT on CRV 2. CPT = EPT
Action | 1. Move CPT to EPT 2. Set CPT
Value
Structure | Curve
Family | Geometry
Atoms | ORD Order | COEF Coefficients
Structure | Current Point
Family | Geometry
Atoms | x x coordinate | y y coordinate | z z coordinate
Values
Constraint | Check track
Check | Curve tolerance
Action | 1. Calculate intermediate point 2. Set CPT
(b)

Note: The "Limit Point" and "Current Point" are structures derived from "Point".
See section 3.2.1.2 for an explanation.

Figure 16 : Use of atom constraints to cause concomitant actions

Constraints associated with the limit points and the intermediate location provide the mechanism for the move. The constraints are defined as follows:

- a constraint on the limit points to check that the new co-ordinate location (the destination) lies on the curve and, subsequently, to set the co-ordinates of the point moving along the curve to the destination co-ordinates[37];
- a constraint on the point moving along the curve to calculate the deviation of the straight move from the curve and, if found to be greater than the specified tolerance, calculate a valid position for the point on the curve by iteration[38].

The instantiation of an atom is derived by the specification of the *atom source*. The control mechanism of GUIDE seeks a value for a particular atom from the user or a specified facility. Table 4 provides a list of the possible atom source options which can be specified in the atom definition.

Atom Source	GUIDE Response
User	Value must be directly input by the user of GUIDE.
System	Ignore any user interaction and obtain the atom value using the mapping indicator (described below) or any atom instances of matching type.
Method	Calculate the value using one of the appropriate methods only.
Any	Match the atom value definition to any other atom value in the system or to the output of any method of a similar specification.
Frequent	Use the most popular past choice for determining the atom value.
Rare	Use the least frequent past choice for determining the atom value.

Table 4: Atom source options

Atoms are characterised as being *stored* or *broadcast*. The value of broadcast atoms is only maintained for a limited time and is eventually discarded. Stored atoms become infinitely persistent once they have been committed and cannot be deleted. Prior to their commitment, stored atoms behave as if broadcast and can be modified; after commitment, they can be updated by the generation of a new version, but the old one is kept. Associated with every atom are its *physical mapping* and a *mapping method*. These define the physical location where atom

[37]The constraint is designed to execute recursively until the point moving on the curve and the limit point are coincident.

[38]The constraint calculates the mid-point between the current position and the destination, projects it onto the curve and sets the destination point to this. The constraint check recurses until a suitable co-ordinate location is established and this causes iteration.

instances are stored and from which they can be retrieved and the method to perform these operations[39].

Atoms are allowed to share physical locations. In this way different atoms associated with different structures may be implicitly linked through their common value. The sharing of physical locations can be used to conflate data with linked engineering significance. For example, the atom describing the diameter of a hole in a component can share the same physical mapping location with the atom describing the diameter of the tool required to form it. Instantiation of either the tool choice or hole size will affect the hole and tool structures. This sharing of physical mapping is an implicit constraint on the atom definitions.

Broadcast or stored but uncommitted atoms which have been invested with values can be modified either by their creator or another user of GUIDE. The modification of atom values depends upon the *action on change* indicator in the atom definition. The options currently supported are described in table 5.

Action	**Response**
Ignore	The atom value can be changed provided that the new value has been validated by execution of all constraints upon it.
Warn	The atom value can be changed provided it has been validated and notification has been sent to all the atoms and structures which reference it; this implies that the value change will depend upon the successful update of other atoms and structures which depend upon it.
Refer	The atom value cannot be changed automatically; following its successful validation, the original value creator is notified and asked to accept or reject the update. If the modification requester is the original creator the change is effected immediately.
Reject	The atom value cannot be changed under any circumstances.

Table 5: Actions relating to atom value change

The definition of an atom is made complete when the units of its value are specified. Systems of units can be defined independently of the atom definitions. Definitions of units include information on:

[39]A detailed description of the atom instance mapping operation is given in section 5.2.2.

- the type of unit
- the conversion algorithm to another type of unit (e.g. from millimetres to inches).

This *unit conversion* mechanism is used by GUIDE to maintain consistency of values across various systems of units.

4.2.1.3 Characteristics of structures

A structure's behaviour on instantiation depends upon the dynamics of the atoms it references. Pointer atoms associate one structure with a variety of others, the associations being controlled by constraints imposed upon the pointers. Hierarchical knowledge representations can be formed by association of a *parent* structure with a number of others, the *children*. Whilst structures can be freely associated, the links between them are tightly controlled. Figure 17 provides an illustrative example of a structure's definition characteristics.

The structure illustrated represents a component within a physical assembly. It references atoms which are attributes of the component such as its part number, the engineering change level and associated documentation. The latter is a pointer atom used to link various documents such as conformance standards or paper drawings to the component. No restriction is placed on the number of documents which can be attached to an atom. A constraint can be associated with a pointer atom to ensure that links formed are with the correct type of structure e.g. with a **drawing** or a **specification document**. The component structure has two more structures associated with it in the illustration. One attaches the material to the component; the other identifies the responsible engineer. These structures are *children* of the component structure; furthermore, only structures of the required type can be linked with the component structure at any instance.

This convention provides for the use of children structures mainly as representation of knowledge about the parent structure which is always true and predictable: a part will always be made out of a given material and by a responsible engineer. In contrast, the part may have some documentation such as drawings or standards associated with it through pointer atoms. The convention is flexible and extendible. It is flexible because it allows the designer to assess the suitability of associations formed; it is extendible because the number and type of associations that might be formed are determined by means of constraints on the pointer atoms which form them.

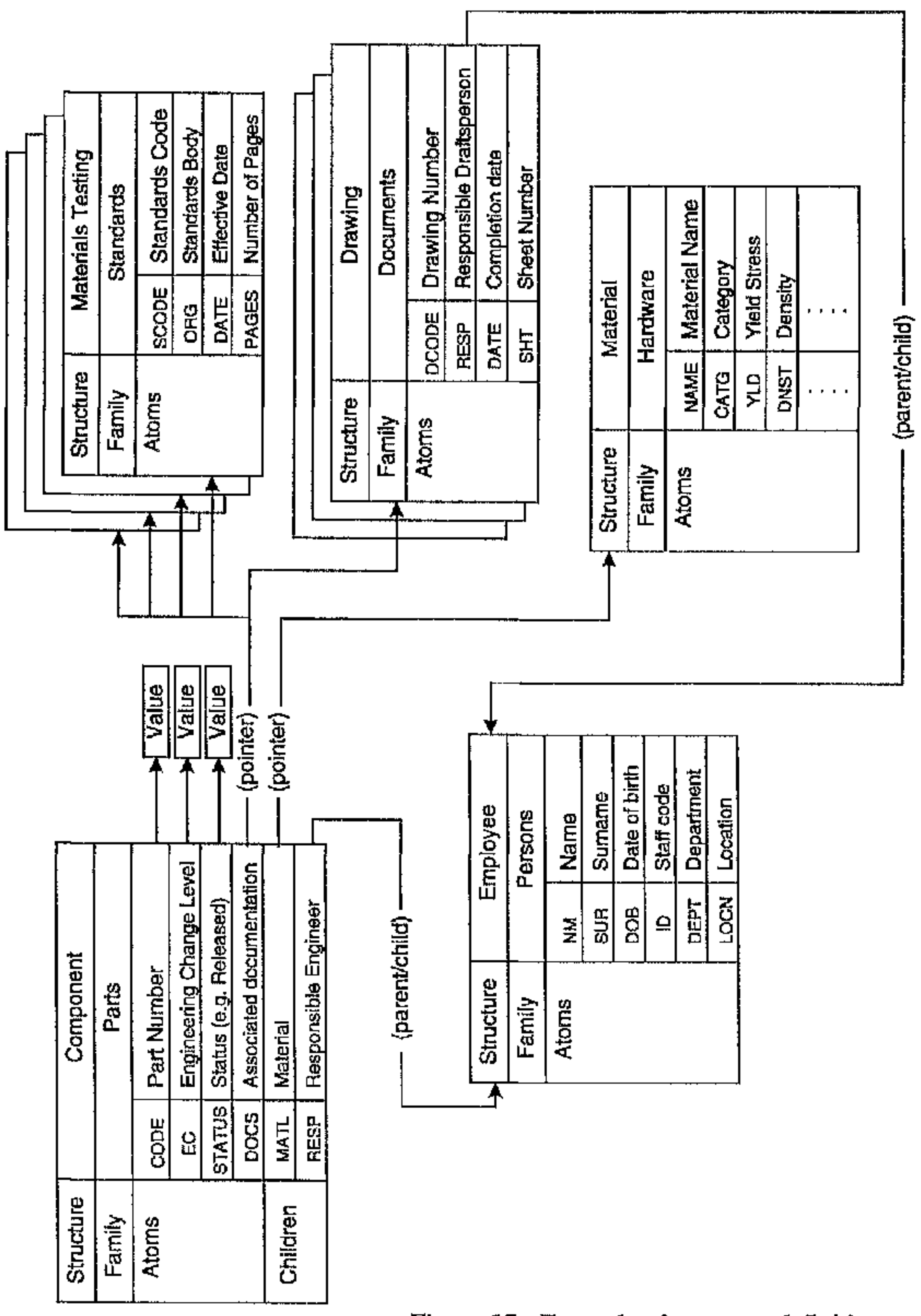


Figure 17 : Example of structure definition

Structures may be freely associated and GUIDE provides control mechanisms to manage the associations. As an illustration, the structure describing a component in figure 17 may be associated with any number of component structures: this allows the component structure to serve also as a representation of the assembly when so required. This characteristic of GUIDE structures allows them to be employed *cyclically* and this facility is a departure from object oriented methodologies. Associations of this nature may be described either by the use of *pointer* atoms or, explicitly, through *parent/child* relationships. The former method is easier to manage and manipulate; the latter requires additional controls and restraints to be defined in addition to the relationship[40] to safeguard the integrity of the construction. The use of poorly constructed structures employing parent/child relationships can be dangerous; the employment of well constructed and aptly constrained structures can be beneficial and, representationally, succinct.

4.2.1.4 Derived structures and atoms

In designing, entities occur which are *derived* from others, but have a different name or are relevant only under peculiar circumstances. They represent a *specialisation* of existing knowledge. Their representation is derived from the characteristics of existing entities[41] with some modifications (e.g. in their name) or extensions such as the association of additional constraints.

The principle of specialisation or *sub-typing* is a salient characteristic of frame based and object oriented methodologies. Specialised entity representations are accommodated and managed by GUIDE as *derived* structures and atoms. Derived structures possess the characteristics of the structure which provides them with their definition, but may have a different name and belong to a separate family. Derived forms inherit the contents and associations of their root[42]: the atoms,

[40]Consider for example, the case when a structure can be an assembly as well as a component by having a number of children of the same type as itself: the representation should have constraints associated with it to enable an evaluation of the capacity in which the representation is being used, i.e. whether it is a component or an assembly, and disallow the association of children to it if the former is true. While it is possible to define such a constraint, it is its **provision** which is **mandatory** to maintain the structure's integrity.

[41]In this context entities are used to denote frames or slots in frame based representations and classes or types in object oriented methodologies.

[42]A root structure is the *representational parent* of a derived structure. It is possible to have root structures which are of derived form.

children structures and constraints of the root structure are inherited by the derived form.

The derived forms can have additional elements associated with them. Table 6 provides a summary of the representational characteristics of derived structures.

Characteristic	Behaviour
Name	May be different to the name of the root structure
Family	May belong to a family different to the one of the root structure
Atoms	Inherits all the atoms of the root structure; additional atoms can be associated with the derived structure
Children structures	Inherits all the children associated with the root structure; additional children structures can be associated with the derived form
Constraints	Inherits all the constraints of the root structure; additional constraints may be specified for the derived form. Constraints are executed on the derived form first, then on its root. If the root structure is a derived form of another, then the constraint execution is propagated upwards through the hierarchy.

Table 6: Characteristics of derived structures.

Derived atoms are defined similarly to derived structures. They inherit all of the characteristics of their root, including any knowledge for investing them with values. Table 7 summarises their representational characteristics.

Characteristic	Behaviour
Name	May be different to that of the root atom
Disposition	Same as the root atom
Source	Same as the root atom
Initial value provider	May be different to that of the root atom; if none specified for the derived atom explicitly, the root atom *source* gets inherited
Mapping	Same as the root atom
Sharing	The derived atom is not linked to the structures which may be sharing its root
Action on change	Same as the root atom
Units	Same as the root atom
Constraints	Inherits any constraints associated with its root. Additional constraints may be associated with the derived form. Constraints are executed for the derived atom first, then for its root; constraint execution is propagated upwards through the atom definition hierarchy.

Table 7: Characteristics of derived atoms.

Derived structures and atoms provide economy in the construction of representations. Derived forms are GUIDE elements in their own right. Duplication is avoided while the integrity of the representations is maintained. Root structures and atoms may themselves be derived forms: the definition hierarchies are automatically maintained and the ability is provided for the individual descriptions to evolve over time. The derivation of elements is illustrated in figure 18.

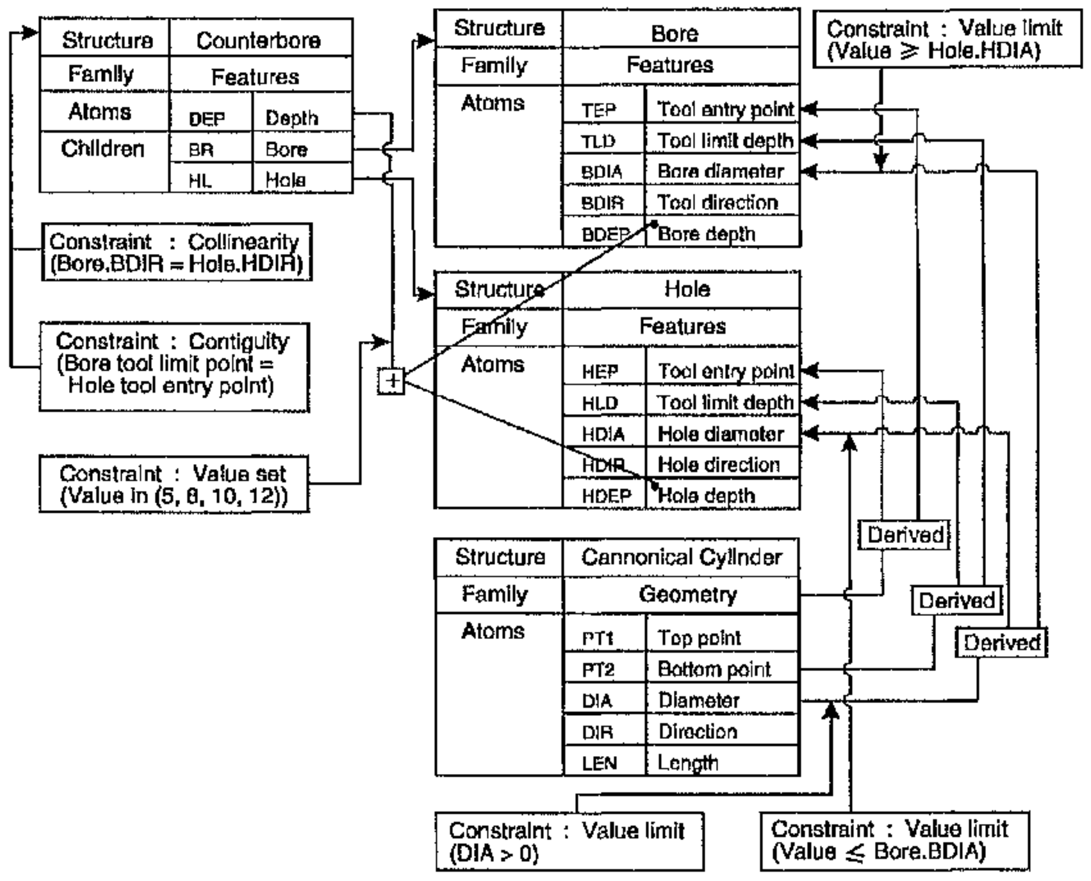


Figure 18 : Knowledge specialisation or entity subtyping

In feature based design representations, simple geometric definitions can be combined to form relevant higher level constructions. The definition of a counterbore shape using two simple cylindrical forms is shown in figure 18. The counterbore is split into the bored part and the hole; additional constraints to ensure contiguity and collinearity of the construction are associated either with the counterbore structure or with its components which are children structures. The bore and hole definitions make reference to attributes derived from a canonical cylinder. There are several advantages: the derived attributes are entities in their own right and inherit the characteristics of their parent definitions; furthermore, higher level constructions are linked implicitly by lower level inter-dependencies. The creation of a counterbore causes the concomitant instantiation of the geometric form used to describe it; any associations built into the geometrical definitions are imparted to any of their derived forms.

4.2.1.5 Activity and relationship structures

Activity structures in GUIDE are defined as collections of *events*. Events are associated with operations performed on structures, atoms and methods within GUIDE such as, for example, the setting of defaults values, the association of children structures with their parent, the modification of atom values and the execution of constraints. Structures and atoms provide the knowledge associated with a design or a particular aspect of it. The design record[43] describes the route followed in manipulating the knowledge and generating the design solution. Taken together, they represent the product descriptions.

Activities are distinguished from other structures through their membership of a particular family and the mandatory association of three atoms. Table 8 provides the definition.

Name	**Atom Value Type**	**Function**	**Constraint**
Requester	String	This is the person responsible for raising the requirement for the particular activity. Upon completion of the relevant activity this is the person which will be notified of the status of it and will respond accordingly by accepting or rejecting the final outcomes.	Must be persons empowered to assign tasks to others or themselves.
Actor	String	This is the named person chosen, by the requester, to undertake the activity. A special case results when the requester is also the actor: the resulting activity is considered as *private* and the GUIDE control mechanism treats it separately from the rest of the activities in the system.	Usually a subordinate of the requester or a subcontractor empowered to accept requests and will usually be chosen from a list compiled as the result of the specification of the requester.
Reason	String array	A description of what is required and what the goals, objectives or functional constraints might be.	Restrictions may apply to the format but, usually, it will resemble a specification document.

Table 8: Description of mandatory atoms for activity structures.

[43]The design record is constituted from activities and the relationships of them determined during design.

Any number or type of GUIDE structures and atoms can be associated with activities. Activities can be defined in terms of children activities defining, *ab initio*, the processes of a particular task: constraints on the activities can be used to assign priorities to the child activities and therefore, to dictate the timing and flow of operations. Highly sequential or simultaneous methodologies can be developed by these means. Figure 19 illustrates the concept of an activity within GUIDE.

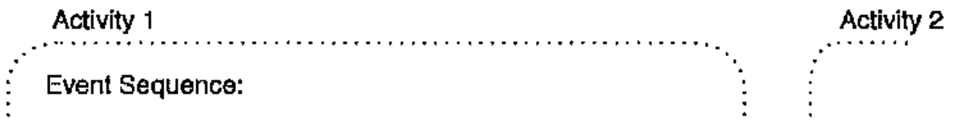


Figure 19 : Concept of an activity within GUIDE

Activities can be children of other GUIDE structures. This facility enables tasks to be performed organically with the manipulation of design entities. One application of this facility is the use made by GUIDE to pass requests between designers: for example, an attempt by one designer to modify an atom value instantiated by another designer generates an activity to manage the modification of the instance as a child of the current activity. The child activity is assigned to the individual responsible for creating the atom instance originally. All children activities must be completed before the activity requesting them can itself be declared complete.

Relationships are also special structures in GUIDE which define explicit links between structures and atoms. Logically, relationships should reference at least two GUIDE elements which provide the left and right sides of the relationship. The elements can be other GUIDE structures or atoms. Structures are associated with the relationship structure as children. Atoms can be pointers or take values.

Relationship structures define associations between the GUIDE elements they reference. Constraints on the elements of a relationship may validate the proposed associations or narrow the scope of the relationship. Relationships may take other relationships as their children which enables the construction of a *relationship tree*. The creation of a relationship can be validated by the association of a constraint with the relationship structure. Relationships can be dissolved and the disposal of the elements linked through the relationship structure is controlled by a constraint: whether or not the elements in the relationship are deleted as well, depends upon the definition of this constraint.

Relationship structures can represent *one to one, one to many, many to one* and *many to many* associations and link diverse data forms within GUIDE elements. The advantage of employing structures to describe activities and relationships is that the knowledge schema - the representational capabilities of structures and atoms - is retained and this reduces demands on the control mechanism.

4.2.1.6 Methods and method data

GUIDE associates sets of calculations or constraints with design entities and their attributes through the provision of *methods* or programmable procedures[44]. Associated with methods are the values which are inputs to and outputs from the algorithms involved. Method *inputs* and *outputs* can be linked to atoms and, in conjunction with the atom source definition, provide atoms with sets of possible values during instantiation.

Any software tool can be the basis of a method provided that its inputs and outputs can be identified and defined unequivocally. A method may incorporate any constraints which are incurred in its operation. For example, a finite element (FE) analyser can be described in terms of the element geometry and characteristics, boundary conditions and load cases it employs represented as

[44]To draw a parallel with high-level programming languages, methods can be compared to subroutines or functions: they are invoked under certain conditions, have data passed to them through specific variables and return the results of the computations in another set of variables.

inputs, and in terms of the stress and strain values calculated as outputs. The representation may include the condition that the FE method can only be executed as a consequence of a change effected either by the execution of a FE pre-processor or an alteration in the material properties or to the loading conditions used in a particular component analysis.

Methods are explicitly bound to atom definitions through their inputs and outputs. For example, an atom may be associated with the output of a method which instantiates it. This concept is illustrated in figure 20.

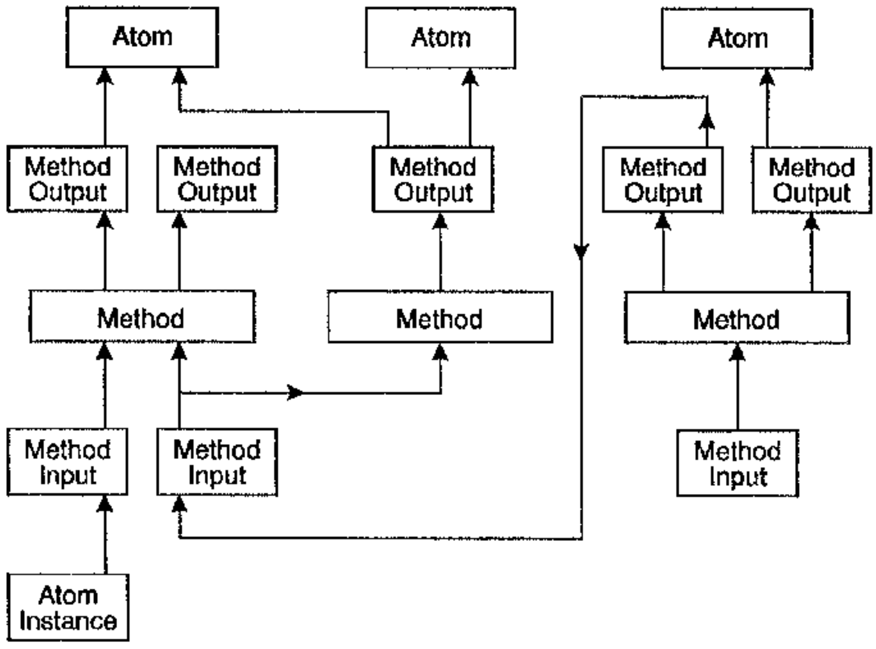


Figure 20 : Association of atoms with method data

Associations may also be defined between method inputs and atoms. The significance is twofold:

- the method execution is made dependent and conditional upon the investment of a particular atom with values;
- an implicit relationship is formed between the atoms involved: the input atoms which invoke the method are the *parents* of the calculated atoms.

The former is significant in terms of the flow of operations conducted using GUIDE: the execution of a method may cause GUIDE to request the instantiation of atoms it requires as inputs in advance of the execution of the actual method. The latter is significant in the presence of change: any change in the value of parent atoms must be followed by a recalculation of their dependants or, alternatively, the dissociation of dependants and parents and their resetting to initial (possibly default) values.

Direct links can also be formed between method inputs and outputs, as shown in figure 20. This type of association can be used to link methods together to create calculation cascades. A special case of this cascade occurs whenever a method input is linked to one of the same method's outputs: method iteration is effected.

Data associated with methods and atoms must be similarly represented. However, method data are value-oriented; their definition needs only to incorporate the relevant information (table 9).

Characteristic	Function
Type	Integers, reals, strings and arrays of them. *Pointer* types are not supported.
Nature	This denotes whether the data are inputs to or outputs from methods.
Mapping	This provides initial values. It can be used to bind data to GUIDE atoms. In certain instances, it can cause the execution of a method to retrieve possible default values from specific locations (e.g. a database).

Table 9: Characteristics of method data

Methods may have several outputs which can, in turn, be associated with a number of different atoms. The execution of a particular method to furnish a set of values for a particular atom may, therefore, cause concomitant instantiations. Figure 21 illustrates the concept.

The request to generate a value for the diameter of a hole causes the execution of a method to aid in the selection of a tool to form it: the method provides as output a value for the hole diameter, sets the particular tool characteristics and defines the manufacturing process to be employed. The method depends upon the provision by another structure of the material name for the component in which the hole is to be machined. The instantiation results in the material type atom being the parent of the hole diameter which in turn is the parent of the tool diameter, length and manufacturing process instances. The ability to associate method inputs and outputs with atoms is a powerful mechanism which can be used to generate and link several design parameters in a single operation. It provides one of the fundamental mechanisms available within GUIDE for the creation of interrelated values.

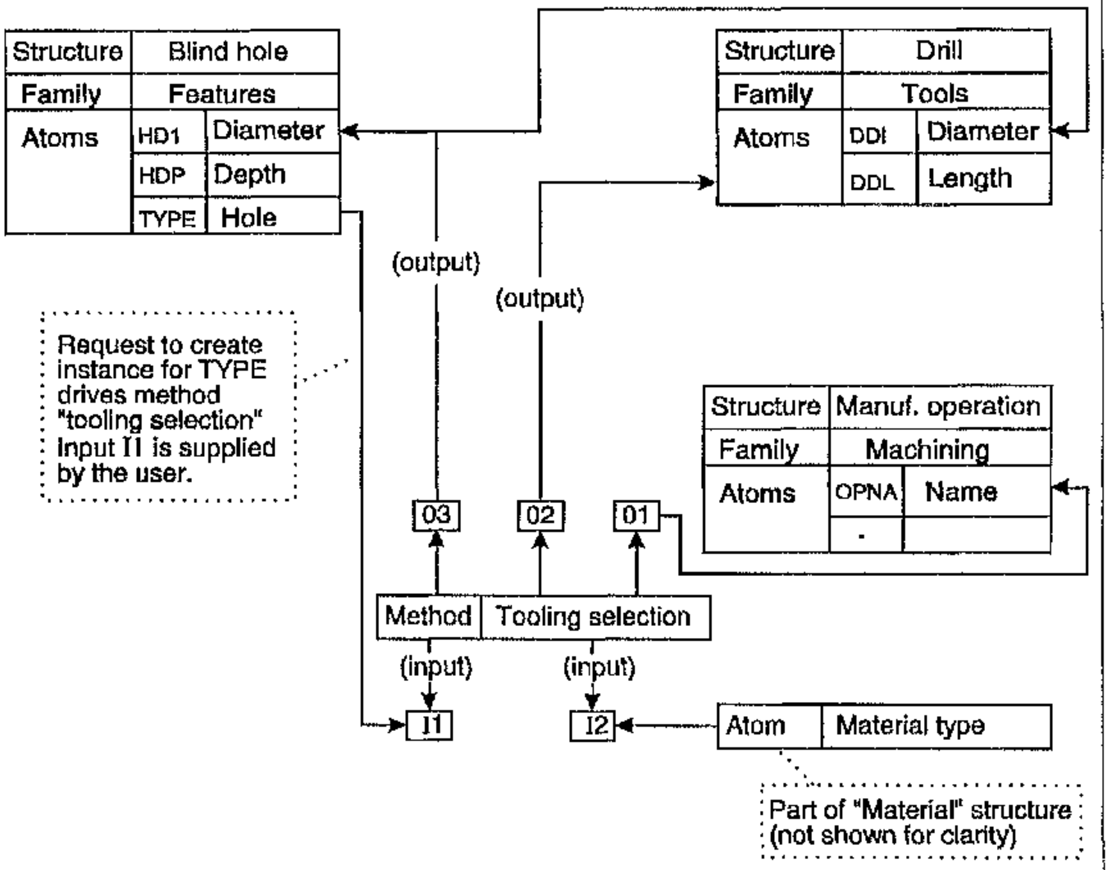


Figure 21 : Multiple instantiations derived from a single method

Constraints receive values which they proceed to validate. In essence, they have a single input variable and provide a single output which is used to determine whether the validation was successful or not; they can be represented in GUIDE as methods. Constraints are traditionally employed in a passive role, limiting a parameter to a range of possible values. In GUIDE, constraints are represented as active methods. The *constraining method* may be designed to incorporate a consequential action part, similarly to a rule in a production. For example, a passive constraint may be used during the process of selecting the set of possible tools for a particular shape to determine whether or not a process planning option has been chosen. An active constraint can detect that a process plan does not exist, cause its generation and subsequently allow the tooling selection process to continue. Constraints may invoke methods or instantiate atoms and structures as a result of their application.

The method definition in GUIDE maintains its validity irrespective of whether or not the output values it provides instantiate atoms. The function of methods is specified in their association with other GUIDE elements. Table 10 provides a description of the associations and their effects.

Type	Associates With	Using
Method	Atom	An explicit relationship between the atom and the method's inputs or outputs as well as an implicit relationship between the method data with the method itself.
Constraint	Atom	An explicit relationship between the atom and the method. The method input definitions are used to match the value to be audited to the appropriate data.
Method or constraint	Structure	An explicit relationship between the structure and the method.
Method	Method	An explicit association of one method's input data with the output values of another, or with itself to cause recursion; the methods are implicitly linked.
Constraint	Method	An explicit relationship between methods.

Table 10: Mechanism for association of methods with other GUIDE elements

The provision of an external relationship mechanism enables the definition of non-exclusive and non-prescriptive associations between GUIDE elements. For example, it enables several methods to be associated with a particular atom; furthermore, it allows several atoms to reference a particular method, thereby leading to more economical descriptions. The opportunity to define generic as well as specific techniques for the determination of values for atoms[45] is also provided. Table 11 provides a classification of methods within GUIDE according to the type of operation which they support[46].

The definition of methods is complemented by information on their *physical type*[47] and *physical location* to enable GUIDE to invoke a method by the appropriate mechanism and to facilitate distributed[48] operation.

[45]This is a substantial point of departure from the way methods or *daemons* are employed in frame based or object oriented representations; in these, the methods are firmly bound to frames or classes respectively. Particularly in the latter, the provision of a method within a class enables the definition of objects to be construed.

[46]The significance of the function of the particular methods and constraints within GUIDE is detailed in the section describing the control mechanisms (4.3).

[47]This is used to determine how they are to be invoked and is dependent upon the specific language of the facilities which are linked to them; details are given in section 5.2.1.

[48]In this sense, referring to an environment comprising multiple, distributed computational facilities.

Method Type	Operates Upon	Function
Method	Atom	Supply one or more values during atom instantiation.
Constraint	Atom	Validate the set of values associated with an instance of an atom.
Method	Atom	Supply one or a set of initial values to an atom.
Constraint	Structure	Determine if the structure selected is appropriate at the particular stage in the design.
Method	Structure	Supply initial values to one or more atoms referenced by the structure and create links between the structure and any of its children structures.
Method	Structure	Perform operations relating to the structure as a whole; the values of all the atoms and pointers to the children or linked structures are made available to it. This method can be applied as a constraint to validate the general structure by checking multiple elements of it simultaneously.
Constraint	Structure	Test the structure instance for completeness.
Method	Structure	Cause, upon structure completion, any additional actions to be performed.
Constraint	Method	Determine the appropriateness of the method selected for execution at the present stage in the design.
Database query	Structure or atom	Retrieve from a database system values which can be used in the instantiation of structures or atoms, or for the specification of initial values.

Table 11: Classification of GUIDE methods according to their function

The specification of methods and the uniform association of methods with structures, atoms and other methods is the basis for GUIDE operations and its overall control function.

4.3 Control mechanisms

The investment of structures and atoms with values is the basic aim of GUIDE operations. Values are sought for atoms and this operation causes the execution of methods and constraints appropriately to generate and validate these values. The acceptance of a value can cause other values or atoms to be instantiated, possibly by recursion.

Activity structures provide for the aggregation of operations which refer to instantiations or have been caused as a consequence of them. The sequence of events within an activity is associated with particular data: a record taken of these can facilitate the retrospective analysis of the constituent events and of knowledge. This also facilitates the formalisation of generic methodologies for dealing with the same or similar type of problem. The generation of a *design* *record* is fundamentally linked with the process of instantiation: events are thereby made organic to the processes of instantiation and recording.

Activities may be initiated freely by designers and involve the selection of a definition from a list of and its investment with values. Activities may be assigned by an authorised individual, the requester, to another who is qualified to accept the responsibility, the actor. The definition of the requester/actor combination can be controlled through a constraint placed on the activity or by a method associated with the *responsible* atom providing a list of options during instantiation. The method can acquire the value assigned to the *requester* atom and compile a list of actors which the requester can assign tasks. Figure 22 illustrates the initiation of an activity. It is possible to assign *group* values to the actor atom: this enables the association of a specified group or subcontractor with the activity. Individuals within the group may then contribute to the activity.

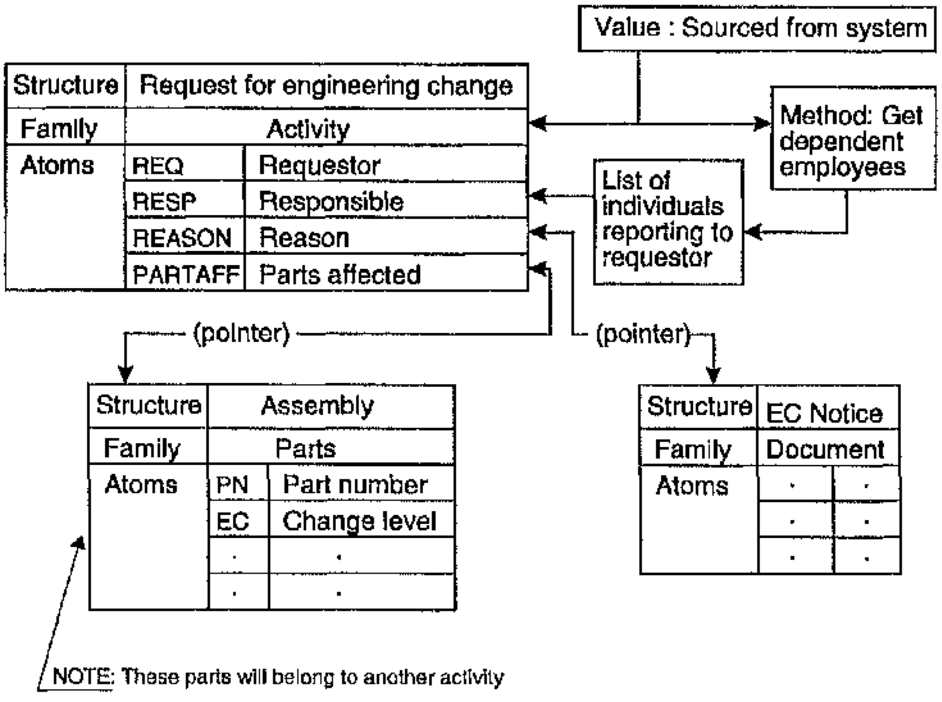


Figure 22: Initiation of an activity

Initiation of the engineering change request involves the requester choosing the structure and instantiating its atoms. The instantiation of the *responsible* atom causes GUIDE to invoke a method which produces a list of company employees capable of undertaking the task. The specification for the change is attached to the activity structure via a pointer atom. Finally, a structure representing the component to be changed is attached to the activity structure as a child. Completion of the instantiation effects the activity and communicates the request to the actor.

Generally, activity initiation involves:

- the allocation of an area to store the record of the events which constitute the activity;
- the notification of the activity to a central register;
- the dispatch of the request to the actor.

The register contains information on the status of the activity and associated data[49]; the options for the status value are listed in table 12.

Status	Description
Requested	The activity has been initiated, but no operations have been conducted. The actor has been notified.
Open	Operations are being conducted.
Suspended	Operations have been suspended.
Closed	The individual responsible has completed the activity and returned to the requester for verification. The requester can accept that the activity is completed or re-open it.
Committed	The requester has verified the completeness of the activity. The activity record is complete and cannot be re-opened or changed.

Table 12: Activity states

Only activities which have the status of **requested** or **suspended** may be selected and the responsible individual[50] alone is allowed to engage the activity. Activities cannot be opened by more than one individual simultaneously and an individual may only work on one activity at any instant.

Closed activities cannot be reopened except by the requester who must inform the actor of the action. When an activity is suspended the work which has been

[49]This associated information pertains to the way the activity record is stored and is discussed in detail in chapter 5.

[50]Or one individual in the responsible group.

completed is saved in temporary storage. Upon resumption of the activity, the results of the work are re-loaded.

4.3.1 Instantiation

The instantiation of structures and atoms is performed in three, often contiguous, stages: prepare, edit and commit. Figure 23 is a schematic of the process.

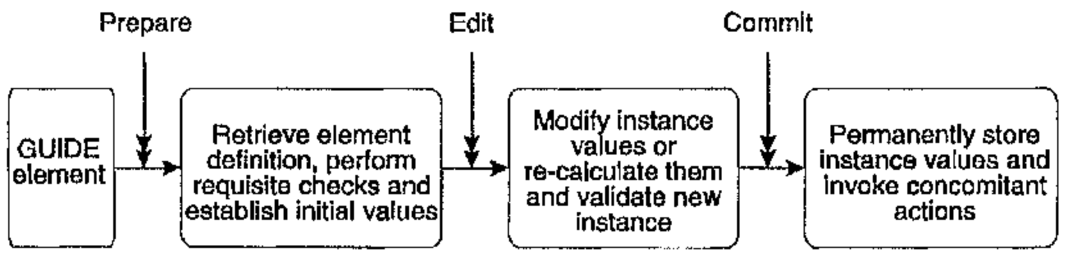

Prepare
Edit
Commit
GUIDE element
Retrieve element definition, perform requisite checks and establish initial values
Modify instance values or re-calculate them and validate new instance
Permanently store instance values and invoke concomitant actions

Figure 23 : Stages of element instantiations in GUIDE

While the flow of the operations in the instantiation of atoms and structures is specific to their definition, this scheme is applied to both cases. During the preparation stage the element definition is retrieved[51] and invested with initial values. Modifications to the value or content of the atoms and structures respectively are performed during the editing stage. The instantiated elements are written to permanent storage at the commit stage: this prevents any further modification of the values referenced by the elements.

Instantiation may be initiated by a request to operate on a structure or implicitly by an operation upon an atom referenced by a structure. Figure 24 illustrates this process within GUIDE. The operation commences with the compilation of a list consisting of all the structure instances within the current activity instance. The user of GUIDE may choose to work with an existing structure[52] within the current activity, or to create a new instance of it.

[51]The description of the repository storing GUIDE element definitions is detailed in section 5.2.

[52]Only structures which have been prepared but not yet committed are allowed.

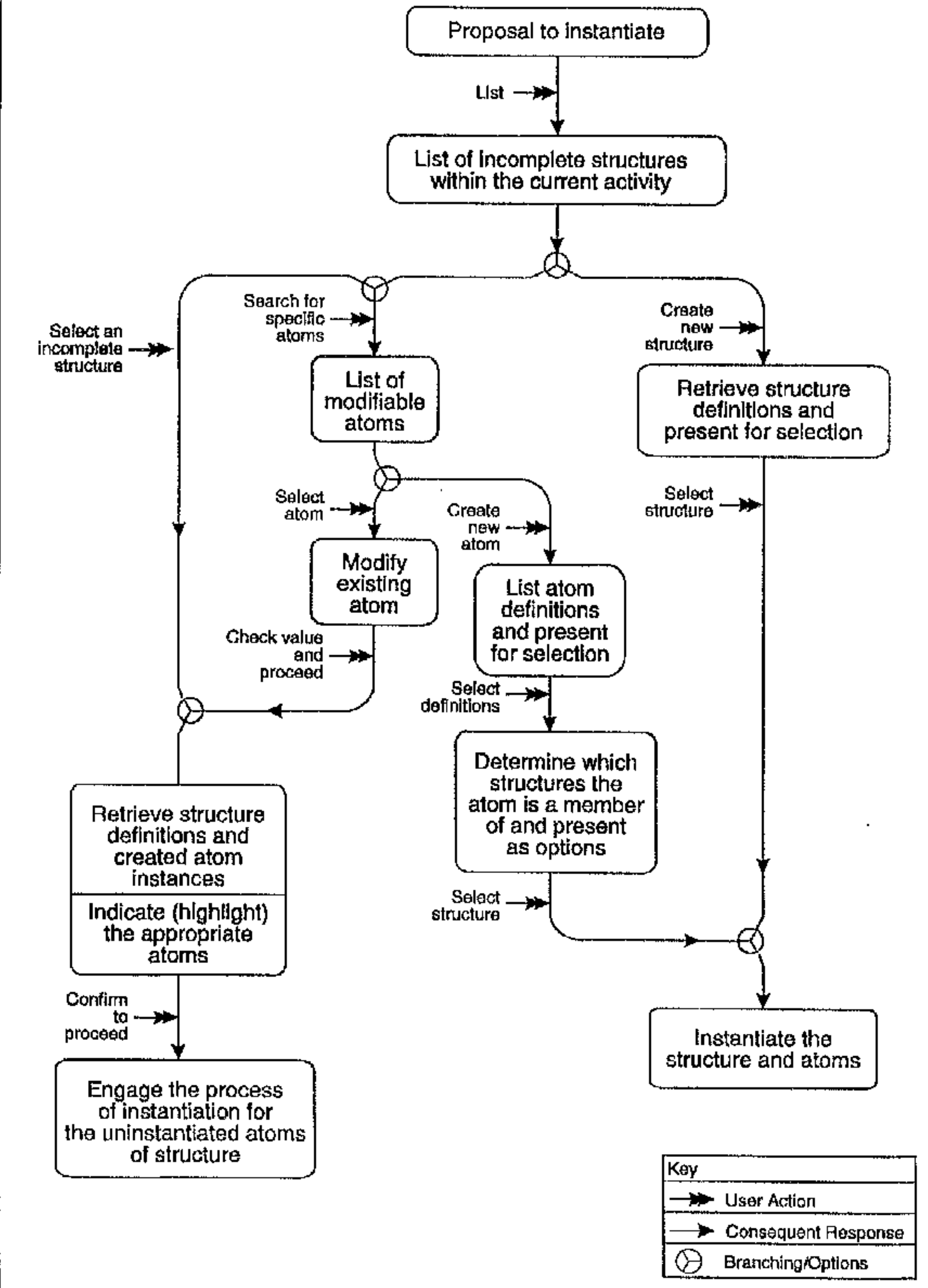

Proposal to instantiate
List
List of incomplete structures within the current activity
Select an incomplete structure
Search for specific atoms
Create new structure
List of modifiable atoms
Retrieve structure definitions and present for selection
Select atom
Create new atom
Select structure
Modify existing atom
List atom definitions and present for selection
Check value and proceed
Select definitions
Determine which structures the atom is a member of and present as options
Retrieve structure definitions and created atom instances
Indicate (highlight) the appropriate atoms
Select structure
Confirm to proceed
Instantiate the structure and atoms
Engage the process of instantiation for the uninstantiated atoms of structure
Key
User Action
Consequent Response
Branching/Options

Figure 24 : Initiating an instantiation

An indirect technique to cause the instantiation of a structure is to choose to instantiate an atom associated with it. A list of atoms within the current activity is compiled and, upon a particular selection, the list of structures associated with the given atom is generated. Alternatively, the choice to create a new atom instance can be made; this generates a list of the structure definitions which reference the given atom and, upon selection of an option, the structure is prepared for instantiation.

The control procedure for structure or atom instantiation permits the selection either of element definitions or of prepared elements respectively to create new or modify existing data. Elements which have been committed are excluded to preserve the integrity of the referenced values and descriptions which depend upon them.

GUIDE permits the creation of a new version of committed elements through the provision of a *copy* function. This involves the generation of an exact replica of a committed element or structure, the *receiver*, using values taken from a *donor*. The receivers are added to the current activity as editable entities. The copy operation is not restricted to stored elements: duplicates of editable (uncommitted) structures and atoms can be made routinely. The copies are always editable irrespective of the state of the source elements. The flow of the copying process is illustrated in figure 25.

The search for committed element instances may be restricted to the current activity or be expanded to all external activities which might be accessed by the individual. The choice of an instance from an external activity requires that a link be made between the activities involved through the GUIDE control mechanism. The record of the current activity contains the relationship between the atoms of the copied instance with their source by default; a record to indicate the dependency of a set of values to elements within the external activity is appended to it. The creation of the two links causes a bi-directional dependency: the receiving elements are considered to be children of their donors and their modification is subject to the appropriate restrictions; modification of the donating elements causes either a concomitant change in the values of their copies or a dissociation of the activities involved.

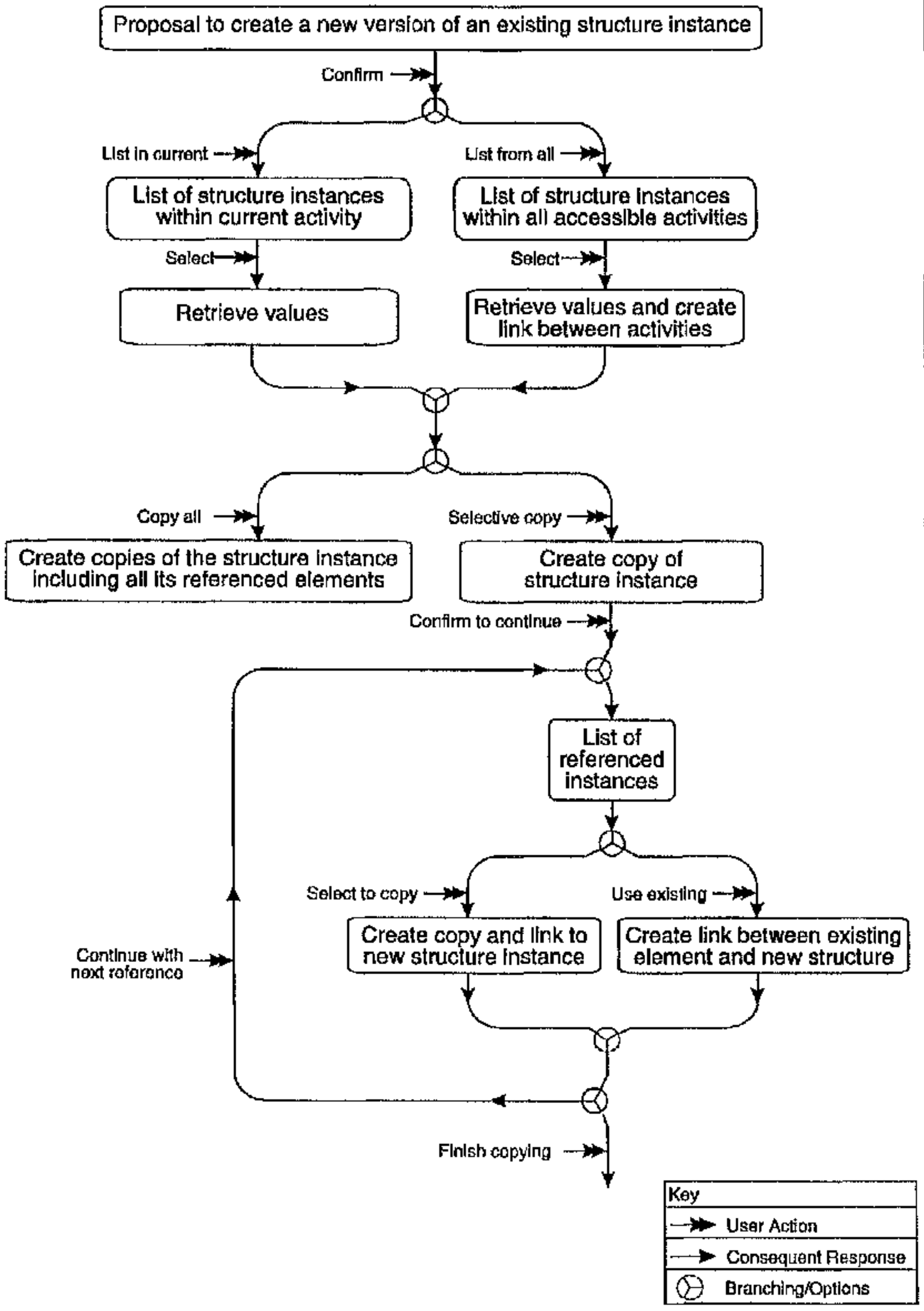

Proposal to create a new version of an existing structure instance
Confirm
List in current
List from all
List of structure instances within current activity
List of structure instances within all accessible activities
Select
Select
Retrieve values
Retrieve values and create link between activities
Copy all
Selective copy
Create copies of the structure instance including all its referenced elements
Create copy of structure instance
Confirm to continue
List of referenced instances
Select to copy
Use existing
Create copy and link to new structure instance
Create link between existing element and new structure
Continue with next reference
Finish copying
Key
User Action
Consequent Response
Branching/Options

Figure 25 : Creation of a new version of a structure instance

GUIDE searches the definition of the element, including its associations, to instantiate consistently and repeatably; its behaviour is determined by the particular definitions employed including branching if necessary. For instance, a completely unconstrained structure will always be prepared by GUIDE: any constraints upon it must be defined externally. The performance of any associated actions can be facilitated through the, predictable, constraint execution. The constraint can be coded explicitly to request, for example, the instantiation of another element or the execution of a particular method. Consequently, it is difficult to predict the duration or extent of any of the three phases of instantiation (figure 23).

The preparation phase can be used to generate a complete set of elements which can be committed without interactive editing of atoms or structures by the user. This mode of operation is appropriate for the instantiation of elements using *black box*[53] techniques. Alternatively, the editing phase can be used to instantiate the atoms and structures, thereby requiring the minimum amount of work to be done at the preparation stage.

The capacity to alter the flow of operations during instantiation is facilitated through the provision of suitable breakpoints within the prepare, edit and commit phases. In order best to capitalise on this capacity a thorough understanding of the conceptual flow of operations is essential. The following sections describe the salient features of these operations.

4.3.2 Element preparation

Element preparation consists of the generation of a new instance which is invested with initial values or dependent elements. The designer can either accept or reject the instantiated values. Structure and atom instances can be deleted provided that the instances are in an editable state. Furthermore, this editing phase can be used directly to ascertain the appropriateness of a given definition at the current stage of the overall design process.

[53]These techniques expose only the inputs and end results of a process; the process internal characteristics are irrelevant and hidden away, as if a black box.

3.3.2.1 Structure preparation

The emphasis during structure preparation is in providing a framework for the atoms and child structures associated with it. Atom preparation concentrates upon the provision of values and their association with appropriate elements and is elaborate by nature and takes place within the context of structure preparation. Atom preparation is a recursive sub-process controlled entirely by structure preparation and cannot be performed in isolation. Figure 26 provides an illustration of the flow of operations during structure preparation.

Structure instantiation begins with a search for applicable constraints[54] on the structure to verify its appropriateness in the current stage of design. The execution of the constraint determines whether GUIDE will proceed with the preparation of the structure. If the constraints succeed or are not defined, preparation proceeds to the next step. The minimum output condition this constraint returns is **true** or **false**. The pre-preparation constraint can be used to perform concomitant actions either by invoking one or more methods itself or by having, associated with its definition, a number of inputs to it.

Of particular importance are actions which can be requested of GUIDE at this stage. For example, a constraint may explicitly request the instantiation of another structure by invoking[55] the relevant GUIDE utility. In this case, preparation of the current structure is temporarily suspended and GUIDE performs the instantiation of the one requested. Upon completion, GUIDE resumes with the operations appropriate to the previous structure. There being no limits on the depth of nesting, the execution of a single constraint can cause a considerable number of associated instantiations.

Following the structure preparation constraint check, GUIDE retrieves the structure and establishes the identity of atoms and children structures associated with it. The definition together with additional information generated, for instance, for identification purposes are saved in a temporary storage. This stored information constitutes the *structure instance header*. For every atom associated with the structure the appropriate preparation process and consequent investment with initial values is engaged.

[54]This is a *pre-preparation constraint*. Its definition is described in section 5.2.1 and in table 21.

[55]Details of the invocation of GUIDE utilities by methods are given in section 5.1.1.

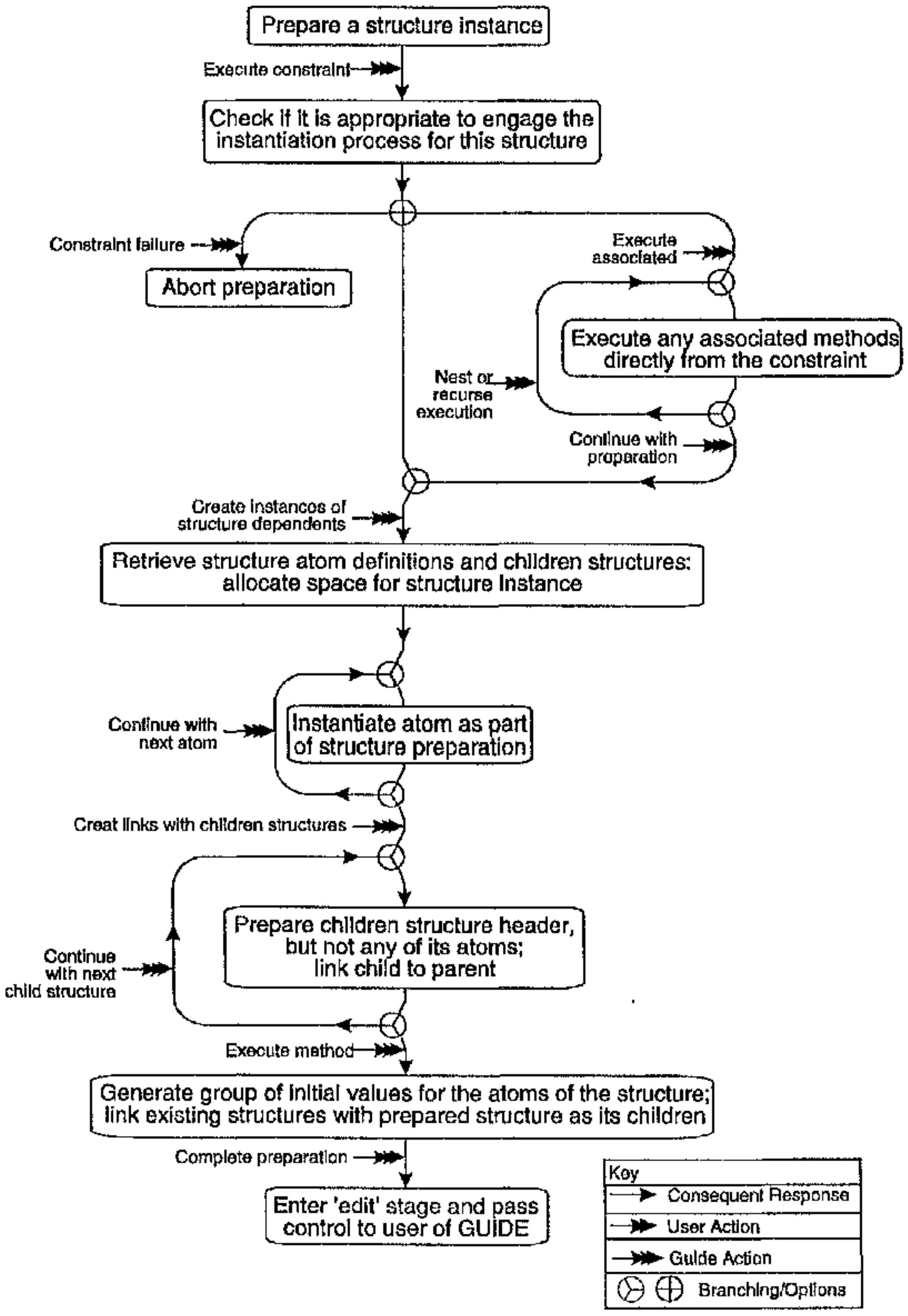

Prepare a structure instance
Execute constraint
Check if it is appropriate to engage the instantiation process for this structure
Constraint failure
Abort preparation
Execute associated
Execute any associated methods directly from the constraint
Nest or recurse execution
Continue with preparation
Create instances of structure dependents
Retrieve structure atom definitions and children structures: allocate space for structure instance
Continue with next atom
Instantiate atom as part of structure preparation
Creat links with children structures
Continue with next child structure
Prepare children structure header, but not any of its atoms; link child to parent
Execute method
Generate group of initial values for the atoms of the structure; link existing structures with prepared structure as its children
Complete preparation
Enter 'edit' stage and pass control to user of GUIDE
Key
Consequent Response
User Action
Guide Action
Branching/Options

Figure 26 : Structure preparation process flow

4.3.2.2 Atom preparation

The preparation of atoms is always performed in the context of structure instantiation. As a result, no check is performed as to the appropriateness of the atom. Figure 27 illustrates the flow of the atom preparation process. The atom definition is retrieved and, similarly to a structure, stored in a temporary area, along with some instance identification information. This constitutes the *atom instance header*. The process of preparation proceeds with the establishment of the mechanism to provide the atom default value; this information is provided in the atom definition, either through the specification of the *source* indicator or by means of an association of the atom with a method within the system. The possible options are described in table 13.

Source	Process Flow
System	Based upon the *mapping* value, a list is compiled comprising appropriate atom instances found either in the current or external activities. Upon an instance selection from the list, the atom initial value is set equal to the donating atom instance value and a link is established between them.
Any	A *static* (default) value is read from the atom definition data. This is specified at the atom definition time and is used as a safe fallback option.
Method	The definition of methods associated with the atom to provide it with initial values is retrieved and executed; the method output definitions are matched with the atom data type and the appropriate value is assigned to it. Any additional method outputs are discarded[56]. If GUIDE fails to retrieve or execute the method to provide the atom with initial values, it completes the value setting using the default specified in the atom definition data (same response as 'Any' above).
User	The response is the same as if the source was 'Any'.

Table 13: Determination of the atom initial value

Once an initial value has been established for the atom instance, a *status* value is set in the atom instance header to indicate the current value condition. The status is set as **valid & parent** if the initial value was copied from another atom instance, **valid** if it was calculated by a method or **from default** in all other cases. Pointer atoms do not have links defined by default in their definition and, consequently, only donor atoms or methods can be used to invest them with initial links: their status is set to **links defined** accordingly. If no links have been defined, or an error occurred during the process of determining the atom's initial value or links, the atom status is set to **undefined**.

[56]The additional method outputs may be used to instantiate other atoms. However, multiple instantiations should be under the designer's control and would be appropriate to conduct only during the edit phase. GUIDE therefore discards the values during atom preparation. The instantiation of multiple atoms during the preparation phase can be performed using the *group initial values method* described in section 4.3.2.3.

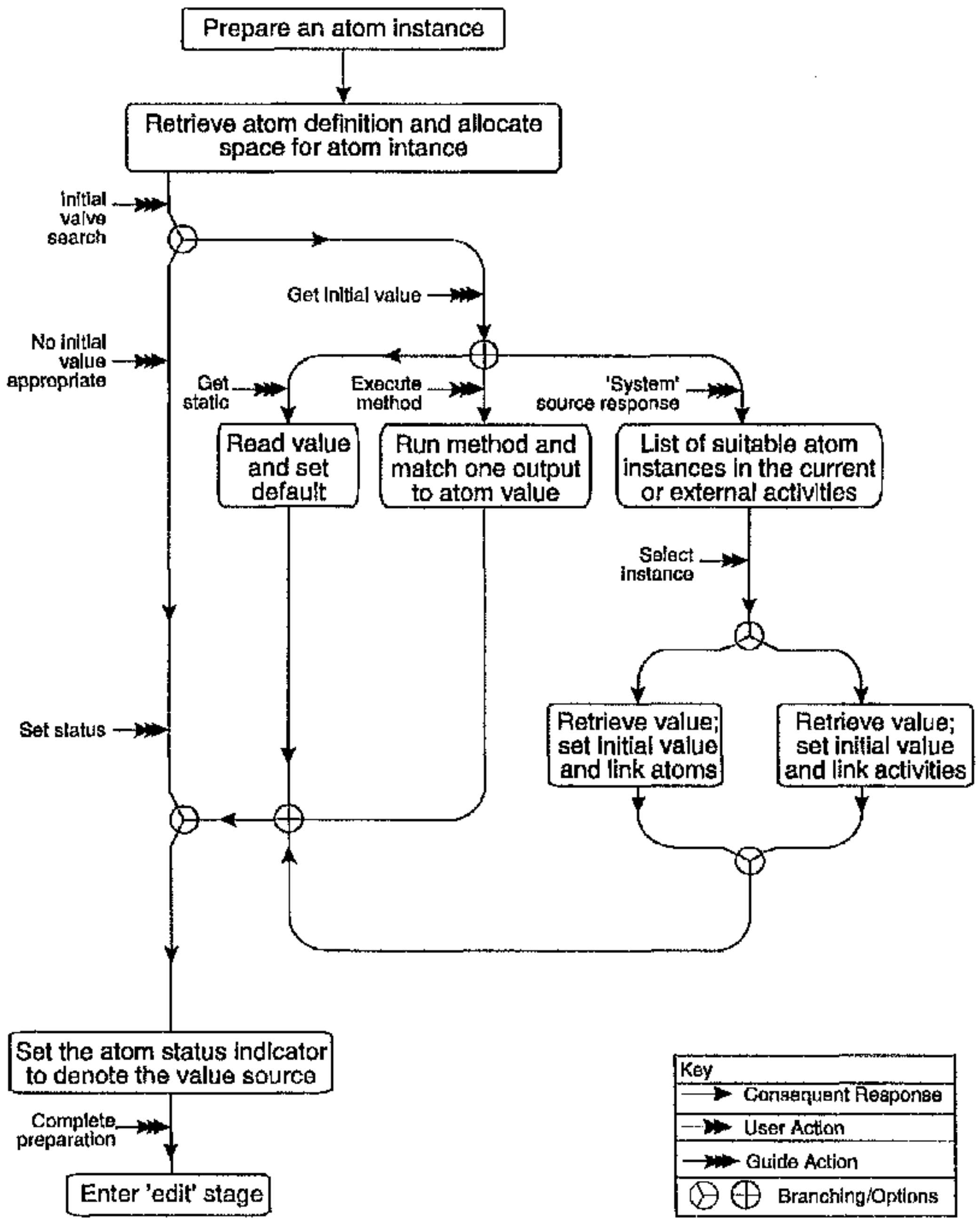

Prepare an atom instance
Retrieve atom definition and allocate space for atom intance
Initial valve search
Get initial value
No initial value appropriate
Get static
Execute method
'System' source response
Read value and set default
Run method and match one output to atom value
List of suitable atom instances in the current or external activities
Select instance
Retrieve value; set initial value and link atoms
Retrieve value; set initial value and link activities
Set status
Set the atom status indicator to denote the value source
Complete preparation
Enter 'edit' stage
Key
Consequent Response
User Action
Guide Action
Branching/Options

Figure 27 : Atom preparation process flow

Atoms whose status is undefined are treated as special cases. Prior to the completion of the preparation process, GUIDE searches within the current activity for atom instances of the same type as the atom for which an initial value is sought. If any instances are found, GUIDE searches for any inheritance links or other types of association (e.g. via relationships) between the possible donors and the receiver. If a relationship exists, the prepared atom instance value is copied from the related atom. The search for relationships is performed in the following order:

1. other atoms within the current structure instance;
2. atoms within any structures pointed to by other atoms within the current structure instance;
3. atoms within the parent of the current instance, if one such exists;
4. atoms within any structures pointed to by other atoms within the parent structure of the current structure instance;
5. atoms within any other children structures of the parent structure of the current structure instance;
6. atoms within any structures related to the parent structure of the current parent instance.

The search process continues upwards through the structure hierarchy until either a match is found or the top level is reached. If a value is obtained by tracking relationships, the atom instance status is set to **from parent**. The process of atom preparation is concluded by the creation of a link between the atom and the structure instance headers; this link enables the instances to be managed as aggregations.

4.3.2.3 Child structure preparation

Following the preparation of atoms in a structure, preparation (figure 26) continues by initiating the partial preparation of the structure's children structures. The child structure definitions are retrieved and used to create the structure instance headers, but none of their associated atoms. The consequences are twofold: a single control process can be employed to facilitate the preparation of structures, irrespective of their being parents or children and the option can be given to the designer to associate an existing structure within the current activity with the parent structure, instead of preparing a new child for it. Upon assembly of the child structure header a link is created with the parent structure instance header.

4.3.2.4 Completion of preparation

Structure preparation proceeds by the search for and, conditionally upon its definition, the execution of a post-prepare method. In essence, this facility provides an opportunity to adjust the atom initial values or children structure associations in advance of the structure instance being presented to the user. This *group initial values* method can be used to effect a number of actions concomitant with the operations on the current structure.

The preparation is concluded by the insertion of the status in the structure instance header. Table 14 describes the possible values the status can take.

Status	Meaning
Prepared	The structure has been fully prepared and can be edited.
Defined	The structure header has been built but no atom instances have been associated with it; the structure must be prepared before it can be edited.
Committed	The structure atom instances have been permanently stored; no modification is allowed of these values and, consequently, of the structure.

Table 14: Possible status values for a structure

Once the structure status has been set, the instance and all of its associations can be edited by the user.

4.3.3 Editing phase: Atoms

The editing phase of structure instantiation facilitates the modification of values assigned to the atoms. Values which have not been determined as a result of initial value acquisition can be set interactively under the user's control. Furthermore, associations can be defined between the structure pointer atoms and other instantiated elements. The setting or modification of a atom value and the assignment of links to a pointer are treated as equivalent operations: a single, context sensitive, control procedure is employed in their management. Figure 28 illustrates the flow of the atom modification process.

The atom definition is retrieved from the atom instance header to determine the calculation technique to be employed for the atom value. A preliminary check is made to establish that the atom value is in a modifiable state. If the atom status value is set to **stored** no further modifications can be performed and the process is abandoned. Provided that the value can be modified, the atom source indicator is used to establish the possible options to determine it.

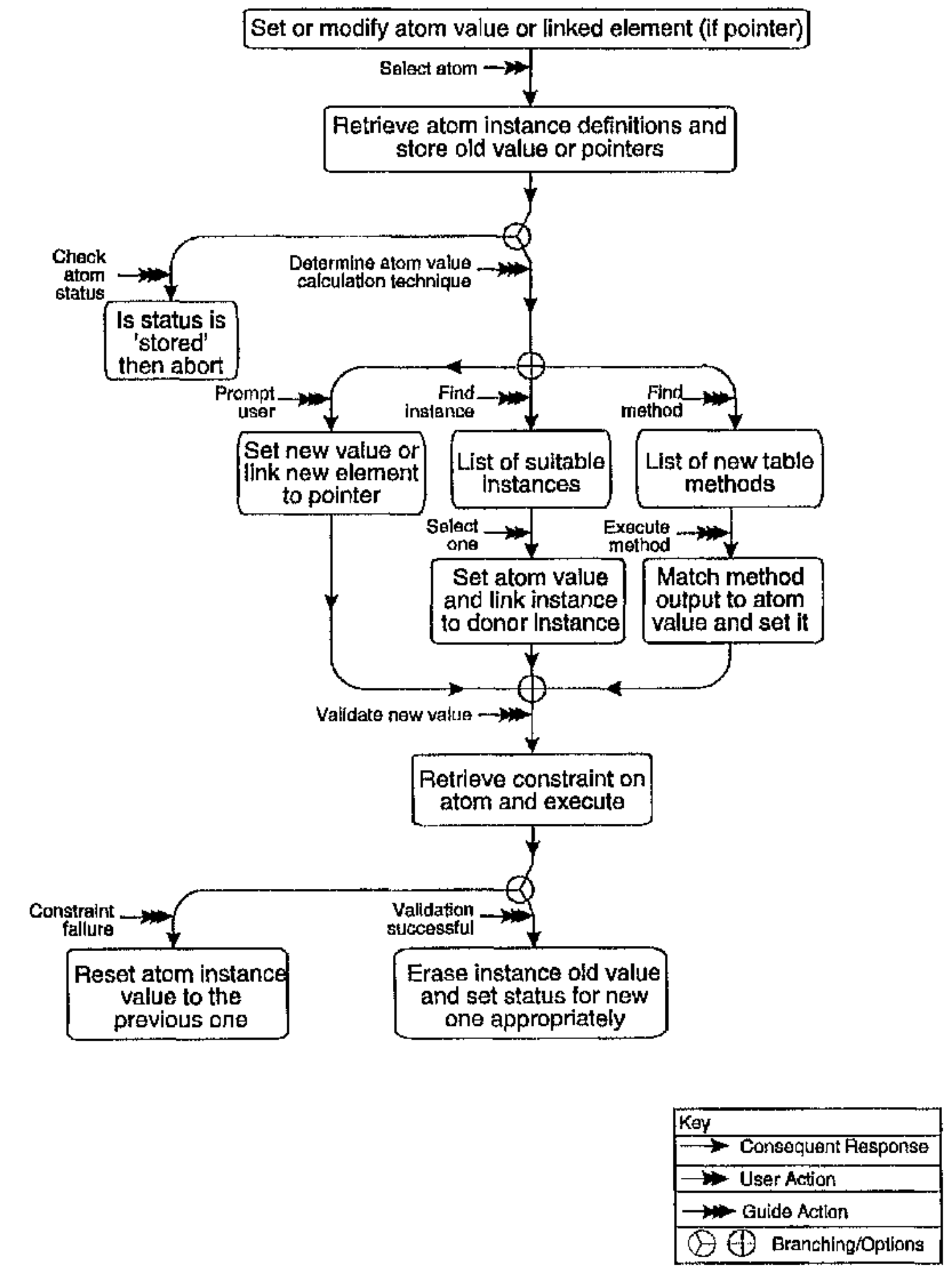

Set or modify atom value or linked element (if pointer)
Select atom
Retrieve atom instance definitions and store old value or pointers
Check atom status
Determine atom value calculation technique
Is status is 'stored' then abort
Prompt user
Find instance
Find method
Set new value or link new element to pointer
List of suitable instances
List of new table methods
Select one
Execute method
Set atom value and link instance to donor instance
Match method output to atom value and set it
Validate new value
Retrieve constraint on atom and execute
Constraint failure
Validation successful
Reset atom instance value to the previous one
Erase instance old value and set status for new one appropriately
Key
Consequent Response
User Action
Guide Action
Branching/Options

Figure 28 : Atom value setting or modification

A description of the possible flow of establishment or modification operations is provided in table 15.

Option	Flow of operations to establish atom value
User	The atom instance header is consulted to provide information on the atom type. This is used to differentiate, in the first instance, between pointer and value taking atoms and, subsequently, between particular data types. The atom type is used to set up a filter which determines the type of interaction to be requested of the user and the possible number of values or elements expected: the possible interactions are described in table 3. Multivalued (array) or pointer atoms cause the invocation of a particular procedure; during this, values can be added or subtracted from the particular array or elements can be linked to or dissociated from the pointers.
System	The atom header is used to establish the mapping value for the particular atom. The value is subsequently used by GUIDE to search through the headers of all the atom instances contained in the current and externally accessible activities. For any matches to this search the instance value is retrieved and added to a value option list. Upon completion of the search the user is prompted to select one or more values, as indicated by the atom type. A link is established between the donor and receiver atom instances to indicate that a dependency relationship exists.
Method	The method data definitions are searched in an attempt to match method output values to the current atom definition. The method outputs are in turn used to retrieve the methods which could supply the required atom instance value. Upon execution of the method the formatted outputs from it are matched with the appropriate atom.

Table 15: Atom value establishment options

Methods or constraints associated with particular atoms are executed as a result of requests to invest them with values. Figure 29 illustrates the flow of the process to invoke a GUIDE method.

The request for method execution causes GUIDE to search the definition of the method's data to compile a list of the values which must be supplied before the method can be executed. The interactions required to supply the method with values are determined by a procedure similar to that which applies to atoms: the method data source and type are consulted and used to define the appropriate interactions. Similarly to atoms, the possible value source can be provided by the user, from an existing atom instance, or from the output of another method. The latter option causes execution of another method. GUIDE suspends the execution of the current method, executes the new method and proceeds as above. The new method inputs may in turn require the execution of yet another method, in which case the process is repeated.

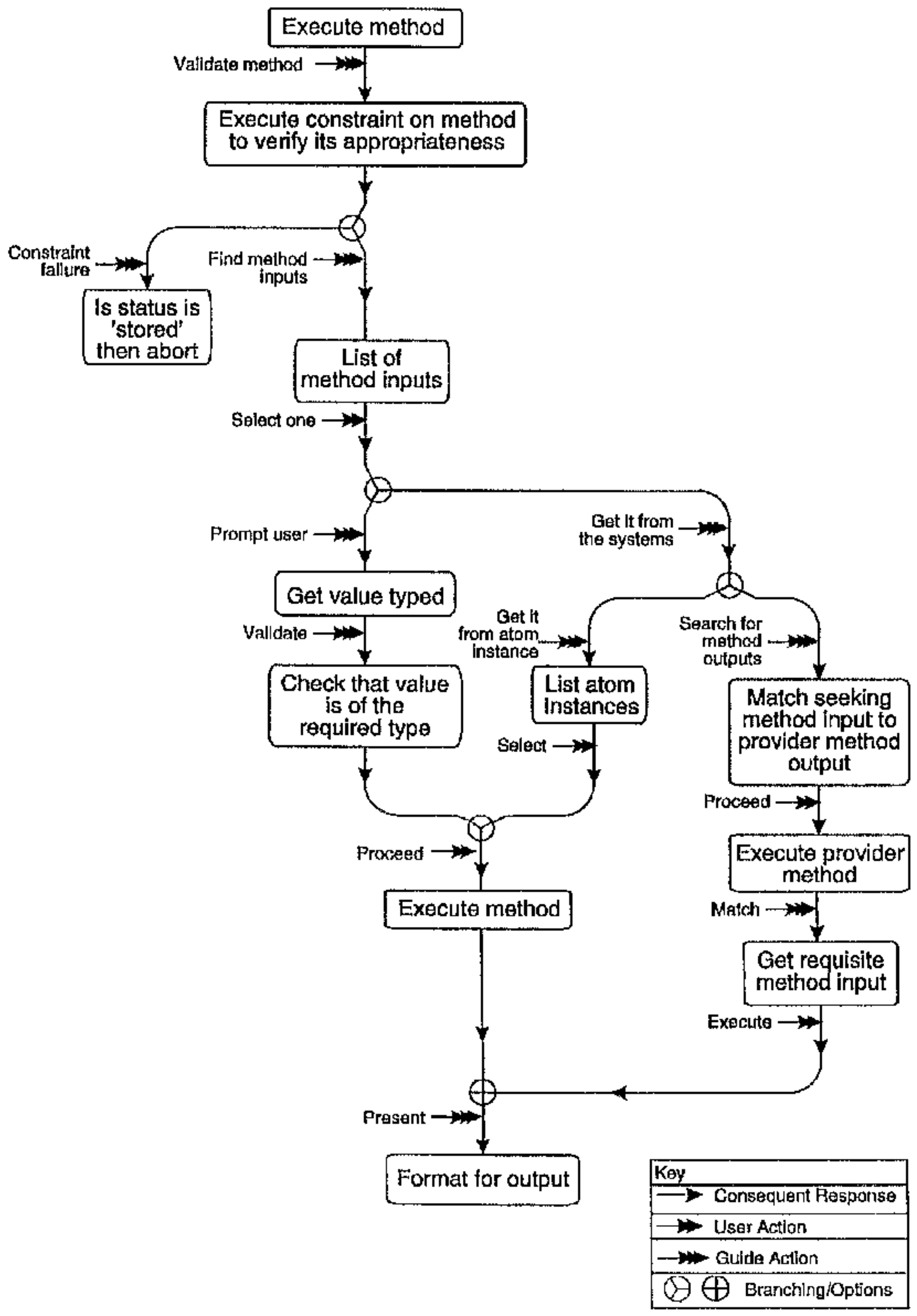


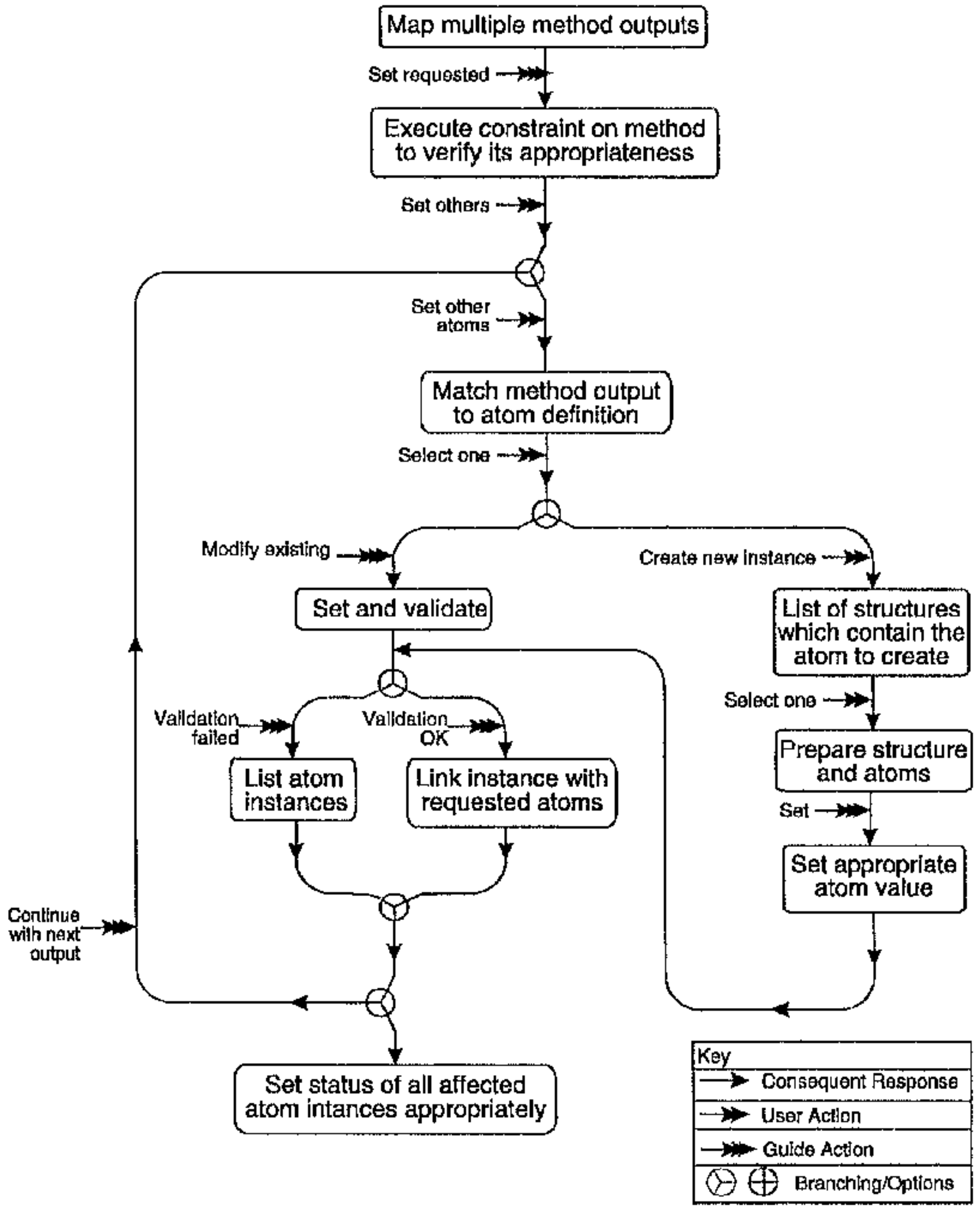
Figure 29 : Process flow for method execution

The ability to nest the execution of methods is a powerful mechanism for the determination of atom instances. Furthermore, it enables the factorisation of cumbersome calculations into discrete units which are easy to manage and invoke. The essence and applicability of individual methods or of their combinations is safeguarded through the associations defined between method inputs and outputs combined with constraints acting upon the methods. Constraints appropriate to a method are executed before it is invoked, thus enabling the suitability of the method to be validated in the context of the design task being engaged.

Method execution in GUIDE provides the possibility to instantiate several atoms through a single atom instantiation. The principle of *multiple value mapping* consists of matching the set of outputs generated by one or more methods during the instantiation of a particular atom with an equivalent number of atom instances within the current or associated activity. Figure 30 illustrates the process of multiple value mapping which is conditional upon the constraint affecting the requested atom, i.e. the parent.

Atom definitions are searched for every method output in the attempt to establish a possible match. Once a match has been found GUIDE searches the current and associated activities for instances of the particular atoms. If a match is found, the method output is proposed as a modification to the existing atom instance. If no match is found, the option is given to prepare a suitable atom instance and the structure which includes it. Upon completion of the preparation process, the method output value is proposed as a modification to the atom instance. The usual GUIDE mechanisms are invoked to validate the proposed values: the mapping of a value to an atom instance is equivalent to the process of value modification or setting.

The status of all the instances, the *children*, which have been generated as a result of the instantiation of a given atom is set to **valid & parent** and their header is linked to the appropriate parent atom instance header. The parent atom status is set to **valid & children** to indicate that dependencies exist upon the parent instance. Subsequent modifications to the values of either the parent or children instances are subject to integrity conditions. The modification of a child atom instance requires that any links from it to its atom parent instance are dropped. Modification of the parent instance is much more severe: the changes effected by its original setting must be undone. A list of its dependent instances is compiled and every item is dissociated from the parent instance and, subsequently, reset to a



Map multiple method outputs
Set requested
Execute constraint on method to verify its appropriateness
Set others
Set other atoms
Match method output to atom definition
Select one
Modify existing
Create new instance
Set and validate
List of structures which contain the atom to create
Select one
Validation failed
Validation OK
List atom instances
Link instance with requested atoms
Prepare structure and atoms
Set
Set appropriate atom value
Continue with next output
Set status of all affected atom intances appropriately
Key
Consequent Response
User Action
Guide Action
Branching/Options

Figure 30 : Mapping of multiple method outputs to atoms

safe default value. Structures which have been prepared as a result of the multiple method mapping process are deleted, provided that no modification of their other atom instance has taken place in the interim period.

4.3.4 Operations on structures

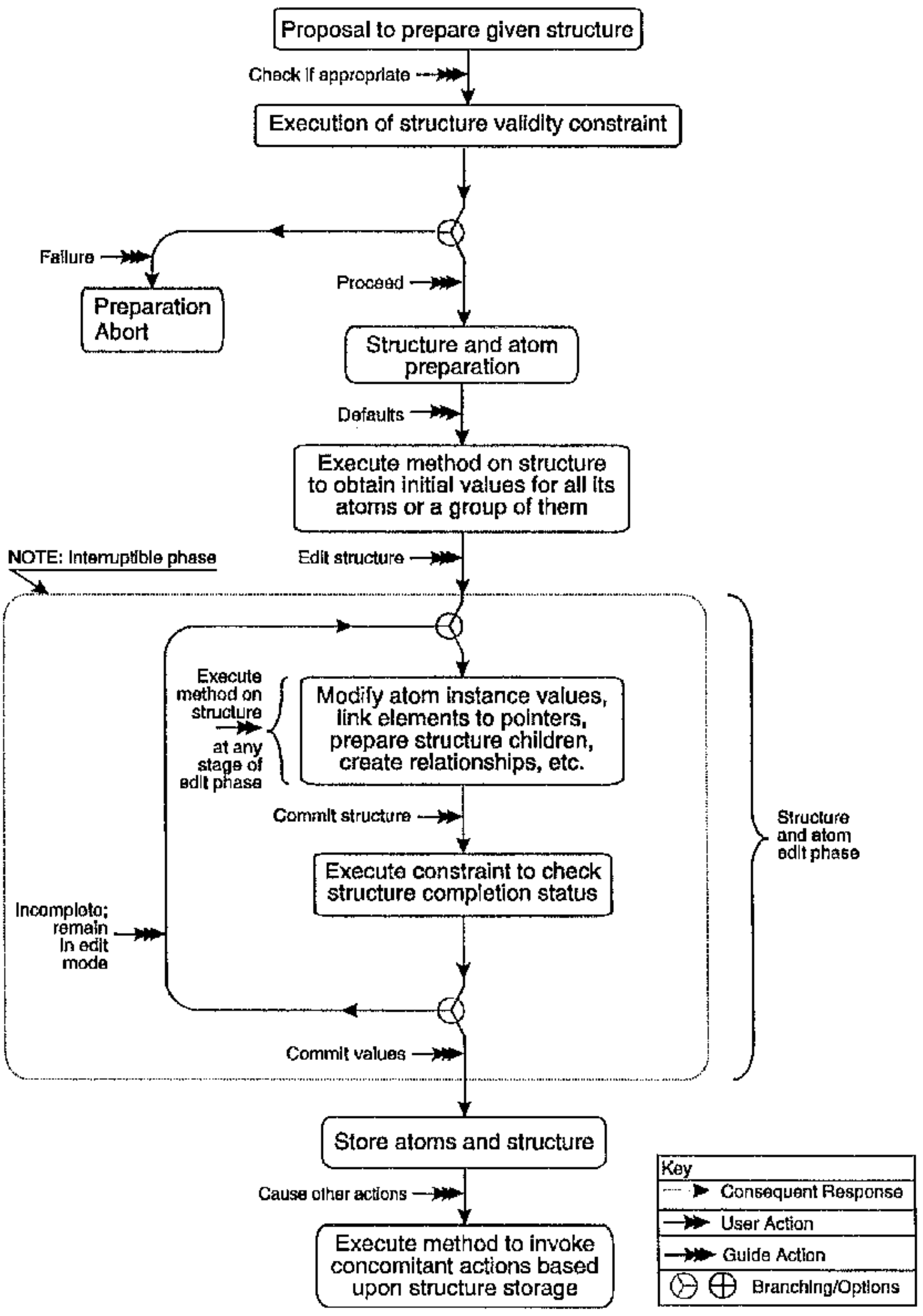
The preparation of a structure instance culminates in the definition of an entity which has editable elements in the forms of atoms or children structures. Figure 31 illustrates the operations which can be performed on a structure instance. Associated with the three possible phases of structure instantiation are several methods providing a constraining function and others producing actions expected by nature of the structure involved.

Constraints are executed at two different phases of the instantiation process: before structure preparation and in advance of the structure commitment. The former serves to establish the appropriateness of the structure instance creation and enables the association of a structure with the overall task which the designer is engaging. The latter serves to validate the instance definition in terms of the values it references. This constraint tests the completeness of the entity which the structure describes prior to the permanent storage of the data which constitute it. A structure is complete when the following criteria are satisfied:

- all the atoms in the structure have been instantiated to valid values;
- all the children structures have been committed.

The structure definition may be required to impose additional constraints on combinations of some or all elements associated with the structure. Since this is not predictable, the control mechanism of GUIDE cannot account for it. The accommodation of context specific requirements is facilitated by means of the completion constraint. This enables the specification of additional criteria and their incorporation in the test for structure completeness.

Methods may be executed during the editing phase of structure instantiation. A *group initial values* method is executable at the end of structure preparation to allow the modification of some or all of the initial values of the atom instances associated with the structure. This method is not restricted to the provision of initial values. Actions concomitant with the instantiation of the structure, such as for example the preparation of other structures or the execution of methods, can be caused by execution of the group initial values method.



Proposal to prepare given structure
Check if appropriate
Execution of structure validity constraint
Failure
Preparation Abort
Proceed
Structure and atom preparation
Defaults
Execute method on structure to obtain initial values for all its atoms or a group of them
NOTE: Interruptible phase
Edit structure
Execute method on structure at any stage of edit phase
Modify atom instance values, link elements to pointers, prepare structure children, create relationships, etc.
Commit structure
Execute constraint to check structure completion status
Incomplete; remain in edit mode
Commit values
Structure and atom edit phase
Store atoms and structure
Cause other actions
Execute method to invoke concomitant actions based upon structure storage
Key
Consequent Response
User Action
Guide Action
Branching/Options

Figure 31 : Operations on structures during instantiation

In the editing phase values referenced by a structure or by elements associated with it can be changed. Editing is the most interactive phase of structure instantiation and is often lengthy. Editing must sometimes be suspended while other functions take place: e.g. the instantiation of other GUIDE structures. The control mechanism allows suspension and resumption of editing at any time. A method employable in the structure instance edit phase must be executable at any stage of the edit process, as the user requires. This provision allows the structure to serve procedurally. Careful construction of these structures can lead to the description of powerful methods to be defined as structures[57].

Operations which are a result of the completion of a structure are accommodated through a *completion action* method. This is executed upon successful completion of the structure commit operation. As an example, the closure of an activity requires that it be checked for completeness. A concomitant action once completeness is confirmed might be that the requester should send the actor a notification that the task has been finished. The completion method can prevent the actor from closing the activity without any check from the requester or be used to close the activity transparently if the requester and the actor are the same person.

The representation of activities as GUIDE structures means that activities can be controlled by GUIDE functions. Furthermore, representations can be built which describe entities used in design tasks as well as the methods which are engaged in the design task. In this way, structures can be used to describe a diverse range of processes all within the control capacities of GUIDE.

4.3.5 Compilation and management of the design record

A record of the actions taken, the methods employed and the results generated would enable the regeneration of past designs by following the track originally laid out during product development. GUIDE keeps this record. The following actions, taken by the user or caused by GUIDE, are recorded:

- structure instantiation;
- atom instantiation;
- constraint and method execution;
- sourcing of atom initial values;

[57]This is analogous to *action frames* in frame based representations.

- sourcing of atom values during their setting, modification or linking;
- sourcing of method input data;
- mapping of method output data to atoms;
- relations between atoms;
- resetting of atom values to defaults;
- relations between instances in different activities.

Included in the record are links to stored instances, incomplete information (such as uncommitted structures and atoms) and particular values which relate to so far unmapped method inputs and outputs. This mix of process information and of the data produced requires that the control mechanism can distinguish *actions* and data derived from actions.

Instance	Abstraction
Activity instance	These are transformed into a single structure which references, by association, all the top level structures which were defined within it as children. In future invocations, the request to instantiate the generic structure which was abstracted from the particular activity safeguards the instantiation of all the requisite structures within it.
Atom instance	Creation of a derived atom based upon the definition of the original atom. All of the characteristics are inherited from the definition of the original atom except the initial value and the source indicator. The source indicator is pointed to the defining atom instance which provides the derived atom initial value.
Atom default	The atom instance value is defined as the default value for the derived atom.
Atom source	The source (user interaction, other atom, particular method) used to provide the ultimate value of the instance is used to set the source indicator variable and the mapping value. If a method was used, the method is specified as the atom default method and will be executed upon a request to instantiate the new atom.
Atom constraint and methods	These are inherited from the definition of the derived atom root.
Structure instance	Creation of a derived structure based upon the original structure which, subsequently, becomes the root.
Pointer atoms	The elements associated with the pointer atoms within the structure are used to provide the defaults for the derived pointer atom in the derived structure. The defaults specified are by type rather than by instance.
Constraints and methods	The definition of these is inherited by the derived elements through their respective roots.
Method data	If the source for these was an atom, then their source indicator is modified to point to the new mapping. This new specification does not prevent the user from specifying an alternative source. In all other cases, the source mapping gets modified to point to the particular activity which can supply the method input value as a default.

Table 16: Transformation rules for design record manipulation.

The instantiation of a structure is considered as an action; the action data associated with instantiation refer to the structure, the requester, the time of the request and links to the action which first caused the instantiation to be engaged.

The instantiation of an atom is an action caused by the instantiation of its parent[58]: the action data refer to the parent action. GUIDE employs the record of this information to construct history trails which can be retraced to create new instances by employing the same processes as were previously engaged.

Complete methodologies can be abstracted from the trail of a group of actions. To achieve this, GUIDE employs the activity record and a set of *transformation rules* described in table 16. GUIDE presents the user with the abstract in the form of a *journal*. After the journal is edited/modified any new structures or element definitions become available for future GUIDE operations.

4.4 Summary

GUIDE represents knowledge using a development of the frame concept which is symbolically rich and appropriate to the representation of stereotypes. GUIDE's elements - structures, atoms and methods - are employed to represent entities and processes in design and facilitate the description of products, technologies and generic or specific methodologies. The elements are implemented using a conventional relational database system.

GUIDE methods impart interpretational simplicity to the methodologies which employ them, provide flexibility in application and extensibility of function. Because methods can be made unitary in their function, they can be implemented readily through easily understood control mechanisms and may be applied recursively. Methods can be effective in bounded and open operational circumstances: they enable the representation of design constraints, company methods and enabling technologies.

Design activities are represented as special GUIDE structures. Activities and the relationships of them determined during design constitute the design record. The knowledge associated with a design provided by structures and atoms and the design record - the route followed in manipulating the knowledge and generating the design solution - represent the product descriptions. The compilation of the design record enables GUIDE to contribute and acquire design knowledge.

[58]*Parent* in this instance denotes a structure or another atom.

GUIDE represents relationships of physical entities and processes as structures. Relationships define explicit links between structures and atoms, link diverse data forms within GUIDE elements and decouple dependencies from the representation of entities and processes.

The control mechanism of GUIDE adopts a unitary approach and is implicitly recursive in its operation. It can invoke external applications to: establish default data; create control rules; link its internal representations with external data; manage user interfaces. As a consequence, GUIDE does not demand the formal methods to be understood by the user and can support highly structured (sequential) methodologies as well as more open application.

Implementation Of The Elements And Concepts Of GUIDE

5.1 Operational characteristics

GUIDE aims to enable the operations of engineering companies by supporting the concurrent exercise of distributed expertise. GUIDE provides an operational ethos and the knowledge engineering and control facilities to secure this; it depends upon:

- stored definitions of the elements - structures, atoms and methods - which it employs to represent knowledge of entities, processes and their relationships;
- the dynamic consultation of the element definitions following a request to operate upon them;
- standard mechanisms for the storage and retrieval of the element definitions and the data or information generated as a result of their exercise;
- the ability to interact with existing company resources.

GUIDE employs, as far as is possible, standard, commercially available facilities. Routine data base management systems provide for the manipulation of data through standard interfaces used by engineering enterprises and bring instant benefit from access to standards and product information they keep. The integrity of the definitions employed by GUIDE is fundamental and data base management systems provide the facilities to safeguard integrity. The speed of access to the definitions is insignificant compared with the substantial length of the process investing them with values. Overall, the use of standard data base management systems has several consequences:

- GUIDE's element definitions can be shared by many individuals simultaneously;
- different levels of authority can be specified for the management of the definitions;
- changes to the element definitions or the data referenced by them are conducted using standard facilities;
- changes to the element definitions take immediate effect and, as a consequence, the integrity of the data generated can be made secure.

The benefit of employing this particular storage methodology is twofold: there is no separate indexing facility and, therefore, no requirement to provide a specific control mechanism to manage it.

GUIDE maintains a record of the information generated and the processes responsible for its creation. This record constitutes *meta-knowledge*[59]. Use of meta-knowledge leads to the compilation of *indices* which refer to the actual knowledge definitions. In GUIDE meta-knowledge is expressed in the creation of links between data and knowledge or the generation of additional definitions. The individual *design records* are an appropriate location for keeping links between knowledge and data. The incorporation of new definitions within the existing scheme further enhances the dynamics of GUIDE.

5.1.1 Invoking GUIDE

The primary function of GUIDE is to control processes and it should be invoked separately from other software tools, but have the capacity to communicate with them as a peer. GUIDE is a highly modular software environment which can be invoked[60] by other software tools through an *Application Programming Interface*[61] (API). GUIDE's routine operation is independent of other software tools and, in multi-tasking environments, it can cause and control the execution of other software packages, such as for example a geometric modeller, using facilities of the operating system.

In more restrictive environments, GUIDE can be invoked as a sub-process; for example, GUIDE can be engaged from a CAD package via a call to the GUIDE API from a user exit; the user exit may request the instantiation of a particular structure which can in turn cause the execution of a particular software tool to assist the instantiation. GUIDE is invoked as a sub-process in the first instance, but the nature of the request causes it subsequently to become a process controller. Figure 32 is an illustration of the concept.

[59]Knowledge about the manipulation of knowledge.

[60]This can be achieved via user exits - predefined external routine calls which most software tools make during their operation. User exits can be coded by the users of software tools to achieve functions which are not normally available within the tool or to customise existing functions. For example, user exits are frequently employed by the users of finite element solvers to facilitate the definition of models describing characteristics of particular materials which are not formally provided in the solver.

[61]An *Application Programming Interface* or *Externally Callable Interface* provides the ability for software programs to acquire services which they require by communicating with the software tool which can provide them via a programming interface.

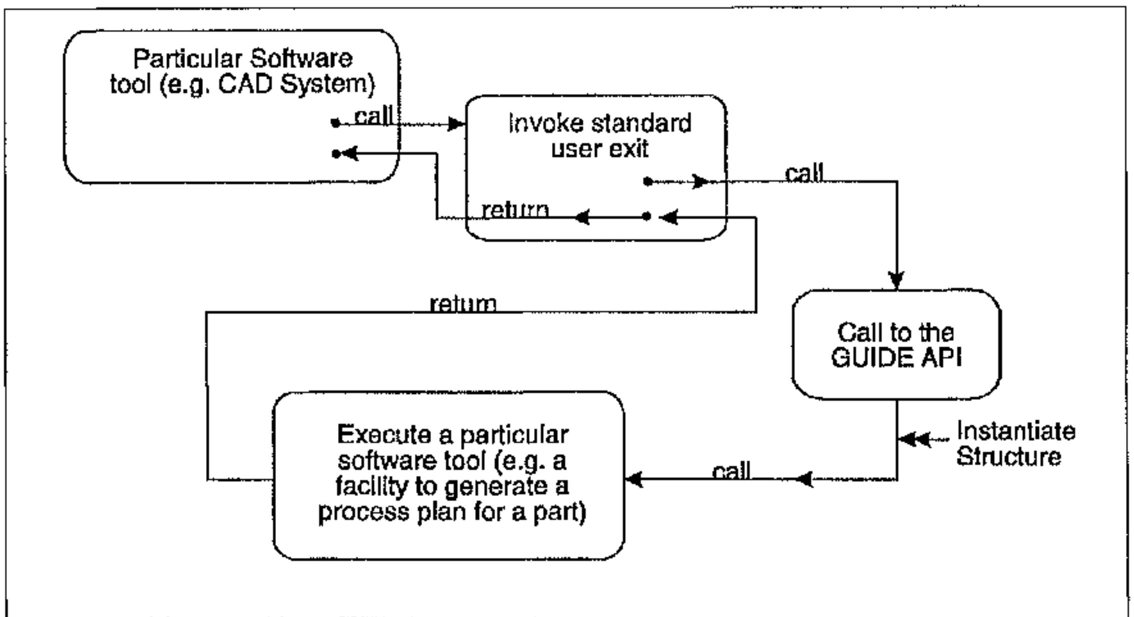

Particular Software tool (e.g. CAD System)
call
Invoke standard user exit
call
return
return
Call to the GUIDE API
Instantiate Structure
Execute a particular software tool (e.g. a facility to generate a process plan for a part)
call

Figure 32 : Invoking GUIDE as a subprocess

The peculiar construction of GUIDE has two important consequences:

- the engagement of a particular GUIDE function can lead to its being called recursively;
- methods invoked by GUIDE can take advantage of its unitary operation and, subsequently, call the application programming interface to perform routine functions (e.g. instantiation), thereby causing recursive invocation of GUIDE .

5.1.2 Distribution

In order to fulfil GUIDE's objective to aid the conduct of evolving engineering operations, it must be able to invoke methods and manipulate knowledge and data at locations other than the one where it is currently being exercised.

Recent advances in distributed computing - the development of the client/server model - have led to the emergence of methodologies for the communication of data across different locations and operating environments. The Remote Database Access[62] (RDA) activity is an attempt to define a protocol for the cooperative inter working of disparate, distributed database management systems. Vendors of systems which will be affected by emerging standards are reluctant to conform

[62]The RDA activity is conducted at an international standards level. Documents ISO/IEC 9579-1:1992, ISO/IEC 9579-2:1992 and ANSI X3.217-1992 describe the underlying interfaces (formats, protocols, function) to enable programs in a distributed computing environment to access data using a remote database manager.

with the standards in their entirety but implement those parts of the standards which are of direct relevance to their products, or else provide their own interpretations of the standards. Using vendor implementations of data base systems could, potentially, hinder GUIDE's ability to manipulate distributed data; the development of an RDA compatible data access method in GUIDE provides a 'safer' solution.

GUIDE adopts a practical approach to the communication of data between application software and distributed data base management systems. It employs methods which rely upon *de facto* facilities currently provided by the client/server computing model. The concept is illustrated in figure 33. The application software communicates the data manipulation requests to a local server process - the *request server* - which is responsible for locating the information and managing the communication with the appropriate data base management system. The request server communicates with a remote process - the *data server*. The data server is responsible for validating the request, communicating with the local data base management system to service the data requirements and, upon completion of the operation, the transmission of the data to the requester.

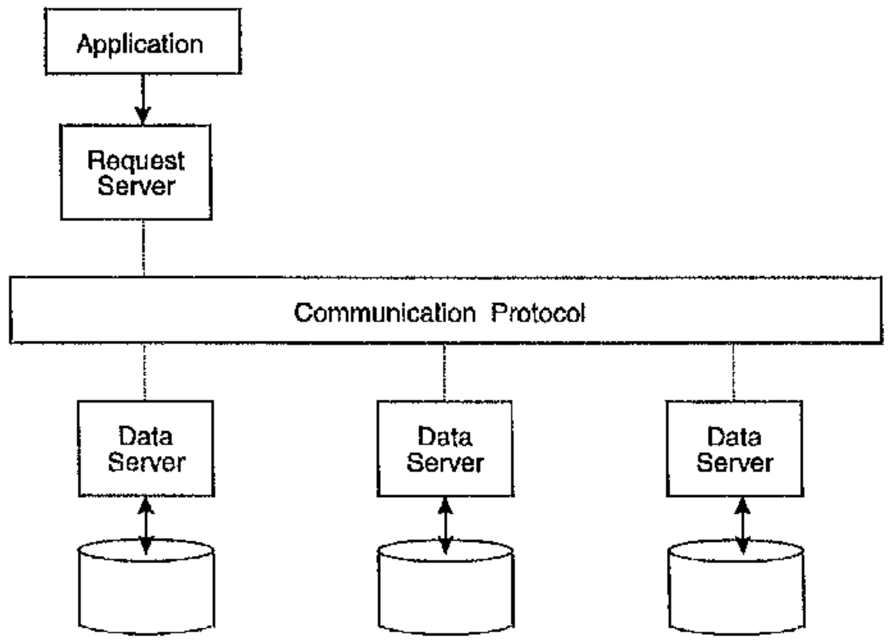


Figure 33 : Access of distributed data base systems

The request server and data server provide the linking functions between the application and the data base management systems, including appropriate transformations to the data exchanged. It is the responsibility of each server to transmit information appropriate to the destination environment[63]. This extends to the provision of request amendment facilities to eliminate syntactic inconsistencies between the sender and receiver[64].

The servers rely upon standard operating system utilities[65] to communicate with each other and the functions provided are extrapolations of the ordinary facilities offered by any operating system within a particular platform. A benefit of this implementation is that the operation of the application and the data base management system combination becomes independent of location.

The operation of the architecture is similar to that sought via the RDA initiative: once the proposed standard has come into effect, the data exchange and transformation mechanism provided by the request and client servers can be replaced by the standards. Relational data base management systems and the Structured Query Language (SQL) are the only facilities covered by the current standards proposal.

The request server/data server principle can be extended within the client/server computing model to incorporate requests which cause the execution of remote programs. The only difference is at the destination server: rather than invoke the data base management facilities to service a particular request it would have to request the operating system to load and execute a GUIDE method.

Three request servers - the *definitions and data*, *execution* and *activities* managers - in GUIDE deal with distributed operations as illustrated in figure 34. Depending upon the nature of a request, the appropriate server is invoked. Each server is capable of identifying whether a request should be serviced locally or remotely. In addition, due to the need to manipulate data contained in ordinary files, the activity server has the capacity to access operating system specific file management facilities such as, for example, those provided as part of the Network File System (NFS) support.

[63]For example, consider the communication of data between two computing platforms one of which (call it A) employs a 32 bit representation for integers while the other (call it B) a 64 bit one. Integer values sent from machine A to B need to be converted to 64 bits before transmission: upon arrival at the destination they are in the appropriate form. Responses sent from B to A must have any integer values converted to 32 bits conversely.

[64]The provision of this function will inevitably increase the complexity of the server software.

[65]In an open systems environment the communications would be managed using *sockets* or *datagrams*.

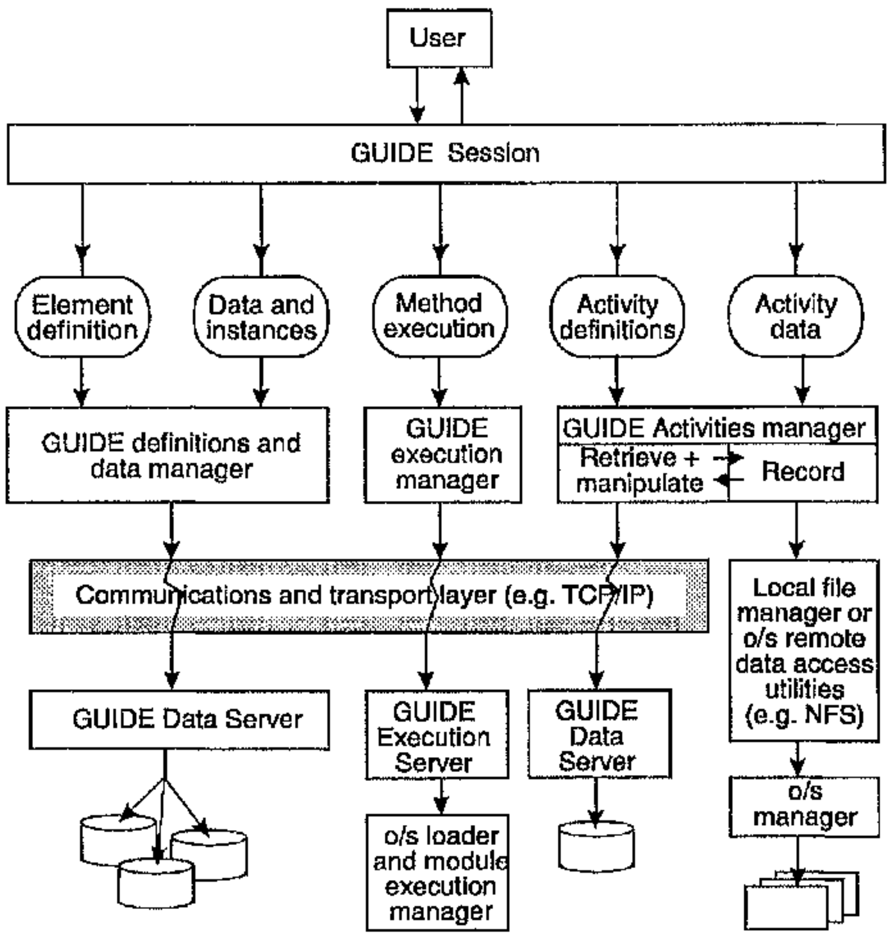


Figure 34 : Implementation of GUIDE for the support of distributed operations

The implementation of GUIDE can be sustained in either mainframe or workstation environments and takes advantage of standards facilities developed for physical computational distribution. Several of the concepts it employs currently are the result of proprietary developmental decisions. The opportunity exists to incorporate any standard techniques when these become available. The implementation of GUIDE has been based upon commonly available technology so that it can benefit from future developments in the facilities which underpin its operation.

5.2 Conceptual schema to support GUIDE

In GUIDE, a set of attributes can be employed to specify the characteristics of a particular atom; the specification of a structure which contains this atom requires a set of attributes to identify its characteristics (e.g. *name* or *family*) and an explicit relationship to link the particular atom to it. The storage of data instances requires, in addition, the provision of attribute types comparable to the atom types employed by GUIDE. Commercially available data base management systems (DBMS) capable of addressing these requirements fall into two main categories: *relational* and *object oriented.* Table 17 provides a comparison of their characteristics.

Relational DBMS's	**Object oriented DBMS's**
Best suited for typical or traditional business applications • Focus on data values, not relationships • Primarily simple structures and data types	Best suited for applications dealing with complex structures (CAD, CAM, CASE, etc.) • Focus on relationships, not data values • Primarily complex structures and user defined types
Can be enhanced to provide some *object* support • User defined data types, functions, referential integrity constraints, etc.	Generally not replacing relational data base management systems
Provide resilience and integrity safeguards required by industrial operations.	Designed to handle complex data and provide navigational access to them
Support multiple programming languages and *ad hoc* queries	Tight integration with 1 or 2 programming languages (e.g. C++)
Support multiple application domains concurrently	Focus on application specific domains
Employ a standard data access and manipulation mechanism (SQL)	Employ proprietary mechanisms for data access and manipulation, frequently based upon the facilities of specific programming languages

Table 17: Comparison of object oriented and relational DBMS's

Relational systems have a number of advantages over object oriented ones:

- they offer a standard interface to the data by means of the Structured Query Language[66] (SQL);
- they are supported by major software vendors and, as a consequence, have inspired the confidence in large engineering organisations to adopt them for product information storage;
- operations within them are underpinned by a sound mathematical theory;
- they will be supported in distributed environments under the RDA proposal.

[66]SQL92 (also known as SQL2) is the current ANSI standard. Its successor, SQL3, provides facilities which extend to encompass the capabilities offered by current object oriented systems: user-defined data types, complex structures and modifiable constraints upon them are included. The proposed standard is currently at the committee discussion level and expected to be adopted early in 1996.

Object oriented and relational systems are regarded as highly complementary. Either could be used to hold the element definitions of GUIDE. Relationship management between entities is performed inherently with the definitions in object oriented systems. In contrast, additional mechanisms have to be employed in relational systems to allow them to manage relationships: in doing so, a diverse, expandable range of relationships can be defined. Similarly, relational systems can be made to handle complex data by defining relationships across simple types which they support *ab initio*. In essence, relational systems can exhibit object oriented behaviour if they are provided with additional facilities.

An implementation of GUIDE based upon the relational data base architecture would benefit from the advantages of the architecture. The extensions required to support diverse data types can be built, initially, as part of GUIDE's control mechanism. The major advantage is the ability to communicate with a variety of vendor data base implementations and perform operations upon existing data without the need to migrate or transform them.

The employment of a standard language such as SQL brings an additional advantage: SQL queries allow the specification of constraints on values and return a list of possible results. This specification enables them to be considered as special GUIDE methods: the query constraints and results correspond to the method *inputs* and *outputs* respectively. As method inputs and outputs they can be freely combined, either with other method data or atoms within the system. SQL queries are recognised by GUIDE as special methods and they can be executed by its control mechanism.

5.2.1 Element representation

GUIDE employs a number of relational tables to represent *structures, atoms* and *methods* and their combinations. Table 18 provides a description of the use of these tables.

The tables holding the definition of elements are complemented by another set which store information pertaining to instances of elements. For example, the declaration of the generic relationship to be used in associations of structure and atom definitions is held in the *relations* table. The association of a particular structure with an atom using the generic relationship to define a peculiar construction is held, as a row, in the *relate* table. The other tables in this set are employed to hold information on committed instances of structures and atoms, instances of activities and information denoting the time of opening or suspension of specific activities. A list of this set of relational tables is given in table 19.

Table Name	Use
Structures	Used to hold the structure element descriptions including the definitions of derived structures.
Atoms	Used to hold the part of the atoms elements description including the definitions of derived atoms. The parts of the atom definition which are common to root and derived atoms are held, for data economy purposes, in another table.
Atomdef	Holds the part of the atom description which is shared between derived atoms and their root. This table complements the *atoms* table.
Methods	Used to hold the definition of all the method types used by the system. In addition, it provides support for the definition of SQL queries in association with the *queries* table.
Mdata	Used to hold the method input and output data definitions.
Units	Used to hold the definitions of systems of units. It is used in conjunction with the data in the *uconvert* table to perform conversions across systems of units.
Uconvert	Holds the definitions of the transformation functions to be employed in the conversion of values from one system of units to another.
Queries	Holds the *skeletons* for queries against data base tables. It is used in conjunction with the *methods* table to provide the specialised SQL method with the setup information which will dictate the operational characteristics during its execution.
Relations	Used to hold the definition of the possible relationships which can be applied to pairs of GUIDE elements in the formation of constructions. Table 20 provides an explanation of these.

Table 18: Tables used by GUIDE for element representation

Table name	Use
Relate	Used to hold *instances* of relationships formed between particular pairs of GUIDE elements.
Instances	Used to hold a record of all the structure and atom *instances* which have been *committed* to permanent storage. It is used for the management, storage and retrieval of particular values which refer to instances.
Register	Used to hold data relating to activity instances and to provide information on the status of the activity and the associated record of events.
Session	Used to hold statistical data relating to specific activities engaged using GUIDE.

Table 19: Tables used by GUIDE for instance information storage

Associations between structures, atoms, methods and method data are formed by employing *external* links, described in table 20. *Internal* links enable the association of GUIDE elements with data appropriate to their definition and rely upon ordinary mechanisms provided by relational data base systems - they are implemented as matched column names across different tables. For example, a particular unit is associated with an atom value by matching the value contained in the integer column of the atom table to that of the integer column of the units table.

Data base queries which involve internal links are by nature very economical: they enable the seamless joining of tables to retrieve in a single query data held in several locations. For example, the atom header information, its definition and the units applicable to its instantiated value can be retrieved in a single operation. Furthermore, internal links provide the ability to *factorise* the relational data model and, consequently, avoid unnecessary duplication. External links are suitable for the definition of links which are of a non-exclusive nature: they are more appropriate for the construction of hierarchical relationships, such as those required between structures and atoms, and are easy to create or delete without affecting the integrity of the linked elements. The use of internal links between elements of GUIDE could restrict its representational capacity: an exclusive association between a structure and a set of atoms, for example, would remove the ability to relate any of these atoms with another structure, thereby eliminating the opportunity implicitly to link the structure definitions and, consequently, the capacity adequately to represent stereotypical situations such as those encountered in design.

Relationship	Associates	With	Meaning
STR2STR	Structure	Structure	Enables the association of children structures with a parent structure.
STR2ATM	Structure	Atom	Provides the structure with its associated atoms.
STR2CST1	Structure	Constraint	Enables the specification of a constraint to validate the appropriateness of the structure at the particular stage of the activity. The method is executed before the preparation of the structure. In the case of an activity structure the method can be used to validate the appropriateness of initiating the activity in the present context.
STR2DEF	Structure	Method	Specifies a method which can supply the structure with a group of initial values after preparation and before the instance has been handed over to the user for editing. In the context of an activity structure it provides the default value of the *requester* atom to safeguard against its improper specification.
STR2MTH	Structure	Method	Provides the specification of a set of methods which might be appropriate for execution during the editing phase of the structure instantiation. In the case of an activity structure the relationship provides the association of a method to be used to confirm the initiation of the activity, register it and create the record repository.

Table continued on following page

Table continued from previous page

Relationship	Associates	With	Meaning
STR2CST2	Structure	Constraint	Enables the association with the structure of a method to validate its completeness prior to commitment to permanent storage. In the case of an activity structure it provides the mechanism to close the activity, change its status in the register and notify the requester to proceed with the appropriate checks upon it. The constraint can be built to enable the activity requester to declare it complete or reopen it for further work.
STR2MTH2	Structure	Method	Enables the association with the structure of a method to cause actions following the successful storage of the instance values. In the context of an activity structure it can be used to dispatch notification of the completion of an activity.
ATM2DEF	Atom	Method	Specifies a method to be used for supplying the particular atom with one or more initial values.
ATM2CST	Atom	Constraint	Assigns a method to validate the atom instance value.
ATM2DAT	Atom	Method data	Associates a method output with the atom thereby implicitly linking the atom with a number of possible methods to supply it with a value. Alternatively, it associates an atom with a method input thereby allowing the provision of initial values to assist the method execution.
MTH2DAT	Method	Method data	Associates input or output data with methods.
MTH2CST	Method	Constraint	Enables the validation of the appropriateness of a given method in the particular context using a constraining method upon it.
MTH2SQL	Method	Query	Provides an association between a query name (the method name) and the particular SQL statement to be executed.
DAT2DAT	Method data	Method data	Enables the association of one method's inputs to the outputs of another and the converse.

Table 20: External relationships between GUIDE elements

Figure 35 illustrates the overall data base schema employed for the storage of the GUIDE element definitions indicating the links (*external* or *internal*) across the tables. One characteristic of the schema is the use of an *element identifier* (oid) linking elements with instance information. The element identifier is an index used to provide any element of GUIDE with a unique identification tag which comprises two parts: the identifier of the computing platform where the element was defined and a time stamp of the definition with a resolution of milliseconds to guarantee its uniqueness.

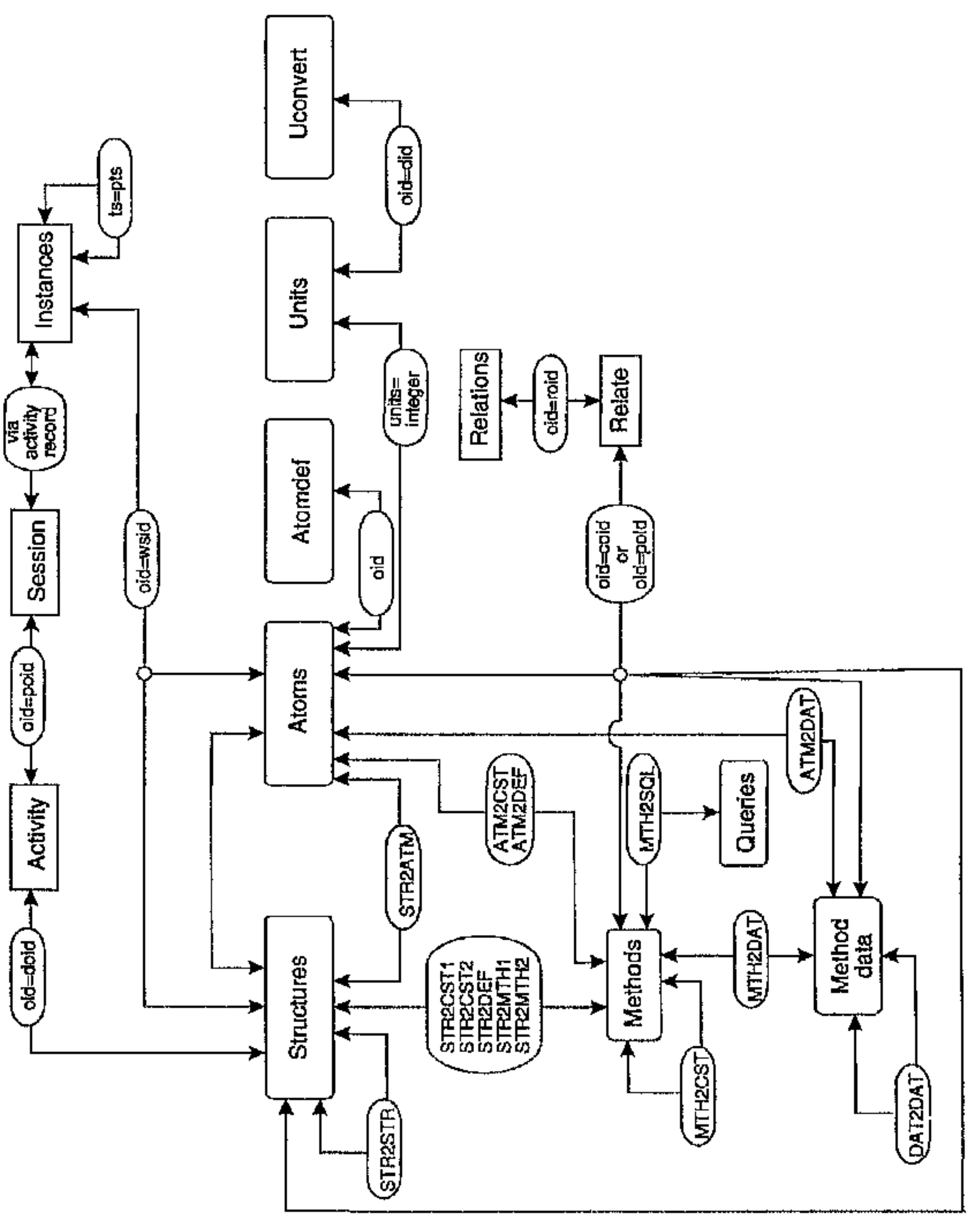


Figure 35 : Conceptual schema for GUIDE

The data base physical schema, or particular table construction, is described in tables 21 to 33.

Attribute	Type	Length in bytes	Description
oid	character	24	The structure element identifier. The first 8 bytes are the Internet address (4 decimal numbers in the range 0 to 255) of the computing platform where the definition of the structure took place, converted to 4 hexadecimal numbers in packed format. The remaining 16 bytes are: **9 to 12** Year; **13 to 15** Day of year; **16 to 20** Seconds in day; **21 to 23** Milliseconds; **24** Zero
descr	character	40	Name of the structure.
family	character	16	Family in which the structure belongs.
type	integer	4	Indicator which defines if the structure is a root or derived. The convention used is: +1 Root; -1 Derived
refstr	character	24	When the structure is of derived form, this attribute is set to equal the value of the element identifier of its root.
version	integer	4	The version number of the structure. A new version will have a different element identifier but the same name. The version number is used to collate and sort all the versions of the same structure.
creator	character	8	The individual who created the structure and, therefore, owns the definition of it. Only the creator of a particular structure is allowed to modify its definition and create a new instance of it. Furthermore, it can be used to filter the structure definitions by creator at instantiation time.

Table 21: Structures table definition

Attribute	Type	Length in bytes	Description
oid	character	24	The atom element identifier.
descr	character	40	The atom name.
type	integer	4	Indicator to denote whether the atom is a root or derived. The same convention as for structures is used.
refatm	character	24	The element identifier of the atom root if it is of derived form.
ddata	character	80	A safe default value to be applied to the atom if the search for defaults using all the other facilities has failed. This value is also used if the atom needs to be reset due to a change in the value of its parent atom instance. Integer, real and character string values are stored in this field. For arrays or pointer values the field contains a pointer to them or an element identifier respectively. The value can be used to point to the element identifier of an atom definition. GUIDE will match this identifier with the identifiers of instances either in the current or any associated activities.
version	integer	4	The atom version.
creator	character	8	The creator of the atom definition.

Table 22: Atoms table definition

Attribute	Type	Length in bytes	Description
oid	character	24	The atom element identifier.
dtype	integer	4	The atom type. Table 3 provides a description of the possible values.
dstype	integer	4	The atom subtype. This value is used in connection with systems which support selections of their own elements (e.g. a line within a CAD system) to denote the element identifier within these.
dlength	integer	4	The length, in bytes, of the atom value. If the atom is of array type, the value denotes the length per row in the array. The value has no meaning for atoms of pointer type.
varname	character	6	An identifier for the atom value.
sourcev	integer	4	The atom value source. Table 4 contains an explanation of the possible values. The mapping employed is: 1 System; 2 User; 3 Any; 4 Frequent; 5 Rare; 6 Method
dispose	character	2	The atom disposition value. The following are possible options: B Broadcast; S Stored. Section 3.2.1.1 provides an explanation of these terms.
mapping	character	254	The atom physical mapping. In its simplest form it is the location where the instance value should be stored. The location is made up by a combination of the address of the particular computer platform which contains the storage location, and a block of information which points to a file or database table where the values are to be lodged. The value could be set to point to another atom element identifier, in which case the value will be stored at the location pointed to by the mapping value of that atom. This mechanism safeguards the referential consistency of the atom value.
mapmeth	character	8	This is the name of the method to be executed to store or retrieve the atom value. It is cross referenced with the ***name*** attribute of the methods table.
changei	character	2	The atom ***action on change*** indicator. An explanation of the possible values is given in table 5. The following options can be specified: I Ignore; W Warn; A Refer; R Reject
nval	integer	4	Denotes the atom value multiplicity. Used for atoms of array type it denotes the number of values in the array.
units	integer	4	An indicator for the atom value units. It is cross referenced with the ***id*** attribute of the units table.

Table 23: Atomdef table definition

Attribute	Type	Length in bytes	Description
oid	character	24	Method element identifier.
descr	character	40	Method description.
class	integer	4	Used to denote whether the method will need to interact with the user of GUIDE or it will be executed in the background. Passive constraints which have **only one** input are frequently non-interactive: this would also cover the case of SQL queries which are executed without any conditions upon them. The variable is also used to distinguish between normal methods and database queries. The options for the value are: +1 Background method -1 Interactive method +2 Unconditional SQL query -2 Conditional SQL query
type	integer	4	Used to denote the state of the code of the method to be executed. Methods can be developed using any high level programming language; these require compilation before execution: at execution time GUIDE will load them into memory, resolve all their external references and run them. Alternatively, the methods can be written in special languages which do not need compilation but require the services of an interpreter; GUIDE can invoke the interpreters to execute these methods. Finally, GUIDE can also invoke operating system specific facilities to execute particular methods written in a variety of proprietary languages (e.g. bsh, ksh, csh, rexx, dcl, etc.). The possible options are: 1 Compiled object code. 2 Interpreted by application. 3 Interpreted by operating system.
name	character	8	The method name. It is cross referenced with the ***mapmeth*** and ***convmeth*** attributes of the ***atomdef*** and ***uconvert*** tables respectively. Frequently it corresponds to the name given to the file holding the method code within the operating system area.
location	character	254	Indicates where on the system the method code is located. The first 8 bytes are the Internet address of the computing platform where the code is stored. The rest of the value is used to point to the appropriate file system within that machine. The GUIDE method execution server compares the method location machine identifier to its own and executes it or directs the request to the execution server of the remote machine.
version	integer	4	The method definition version.
creator	character	8	The method definition creator.

Table 24: Methods table definition

Attribute	Type	Length in bytes	Description
oid	character	24	The method data element identifier.
prompt	character	32	The string which will be used to prompt the user of GUIDE for the required value.
iotype	integer	4	Used to denote if the value is going to be used for input or output. The possible values are: **0** Method output. **1** Method input.
vtype	integer	4	Used to denote the type of the value. The same options as for the attribute***dtype*** of the atomdef table apply except that the method input/outputs cannot be pointers. The effective options are: 1, 2, 3, 4, 5, 6, 13, 14 and 15.
varray	integer	4	Number of elements if the value is of array type.
vlength	integer	4	Length in bytes of the value or per element of the array.
name	character	6	A short name for the method value.
mapping	character	254	This is used in connection with the process of establishing a default value for the method input/output. It can point to the element identifier of an atom within GUIDE, a particular storage location, or an instance of a method value within a given activity record.
mapmeth	character	8	The method to retrieve the default value for the particular method input. It is cross referenced to the ***name*** attribute of the methods table. In connection with method outputs it can be used to provide a default storage location for any instances generated if no mapping to atoms can be found for them using the routine mechanisms of GUIDE.
version	integer	4	The method input/output definition version.
creator	character	8	The method input/output definition creator.

Table 25: Mdata table definition

Attribute	Type	Length in bytes	Description
oid	character	24	The units element identifier.
id	integer	4	A secondary identifier used to link the particular unit to an atom definition.
desk	character	80	The units descriptor (e.g. m/sec^2)
version	integer	4	The unit definition version.
creator	character	8	The unit definition creator.

Table 26: Units table definition

Attribute	Type	Length in bytes	Description
oid	character	24	The units conversion identifier.
sid	character	24	The source units identifier.
did	character	24	The destination units identifier.
convmeth	character	8	The method to be executed to perform the conversion from the source system of units to the destination one. It is cross referenced with the ***name*** attribute of the methods table.
version	integer	4	The unit conversion definition version.
creator	character	8	The creator of the conversion definition.

Table 27: Uconvert table definition

Attribute	Type	Length in bytes	Description
oid	character	24	The SQL query element identifier.
nodeid	character	8	The identifier of the node where the database query should be directed.
database	character	8	The name of the database to connect to in order to execute the query.
refobject	character	18	The name of the particular object within the ***database*** to be queried. SQL queries which span multiple tables should be executed through a single view definition which spans one or more tables as appropriate. Consequently the value of this attribute can be either a table or a view name.
qrydata	character	1920	The SQL statement to be executed. It can either be a query or another operation (e.g. insert, update, etc.)
version	integer	4	The query definition version.
creator	character	8	The query definition creator.

Table 28: Queries table definition.

Attribute	Type	Length in bytes	Description
oid	character	24	The relation element identifier.
name	character	8	The relation name. Table 20 provides an explanation of the currently implemented values.
descr	character	40	A short description of the relation.
type	integer	4	The relation type. It refers to the type of links enabled by a given relation. The possible options are: **1** One to one. 2 One to many. 3 Many to one. 4 Many to many.
version	integer	4	The relation definition version.
creator	character	8	The relation definition creator.

Table 29: Relations table definition

A management facility is provided in GUIDE which supports element:

creation,
association[67],
analysis,
modification and
abstraction from specific instances.

Foils 1 to 5 in Appendix 1 provide illustrations of the management of GUIDE elements. The methodology employed requires that element definition precedes the creation of any associations between them. For example, to create an atom definition, the units, conversion and method definitions must exist. The elements may be linked together once the atom definition is complete. Structures are built similarly and new element associations can be defined at any time. The creation of a new association with an element implies, in GUIDE, its modification which creates a new version. If the element is used in the construction of higher level elements, its parents are also versioned appropriately to reflect the change in their definition. The versioning of elements is the basis of representational integrity in GUIDE: old versions of elements are kept due to the dependency of instances on them.

Associations of elements, created using the relationships provided in the *relations* table, are stored as instances in the *relate* table. For example, the association of an atom with two structures is represented by two instances in the relate table. The relate table structure is described in table 30.

Attribute	**Type**	**Length in bytes**	**Description**
poid	character	24	The element identifier of the first item in the association. For example, it could be the identifier of a structure.
coid	character	24	The element identifier of the second item in the associated pair. For example, it could be the identifier of a particular atom.
roid	character	24	The relation element identifier. This is cross referenced with the *oid* attribute of the relations table.
creator	character	8	The individual responsible for the creation of the specific association.

Table 30: Relate table definition

[67]Using the relationships detailed in table 20.

5.2.2 Storage of instances

The editing phase of atoms and structures is finalised by the *commitment* of the instances to permanent storage. The storage location and the method to be used to store and retrieve the instances are provided by the values of the *mapping* and *mapmeth* attributes of the *atomdef* table. Information on instance commitment is recorded in two places: an event is inserted in the design record and an index to the particular value is inserted in the *instances* table. The construction of the instances relational table is described in table 32.

Attribute	Type	Length in bytes	Description
ts	character	16	This is, usually, a time stamp taken at the time of commitment of the particular instance. If the instance data was stored in a relational database table which has a unique, indexed column, then the *ts* value is made to correspond to the value of the *key* for the particular row which will be used to store the instance value. Figure 36 depicts this concept.
oid	character	24	The element identifier of the stored instance.
wsid	character	24	The instance identifier created when the element was prepared. It is taken from the *header* of the particular instance.
pts	character	16	The parent instance of the current one. During atom storage this field contains the time stamp value of the committed structure instance which references the atom. For structure instances, it provides a pointer to the commitment of their parent structure. The index is used for evaluating dependencies between instances.

Table 31: Instances table definition

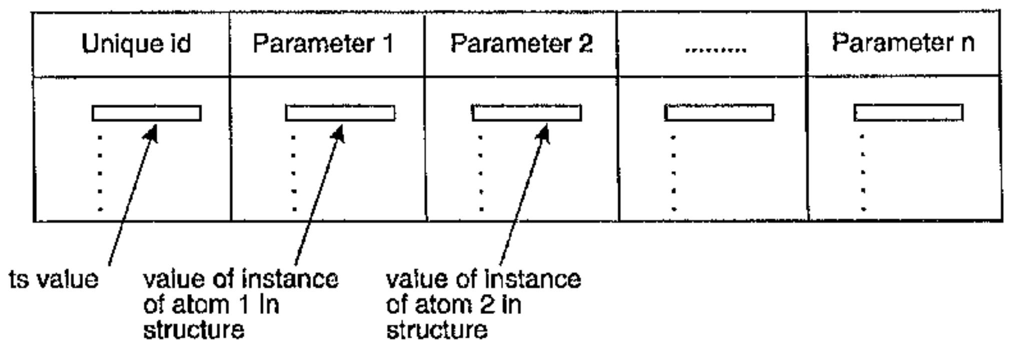

Unique id
Parameter 1
Parameter 2
....
Parameter n
ts value
value of instance of atom 1 in structure
value of instance of atom 2 in structure

Figure 36 : Use made of unique key in data base table for the storage of instances

GUIDE can use consolidated or fragmented relational data base tables to store instances. Figures 37 and 38 provide two illustrations. The structure instance to be stored represents a geometric line. Two alternative representations are presented with the corresponding data base table constructions to store the line instances. The former represents the line in terms of a starting point and a direction vector. The latter represents the line in terms of its end points, represented as children structures. In the latter case, a row of values will be stored for each end point of the line. The *pts* field of the instances table for each of these instances will contain the time stamp value of the line structure instance.

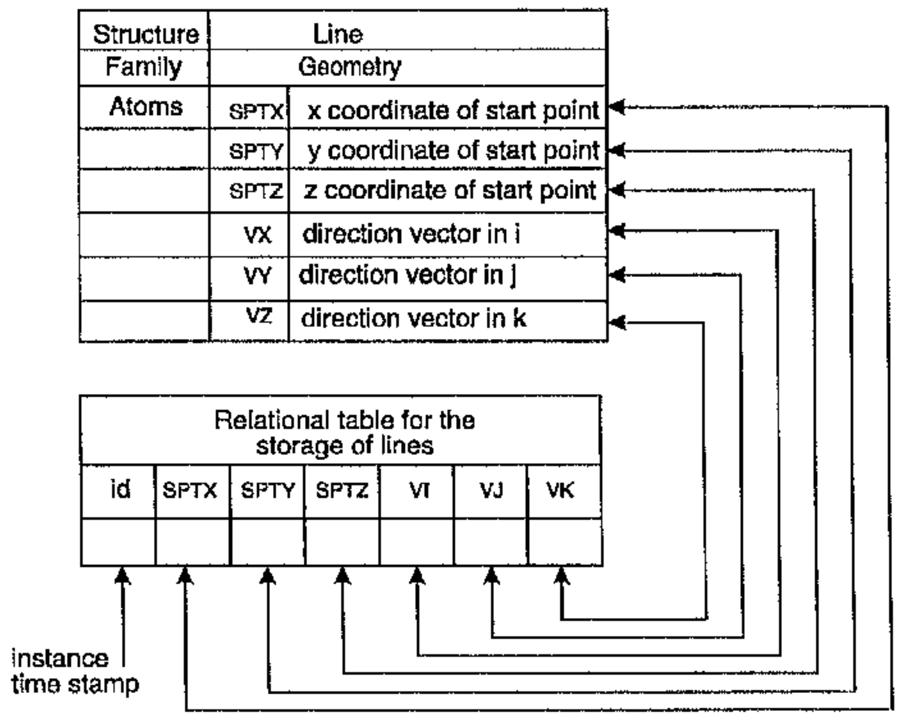


Figure 37 : Use of a single table for the storage of a structure's instances

The design record is linked with the instances table entries by cross referencing the *oid* and *wsid* attributes of the *instances* table with fields in the design record file.

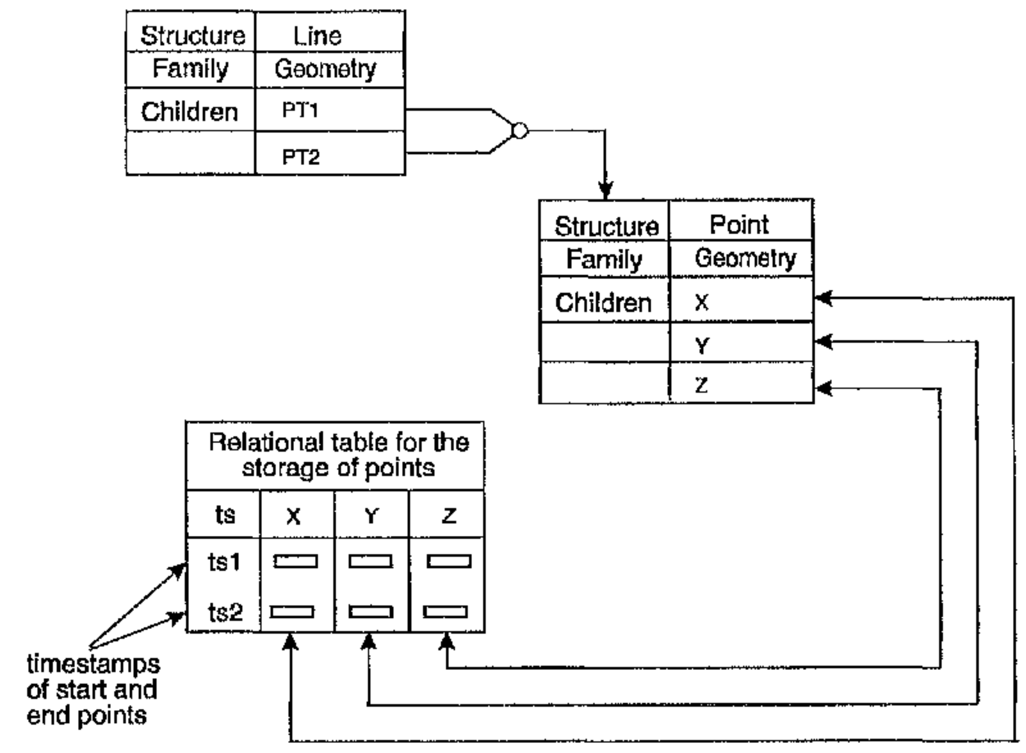


Figure 38 : Use of multiple tables for the storage of a structure's instances

5.2.3 Storage of activities

The storage of activities in GUIDE is based on two data base tables - the *register* and *session* - and an operating system file. The register table stores information about activity structure instances; the session table stores information of the work done on activities over time. Activity initiation creates an entry in the register table; the entry gets updated every time the activity status is changed. Entries are made in the session table at activity opening, suspension and resumption. Details of the particular construction of the register and session tables are given in tables 32 and 33.

Attribute	Type	Length in bytes	Description
poid	character	24	The activity time stamp for which session data are kept. It is cross referenced by the *oid* attribute of the register table.
sesstart	character	16	The time of opening, by an individual of a specific activity.
sesend	character	16	The time of suspension of a particular activity.

Table 32: Session table definition

Attribute	Type	Length in bytes	Description
oid	character	24	This is the time stamp of the activity structure instance.
doid	character	24	Element identifier of the activity structure definition. It is a pointer to the *oid* field of the structures definition table.
poid	character	24	The time stamp of the activity instance parent activity instance. This is used to find the common parent of a number of activities and thereby allow the linkage of elements across them.
status	character	12	The current activity status. Table 12 provides details of the possible values for this attribute.
request	character	8	The individual who has requested the activity. The value is used during the activity completion stage to validate that the user attempting it is the appropriate one.
actor	character	8	The activity actor. The value is used during activity opening and closure to validate the authority and suitability of the individual attempting the operation.
descr	character	40	This provides a short description for the particular activity. It is the same as the *descr* attribute of the structures table for the particular activity.
recnode	character	8	The Internet identifier in packed hexadecimal of the particular computer node where the record of the design is stored.
recname	character	8	The name of the design record file.

Table 33: Register table definition

The *recnode* attribute of the register table associates activities with operating system files which store the design record. A design record configuration file, described in table 34, provides the links.

Field	Length in bytes	Description
recnode	8	The node where the design record file is held given as an Internet address in packed hexadecimal.
filesystem	254	The name of the file system or *path* of the directory where the design record files are held. Multiple locations can be specified for a particular node by means of additional entries of the *recnode* and *filesystem* combination.
localname	254	The design record file directory or file system may be located in a remote node. Under particular circumstances and using operating system dependent utilities (e.g. NFS) this file system may be remotely mounted and become accessible as a *local* file system; in this case the attribute is used to indicate the local name of the remote file system. In locations which possess *automounter* support this entry could be used to instruct the operating system to mount the remote file system and make it available for use at the point of an individual *opening* a specific activity; the same facility can be used to unmount the file system at activity suspension.
comment	80	A short string to provide additional information about a design record file.

Table 34: Structure of the design record configuration file

One design record configuration file is used for every node where GUIDE is engaged. Figure 39 provides an illustration of the links between the different tables and files used.

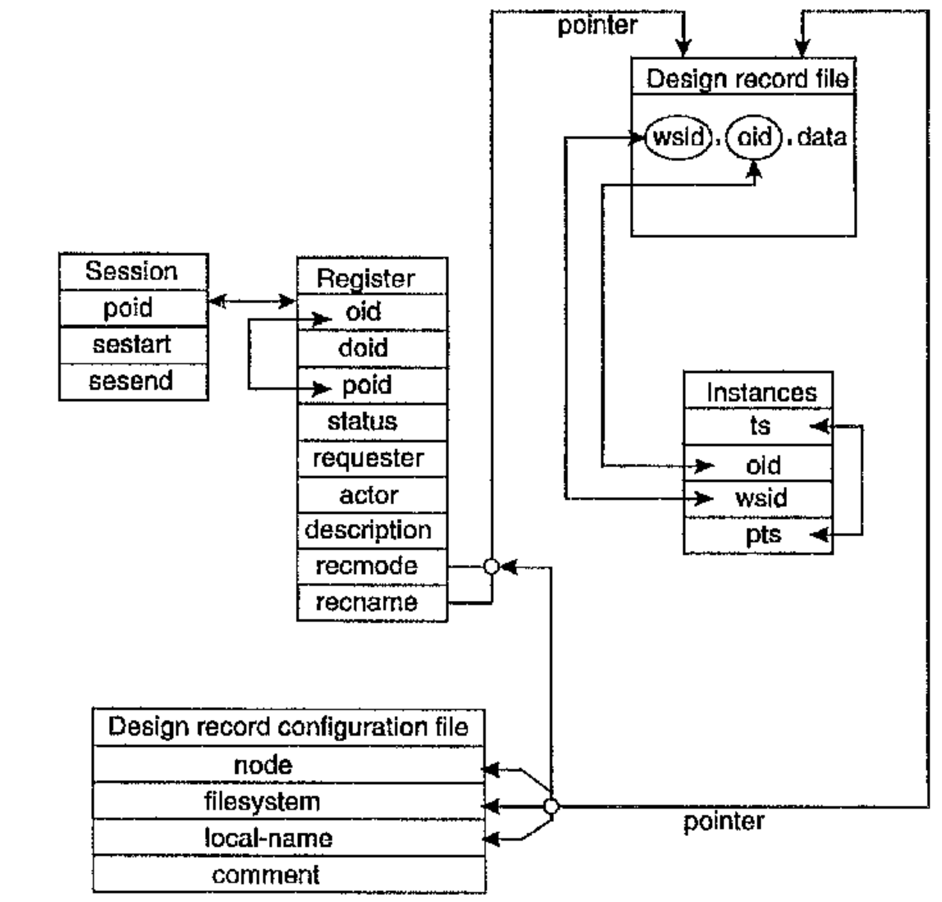


Figure 39 : Associations of activities with stored instances

GUIDE follows the links to retrieve instances by:

- using the activity *oid* to retrieve the appropriate *recnode*;
- using the *recnode* to retrieve the design record file name;
- examining the design record file to locate the particular *wsid* for elements with the required instance *oid*;
- retrieving the relevant *ts* values for particular *wsid* and *oid* combinations;
- using the *oid* value to obtain information on the *mapping* and *mapmeth* values from the element definition table;

- using the *ts*, *mapping* and *mapmeth* combination to retrieve the instance from the appropriate storage location.

A similar sequence is used to identify the activities and, subsequently, the creators of instances held in particular storage locations. The former operation is used in connection with the retracing and abstraction of the activity record. The latter is used in the modification of values which depend upon instances in activities other than the one engaged by the designer and to alert the responsible individuals of links upon particular instances.

5.2.4 Data type extensions

GUIDE atom instances can be of *array* or *matrix* type which are not supported by relational database systems at present. Extended data types[68] are supported using a combination of two relational tables managed by GUIDE's control mechanism. One table is the *long field register* and holds information about the type (e.g. integer, real, character), length (in bytes per row), construction (i.e. array or matrix), cardinality (i.e. rows first or columns first), and numbers of rows and columns in the array or matrix. The other table is the *repository* and stores the data. The repository is linked to the register table and has rows of the maximum[69] length the data base manager will allow. GUIDE packs the rows and columns of data and inserts them into one or more of these long rows. The same mechanism is employed for their retrieval.

5.3 Integration with company data

GUIDE structures and atoms can be used to represent descriptions of existing data in relational data base tables. Atoms can represent columns and structures can represent *tables*, or *views*. Linked structures can represent complete data base schemata. Software used or developed by engineering companies to store and retrieve data can be represented as GUIDE methods and associated with atoms: particular requests to instantiate atoms would cause the execution of the linked methods and the consequent storage or retrieval of values.

[68]Sometimes referred to as: *long fields*.

[69]In most relational data base systems the maximum row length described as a *long varchar* field is 32 kilobytes or 32768 bytes. Some systems impose a restriction of 2048 bytes per row. GUIDE's control mechanism senses any restrictions and adjusts the storage and retrieval of values as necessary.

Benefits stemming from GUIDE's capacity to represent descriptions of existing data are:

- structures and atoms can be used to *overlay* data bases, tables and their attributes and, subsequently, impose constraints on the data;
- existing data can be used to provide default values to or constraints on atoms during instantiation;
- past product descriptions can be used in the generation of new data with links and dependencies formed and managed by GUIDE;
- the company's data are integrated organically with GUIDE elements and this safeguards their integrity in the presence of change;
- the record of data access provided can be used to compile reference lists to the data and, subsequently, be used either for the elimination of duplicates or unreferenced data;

5.4 Summary

GUIDE is a modular software environment which employs server processes to enable its operation in distributed computational environments. GUIDE uses standard facilities to access data and execute methods at locations other than the one where it is being exercised.

GUIDE employs standard relational data base systems to store representations of the stereotypes it employs and, depending upon user specification, the instances generated. Database access is considered a specialised method and is managed by GUIDE's control mechanism. Instances can be stored on a variety of media using GUIDE methods. GUIDE can be used as a data manager and can engage company developed data storage and retrieval software. Existing product data used through GUIDE are enriched by the addition of links and dependencies to other data and knowledge representations. Activity information is registered in data base tables and the design record is stored in operating system files.

GUIDE provides the ability to manage the data descriptions and diverse data forms which exist within an engineering enterprise using a single application. This benefits the conduct of operations within the company and safeguards the integrity of the product data which is generated.

GUIDE In Practice

6.1 Use of GUIDE in design

Three examples given here illustrate GUIDE supporting designers.

In the first example, the design of a housing for the front crankshaft oil seal in an engine illustrates GUIDE's capacity to: aid the designer's use of a solid modeller to create and manipulate component representations; enable the retrieval of data from other designs and data bases and incorporate them in the housing design as constraints or to set design parameters; create several compatible entities as the result of a single decision by the designer.

The second example concerns the management of engineering change affecting a computer monitor. It illustrates GUIDE's capacity to: represent a highly structured (sequential) methodology and, by means of its control mechanisms, allow the method to be conducted concurrently[70]; use existing product descriptions to control the management processes; support bi-directional communication of design decisions within the company and to external contractors. The example is taken from work conducted in association with IBM's design and manufacturing facility at Greenock, UK, to develop a methodology for the communication of product information within an Extended Enterprise using PDES and STEP. Interim results of this work are reported in **[Tsiotsias et al, 1994]**.

The third illustration is of the application of GUIDE to create a prototype design system supporting a technological development. The focus was the design of ceramic reinforced ceramic composite materials (SiC-SiC) for aerospace propulsion applications and was taken from work conducted with Rolls Royce Ltd. between October 1990 and September 1993. The results of this work are reported in **[Hopper et al, 1993]**.

[70]The representations used in GUIDE and the way its control mechanism manages them allow the remodelling of the Engineering Change procedure from a strict sequence to one which is suitable for concurrent operations.

6.2 Design of a crankshaft oil seal housing

The housing design is constrained, primarily[71] by the physical characteristics of the part it encloses - the dimensions, material, operating temperature - the loads exerted at its interfaces with conterminous parts and the method to attach it to a rigid support. Some design constraints are:

- the housing hole diameter is supplied by the external oil seal diameter adjusted to give a press fit between the two parts;
- the housing thickness is equal to or greater than the oil seal thickness;
- the number and dimensions of the fasteners needed to attach the housing to the engine block determines the housing's overall shape and the dimensions of the holes in it;
- the engine block material may constrain the housing material type or dictate the use of a gasket at the joint between them if an unresolvable incompatibility of materials arises.

The finalised design of the oil seal housing with its associated components is shown in plate 10.

Aggregations of geometric shapes, represented in solid modellers as features, can be used to create geometric representations of engineering parts. The part shape is manipulated through the specification of particular values to a predetermined set of parameters. Values can be specified explicitly or as the result of a calculation based upon the value of other parameters; constraints can be imposed upon the parameters to limit the acceptable values to a specific range. The use of *parametric modellers*[72] for design relies upon the designer's capacity to:

- interpret the significance of the values required;

- audit the engineering validity of the solution;

- establish, record and manage the interdependencies of parts and control them in the presence of change.

[71]To preserve clarity only a subset of the possible design constraints is considered in this example. Additional constraints could easily be in force: for example, the housing dimensions could be restricted to a range of possible values taken from a catalogue of standard parts supplied to the company.

[72]The term is used to describe solid modellers which have the capacity to manipulate complex shapes via the specification of a set of predefined parameters.

In its application, GUIDE manages company data and knowledge embedded in software facilities and complements the use by the designer of the parametric modeller. Specifically, GUIDE is used to:

- create a formal representation of a generic housing and its design parameters;
- define methods investing the design parameters with values;
- create relationships between parts by enabling the extraction of values from existing part descriptions (e.g. the oil seal diameter) and using them to set the value of appropriate housing parameters;
- impose constraints upon the parameter values;
- create, by invoking the facilities of a parametric modeller, the geometric representation of the part;
- compile an active record of the design activity as an integral part of the product description or for the abstraction of a procedure for the design of similar products in the future.

6.2.1 Representations used to support the design

The design of the oil seal housing depends upon representations of the seal, housing and fastener parts. The housing stereotype includes, as associated children structures, representations of the holes to accommodate the oil seal and for fixing to the engine block. These and related stereotypes are defined in figure 40.

The CATIA[73] solid modeller provides geometric representations of the oil seal, housing and fastener. The housing shape parameters are illustrated in plate 6. The geometry is stored in proprietary CATIA files and facilities are offered by the solid modeller application programming interface (API) to:

- retrieve the parameter definitions for a shape;
- obtain the default values associated with a given parameter;
- set or modify the value of a parameter instance;
- create the solid geometry corresponding to a given set of parameter values.

[73]CATIA is a commercial computer aided design system and offers facilities for parametric shape definition. It is a registered trademark of Dassault Systemes, the CATIA developers.

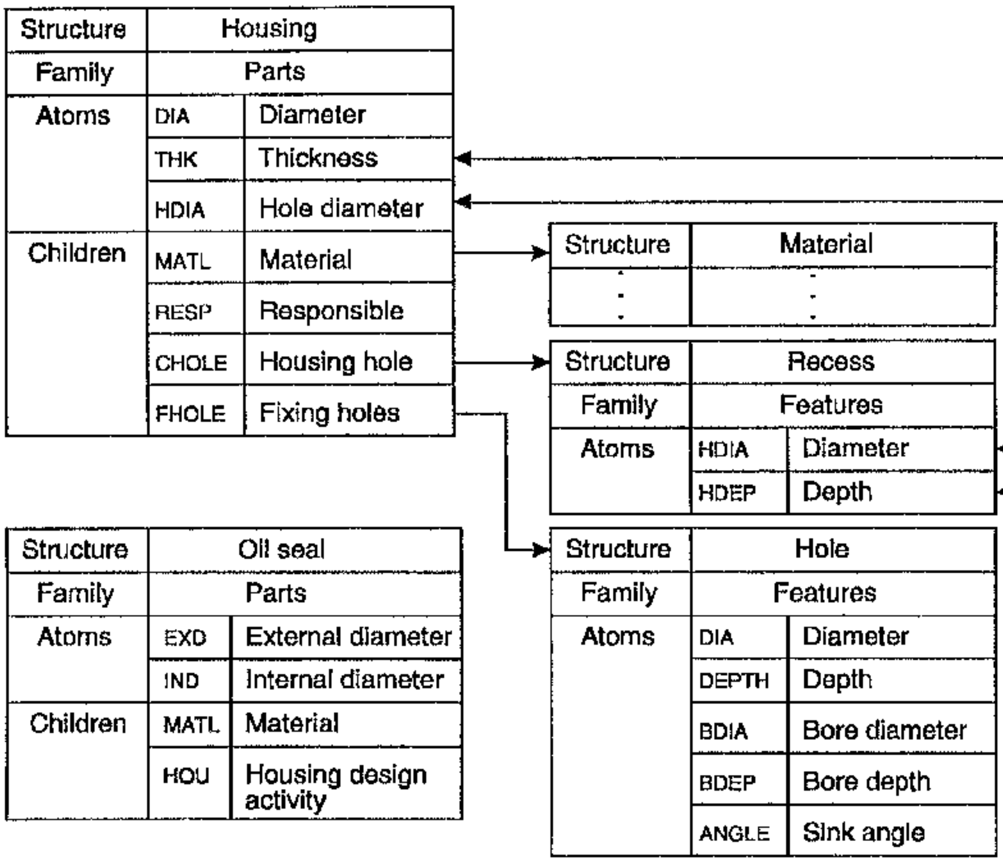


Figure 40 : Structures used in the housing design example

The geometric representations are linked to their corresponding part stereotypes (structures and atoms) using GUIDE methods; figure 41 illustrates these connections.

GUIDE methods supporting the design[74] provide the following functions:

get_definition Used to retrieve the parametric geometry definitions from the CATIA data base. The method is associated with the part structures as a pre-instantiation constraint. If the parametric definition cannot be retrieved, the instantiation of the part structure is inhibited thus guaranteeing the association of the CATIA geometry with the corresponding GUIDE structures.

[74]The methods can be written by the designer or by programmers supporting the company's engineering activities. The methods call API routines of the parametric modeller, are external to GUIDE and are associated with GUIDE structures and atoms. During instantiation, GUIDE presents them to the designer as value supplying options.

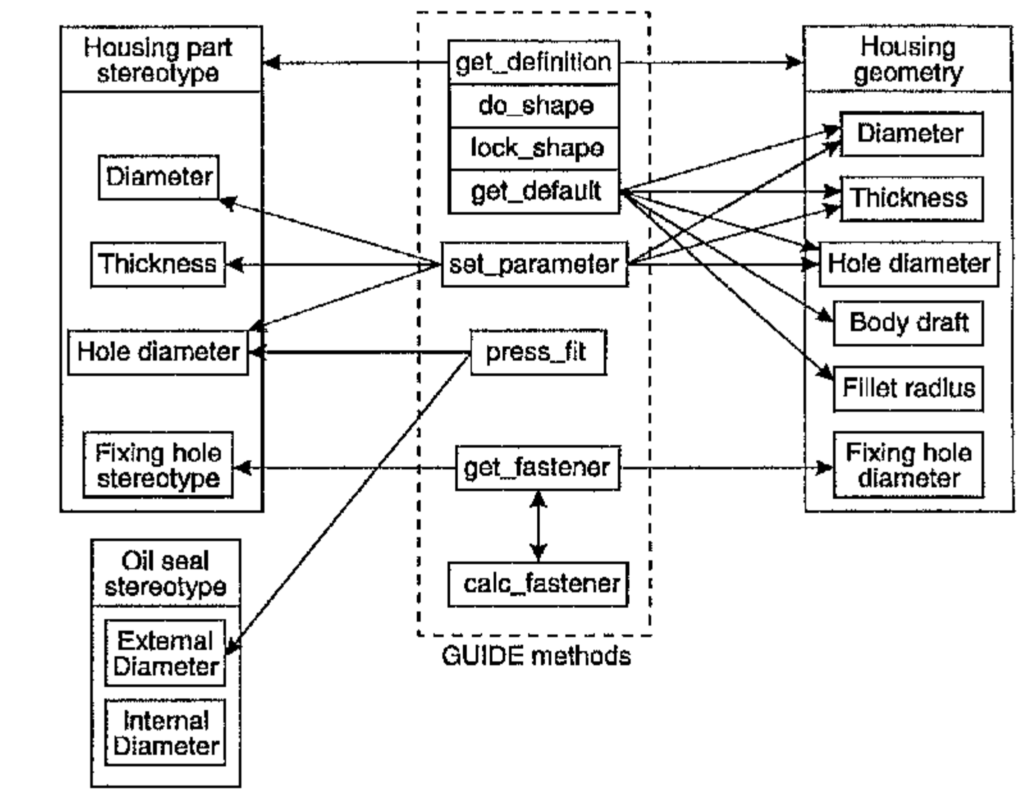


Figure 41 : Correspondence of the GUIDE housing part structure with the CATIA shape definition parameters

get_defaults Used to compile a list of parameters associated with the geometric definitions, match the parameters to their corresponding part-structure atoms, retrieve the parameter default values and set the atom instance values. The method is associated with the part structures to supply multiple atom instance default values after their preparation. Subsequent to the setting of the default parameter values the method invokes the **do_shape** method to create the initial geometry.

do_shape Used to ascertain that all of the geometric parameters have been specified and subsequently to modify the part shape according to these values. It is associated with the part structures as an edit method and can be invoked any number of times during instantiation of part structures.

set_parameter	Used to validate the atom instance values proposed by the designer. The method checks for any geometry constraints imposed by the solid modeller; subsequently it validates the chosen value against any additional GUIDE constraints. If all checks are successful, the geometric parameter and atom instance values are set and the **do_shape** method is invoked to modify the part geometry. This method is associated with every atom of the part structures, as a constraint.
lock_shape	This method is invoked after the part structure instance has been committed. It prevents any modification of the geometry taking place externally to GUIDE and thereby maintains the part geometry congruently with the part definition.
get_fastener	The method is used to retrieve a valid set of fastener dimensions from a company database. The fastener dimensions are stored in a relational table and the method (which is an SQL query) is used to set, in a single operation, all atoms associated with the fastener structure.
press_fit	Used to calculate the diameter of the hole, given the shaft diameter, to produce a press fit between them. The method is associated with the atom representing the housing hole diameter.
calc_fastener	Used to calculate the minimum fastener diameter required to withstand a given load.

6.2.2 Housing design activity

The oil seal part stereotype has associated with it, as a child structure, the design activity of the component to house it. The housing design activity is initiated by the request to instantiate the child structure prior to completing[75] the oil seal design.

[75]It is assumed throughout this example that the oil seal design is complete and, therefore, the seal's parameters have been set and validated.

The housing design progresses as follows:

1. Start Housing Design

   The responsible designer opens the housing design activity, reads the specification (reason), retrieves the oil seal structure description from the parent activity (shown in plate 7) and initiates the housing preparation.

2. Create Outline Housing

   From a list of GUIDE structures available for instantiation, the designer selects a housing which can accommodate six[76] holes: the pre-preparation constraint locates the CATIA parametric shape definition and loads it into memory; the group defaults method retrieves the housing default geometric parameters and the structure prepared is presented to the designer, as shown in plate 8, for further manipulation.

3. Modify Housing

   The designer decides to adjust the housing hole dimensions to suit the oil seal diameter. From a list of available methods to provide a suitable value, **press_fit** is selected. This method requires as an input the shaft diameter. The designer may decide to obtain this from a list of atom instances in the oil seal design activity which includes the oil seal external diameter. GUIDE retrieves the instance value, executes the **press_fit** method, validates the output of the method, sets the housing hole diameter to the calculated value and records the dependency of the new instance in housing and seal design activities.

4. Create Fixing Holes

   The designer requests the preparation of the child structure corresponding to a fixing hole in the housing. Following the structure preparation, the hole diameter requires modification: the ability to set the hole diameter to accommodate the dimensions of the fastener is provided by the **get_fastener** method. One of the method inputs (equivalent to a database query search constraint) refers to the fastener diameter. From a list of methods available in

[76]To maintain simplicity in the example, it has been assumed that six metric socket head cap screws, arranged on a hexagonal pattern, will be used to secure the housing onto the engine block.

GUIDE the designer selects the **calc_fastener** method to calculate the appropriate fastener diameter and, hence, to constrain the database search. The results of the search are presented and the designer decides to assign all the fastener data returned (shown in plate 9). GUIDE sets the value of the hole structure atoms appropriately and offers the option to create the structure whose atoms correspond to any unassigned values. Acceptance by the designer causes the instantiation of a fastener structure. The remaining five fixing holes are created using instance values taken from the first hole structure.

5. Complete Housing Design

The designer commits the finalised housing structure (shown in plate 10) and, using the management facilities of GUIDE, completes the housing design activity, causing a notification to be sent to the requesting authority. The requester checks the housing design by examining the GUIDE activity record (shown in plate 5), accepts the solution provided and completes the oil seal design activity.

6.3 The management of a computer display monitor design change

Engineering operations within International Business Machines (IBM) are conducted in accordance with company-wide documented procedures[77] defining the role of the company's engineering teams and external contractors in product development. Changes in IBM's business practice are transforming the product development process into a distributed activity in which responsibility for engineering functions is shared with the company's vendors[78]. This change in product development methodology has significant implications for the composition of the company's product model and the procedures used to manage product development. For example, the company's contractors must be able to initiate and manage engineering changes to products. A product model which contains knowledge of the development procedure (the product description) as well as data is required to support this operation and secure product integrity.

GUIDE's capacity to create and manage representations of procedures and product descriptions provides the opportunity to establish a prototype product development facility appropriate to an extended enterprise and to exercise it for engineering operations upon company's products. The example described involves the management of an engineering change made to IBM's 9507 flat screen display monitor, illustrated in plate 11. GUIDE :

- creates the monitor product description;
- links the product description to existing company data;
- employs representations of company procedures appropriate to the management of engineering change.

6.3.1 A description of the engineering change procedure in IBM

Engineering change management within IBM is described in several documents and software packages (e.g. REALM, ECMS, GTS) developed to aid engineers to conduct the company procedure. A request for an engineering change raised as a result of a problem reported by one of IBM's manufacturing contractors is managed as illustrated in figure 42.

---

[77]**Engineering practices** is the term used within IBM to describe them.

[78]The product development is, essentially, conducted by an *extended enterprise*.

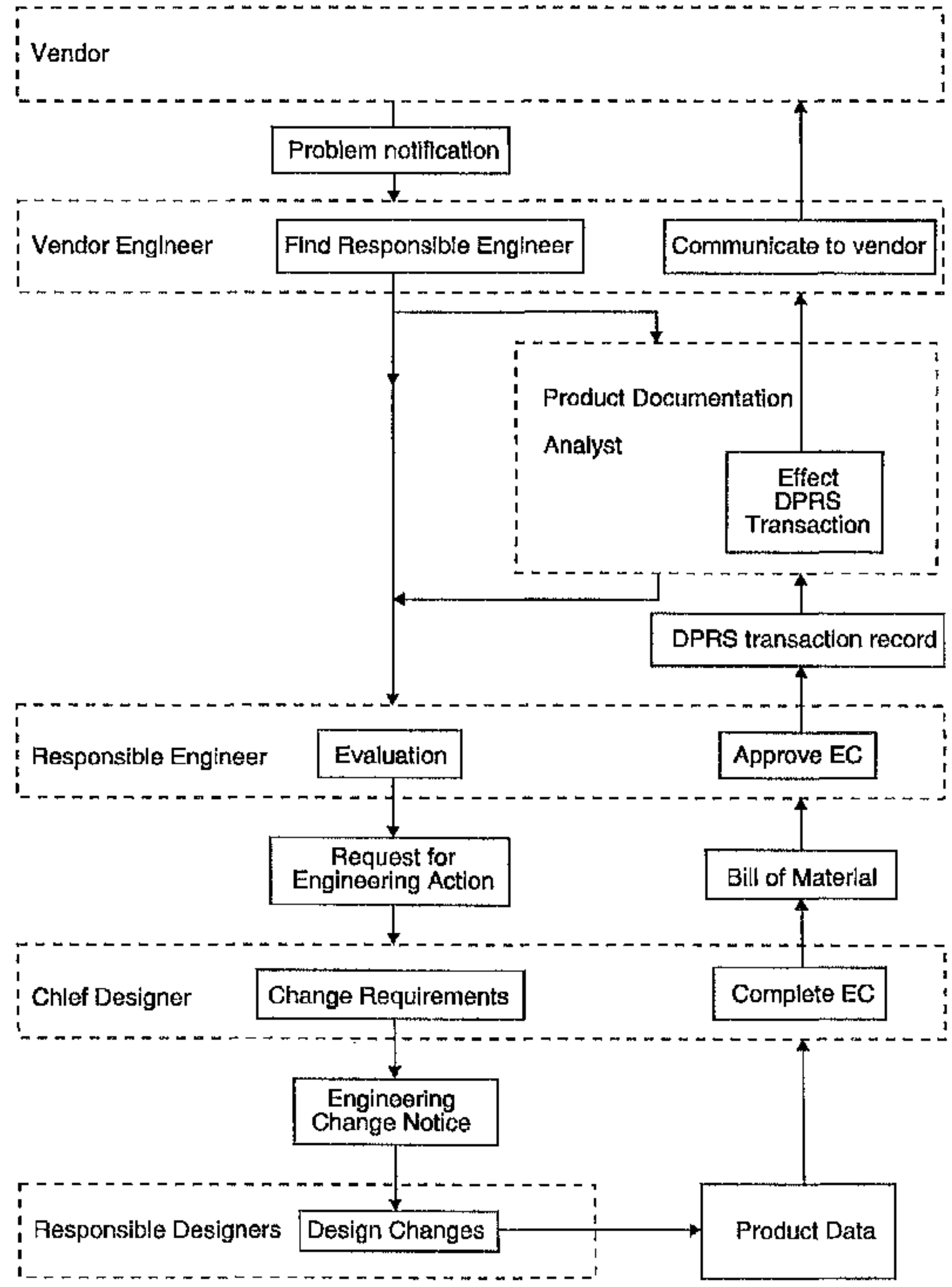


Figure 42 : Flow of Engineering Change procedure in IBM

The conduct of the engineering change involves the following actions:

1. A vendor contracted to manufacture a part in a given product and encountering a problem sends notification to the IBM buyer or Vendor Engineer (VE).

2. The IBM vendor engineer tracks down the product Responsible Engineer (RE) and passes on the problem information as received from the vendor. If the product is developed at a different company location (*plant of control*) the vendor engineer passes the problem information to that location's Product Documentation Analyst (PDA) to communicate it to the appropriate responsible engineer.

3. The responsible engineer performs a 'sanity' check: this helps to establish if a change is appropriate and whether a solution can be effected by a change of manufacturing procedure or by redesign of the part. If a design change is required, the problem notification and the responsible engineer's comments are forwarded to the product Chief Designer (CD) as a Request for Engineering Action (REA).

4. The chief designer determines the required corrective actions, creates an Engineering Change (EC) package and tasks individual designers with particular change responsibilities, using the Engineering Change Management System (ECMS).

5. The designers perform the required changes and log notifications on ECMS which are made visible to the chief designer.

6. The chief designer establishes the effect of the changes on the product: functional[79] changes require the issue of the modified part with a new part number; non-functional changes require the issue of a new engineering change number. The new part and engineering change numbers are used to define the new product structure and bill of material; completion of the engineering change is logged on ECMS.

7. The ECMS notification triggers the responsible engineer to approve the engineering change and, via the PDA, to create the Development/Production

[79]A change in specification, form, fit or function qualifies as a functional change; a new part number is issued to avoid confusion during production.

Record System (DPRS) transaction which will propagate the changed data to the rest of the company's sites.

8. The Product Design Analyst effects the DPRS transaction and this notifies the vendor engineer.

9. The vendor engineer dispatches the product release data to the vendor.

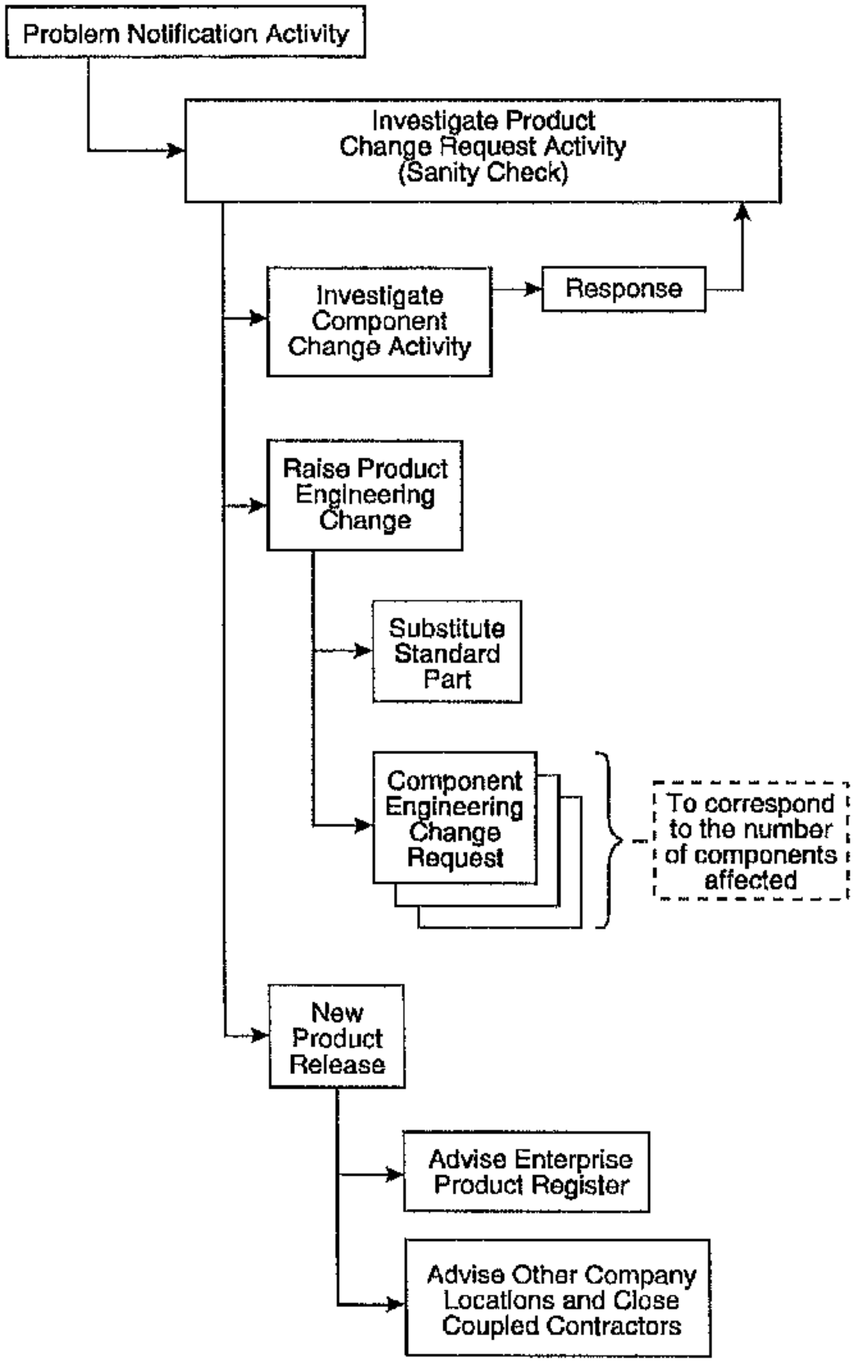
IBM's product model in its current form is appropriate for the communication of data within the corporation. It depends upon specific software tools and procedures implemented within the tools. Electronic data interchange between IBM and its subcontractors is constrained by the software capabilities and the product model is assumed to be fixed in structure and content. A dynamic product model based upon evolving and emerging product descriptions is more appropriate to an organisation consisting of multiple interconnected agents but requires additional mechanisms to manage and communicate it. GUIDE can provide:

- a record of the product development;
- access to existing sources of data which may be in different formats during product design and attachment of the data to product descriptions;
- linked implementations of engineering and business procedures;
- data translation and communication methods (e.g. based on the PDES/STEP standard) internal to the stereotypes describing products.

Ultimately, the design record exposes the parts of the product model which must be communicated to support operations, such as an engineering change, across the extended enterprise.

6.3.2 Representations used to support the engineering change operation

GUIDE represents the procedure employed by the company as a set of activity structures. Activity initiation is controlled by constraints to ascertain the suitability of the requesters which can be IBM engineers or subcontractors. GUIDE creates links between activities to identify sequences of operations and product dependencies. Figure 43 provides an illustration of how an engineering change might progress.



Problem Notification Activity
Investigate Product Change Request Activity (Sanity Check)
Investigate Component Change Activity
Response
Raise Product Engineering Change
Substitute Standard Part
Component Engineering Change Request
To correspond to the number of components affected
New Product Release
Advise Enterprise Product Register
Advise Other Company Locations and Close Coupled Contractors

Figure 43 : GUIDE activities used to represent IBM's engineering change procedure

The operation could be concluded earlier if a change in manufacturing process alone would have been sufficient. Formal associations are employed in two instances to secure specific elements of company practice:

- a product engineering change will involve the change of at least one part and cannot be concluded until the part change is complete;
- a new product release **must** be entered in the enterprise register followed by notification of the change to any party involved in the development of the product.

Central to the representation is the definition of the part[80] description structures: these provide the part identification (by specification of its part number and engineering change level) and a pointer to the specific activity, represented as a GUIDE structure, which describes the part design. Figure 44 is an illustration of the GUIDE structures employed. The overall product structure becomes an index to the part generation activities and enables:

- a track to be followed from the product to the design of the parts;
- several product configurations to be established using different combinations of parts;
- the design of the individual parts to become distributed and, possibly, be conducted outside the company.

Component relationships are represented as links across activities[81] and this enables product dependencies to be established. A Bill of Materials, represented as a child structure of the product description, is generated from a trace of the part design activity links. The bill of material for a specific release of the 9507 monitor is illustrated in plate 12.

The GUIDE methods supporting the engineering change provide the following functions:

- creation of the structure representing the product bill of material;
- generation of part and engineering change numbers according to the company's convention;
- evaluation of the capacity or authority of the engineering change initiator to conduct the change operation;
- retrieval of standard parts from a central company register;
- product data translation and communication (using PDES/STEP facilities if required).

[80]Denoted in figure 44 as assembly or component descriptions.

[81]In the design records of the activities involved.

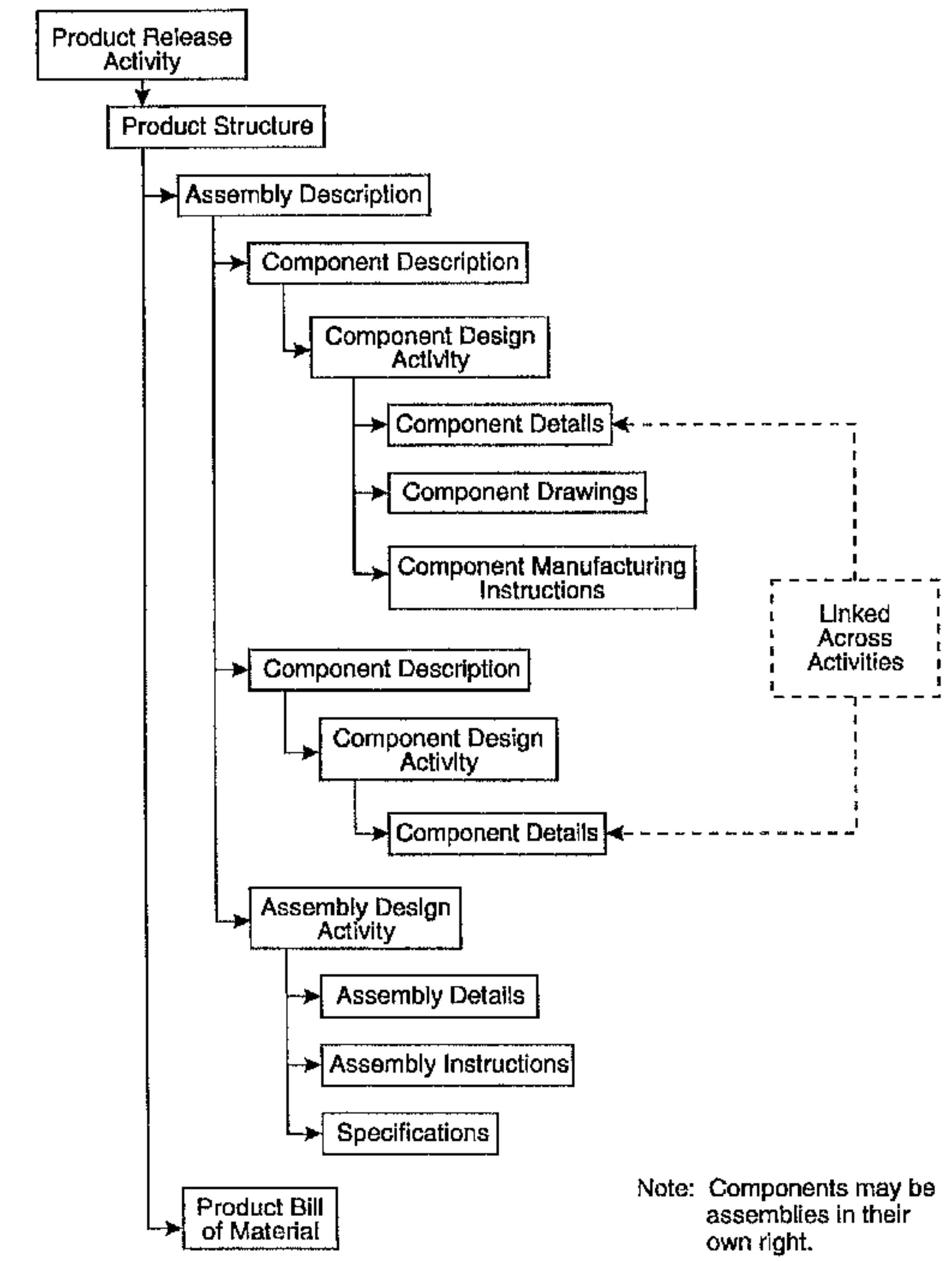


Figure 44 : Product structure representation as an aggregation of part design activities

6.3.3 Conduct of the engineering change on the 9507 display monitor

An IBM vendor contracted to manufacture the monitor bezel is experiencing problems with the flow of the plastic material during injection: the diameter of the injection point, corresponding in the bezel to the diameter of a fixing hole, needs to be increased by at least 15 percent. The procedure to rectify the problem progresses as follows:

1. Design change investigation

   The vendor's engineer examines the design record of the part and establishes the dependencies of the hole diameter value. GUIDE indicates that the hole diameter was obtained from the diameter of the fixing screw in a different design activity which refers to a company standard part and is, therefore, not amenable to design modification.

2. Replace standard part

   The vendor's engineer raises a problem notification using GUIDE (shown in plate 13), initiates a change activity and selects an alternative fastener from a list of company standard parts, retrieved by GUIDE, to replace the existing part. GUIDE warns that the replacement affects instances of other parts - the monitor back cover and the metal frame inside the monitor used to support the electronic components shown in plate 11 - suspends the replacement, initiates two design change activities as children of the fastener change activity, assigns them to the engineers responsible for the affected parts and notifies the product engineer of the proposed changes.

3. Product Release

   Upon completion of the part changes and the fastener substitution, the product engineer retrieves the new part descriptions and executes a GUIDE method which assigns new part numbers to the bezel, back cover and frame and a new engineering change number to the assembly which references them. The product engineer creates a copy of the product structure, replaces the changed parts and commits the new structure. GUIDE executes the method appropriate to the product structure commitment, which generates new activities to place an entry for the product in the corporate register and transmit the change information to the company's other locations and external contractors.

The provision of accurate product data translation services the conduct of functions such as engineering change but is not a goal in its own right. In

GUIDE's design record the influences and dependencies of the company's engineering function on the product model employed identify which parts of the product model need to be exchanged - the content of the communication - across the parties of an extended enterprise.

The conduct of the engineering change demonstrates GUIDE's provision of an operational environment where the extended enterprise behaves as a single company and site, with data communication taking place organically with the tasks engaged and transparently to the designers.

6.4 Design of a facility for design with ceramic composites

The field of design with ceramic composite materials is immature; little expertise exists for most of the design problems engaged. Component manufacture is based on a pre-form of fibres often constructed from several elements which might be made by knitting, weaving or braiding of the fibres. Component design and material design are, consequently, coincident activities and the manufacturing constraints are severe. Furthermore, components suffer micro-cracking under service conditions, which must be accounted for in the design **[Hopper et al, 1993]**.

It is not possible to prescribe, in terms of specific designer actions, an overall procedure which should be followed to reach a successful component design. The specific path followed will vary from design to design depending on, amongst others, the degree of similarity of the problem to others previously tackled, the experience of the designer and the availability of related expertise to the team undertaking the design. The facilities offered by GUIDE can be employed to develop a Ceramics Design System (CDS) which exploits distributed design resources and elicits expertise to aid designers in the design of ceramic components, within the operational context of a large engineering organisation.

6.4.1 The process of design with ceramic composites

Figure 45 provides one illustration of the design process which could take place. The component design evolves through series of iterations in which geometry and internal architecture are proposed and assessed. Physical performance, manufacturability and costs provide the basis for the assessment. The geometry and architecture proposals are made on the basis of information from these assessments and from other sources such as, for example, imposed specifications and constraints, component function, related previous designs and standards. The iterations may:

- vary considerably in terms of scope and time scales;
- involve the designer's effort alone;
- incorporate actions from other company or external functions.

The variability of the design process suggests that the system to support the design ought not attempt to prescribe a detailed procedure but, instead, to provide functions to support the generation, representation, communication and analysis of component designs which may be invoked by the designer throughout the process as seen fit.

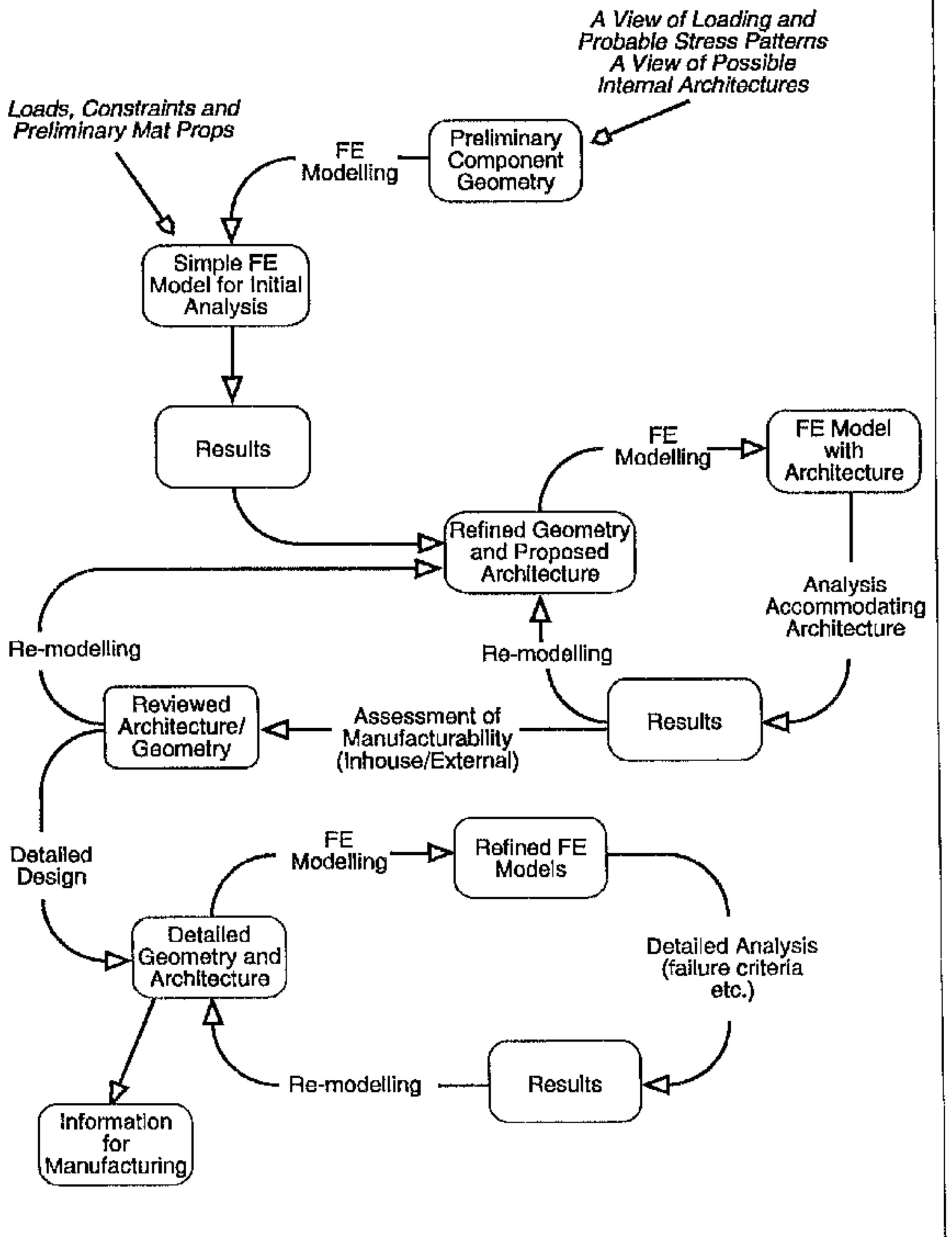

A View of Loading and Probable Stress Patterns
A View of Possible Internal Architectures
Loads, Constraints and Preliminary Mat Props
FE Modelling
Preliminary Component Geometry
Simple FE Model for Initial Analysis
Results
FE Modelling
FE Model with Architecture
Refined Geometry and Proposed Architecture
Analysis Accommodating Architecture
Re-modelling
Re-modelling
Reviewed Architecture/ Geometry
Assessment of Manufacturability (Inhouse/External)
Results
Detailed Design
FE Modelling
Refined FE Models
Detailed Geometry and Architecture
Detailed Analysis (failure criteria etc.)
Re-modelling
Results
Information for Manufacturing

Figure 45 : Illustrative design process for composite component

GUIDE's ability to support working entities which can be graphical, alphanumeric or process related allows modelling functions to be implemented alongside functions for data description, data communication and to invoke proprietary tools and applications. GUIDE can manage a diverse range of functions and make them available to the designer through a single interface.

Specifically, GUIDE is used to develop and maintain facilities to support the designers in:

- the modelling of composite components;
- the conduct of finite element analyses;
- accessing, updating and expanding a data base of material properties;
- the recording and reuse of design paths;
- providing context specific information to the users of the facility.

Composite modelling involves the subdivision of a complex geometric model into sub elements and the definition of cartesian points within each sub element to associate material fibre directions and property data with the model. The component is analysed into *architectural regions* - elementary volumes - which are formally represented as geometrical solids. The orientations of fibres or of principal directions of material properties in these regions are described by architectural tags. GUIDE provides the capacity to attach textual information and numerical data to the architectural regions related, for example, to materials properties and manufacturing technology using the architectural tags **[Hopper et al, 1993]**.

Finite element analyses require the support of modelling and meshing functions and access to applications and data communication. Particularly important is the communication of data which defines the composite components' internal structure from the design to the finite element solver. Data communication provides the ability to support:

- standard analyses from which component stress patterns and principal stress directions can be derived to guide the component internal architecture design;
- stress/displacement and thermal analyses in which the internal structure of the composite component is accommodated in terms of properties and material directions;

- micro-damage analyses in which the directions of the finite element model reinforcements used in the analysis may be derived from the architectural model.

The communication facilities are built upon combinations of process-oriented stereotypes supported by GUIDE methods.

The design record provides a mechanism for the accumulation of expertise which can be used in the future design of ceramic components. In addition to the actual outputs of a design, such as the component description in its various forms, a designer could obtain information concerning the design tasks undertaken and their sequence. The GUIDE design record is employed, with filtering to reduce its granularity, to provide a representation of the composite components design description.

6.4.2 The design of a T-Beam sub-element

The design system facilities provided by GUIDE are illustrated in the design of a T-beam composite sub element as shown in plate 14. The sub element design, using the developed Ceramics Design System, progresses as shown in figure 46 and involves the following stages:

1. Pre-analysis

   The T-beam sub element is modelled as a single architectural region. The geometry is meshed and a standard analysis using ABAQUS[82] is conducted to establish the principal stress directions (illustrated in plate 15) needed to guide the choice of fibre layout. GUIDE methods are used to communicate the information between CATIA, the geometric modeller used, and ABAQUS.

2. First Design Proposal

   The T-beam sub element is modelled as 3 architecture regions to simulate its anticipated construction (illustrated in plate 16). Architectural tags, defined using GUIDE, are associated with the individual regions to hold the particular material data and fibre directions. An ABAQUS analysis initiated to obtain

[82]ABAQUS is a finite element solver for non-linear analyses. It is a trademark of Hibbitt, Karlsonn and Sorensen (HKS), the ABAQUS developers.

values for inter-laminar direct stresses, as shown in plate 17, relies on GUIDE functions to access fibre directions and material properties.

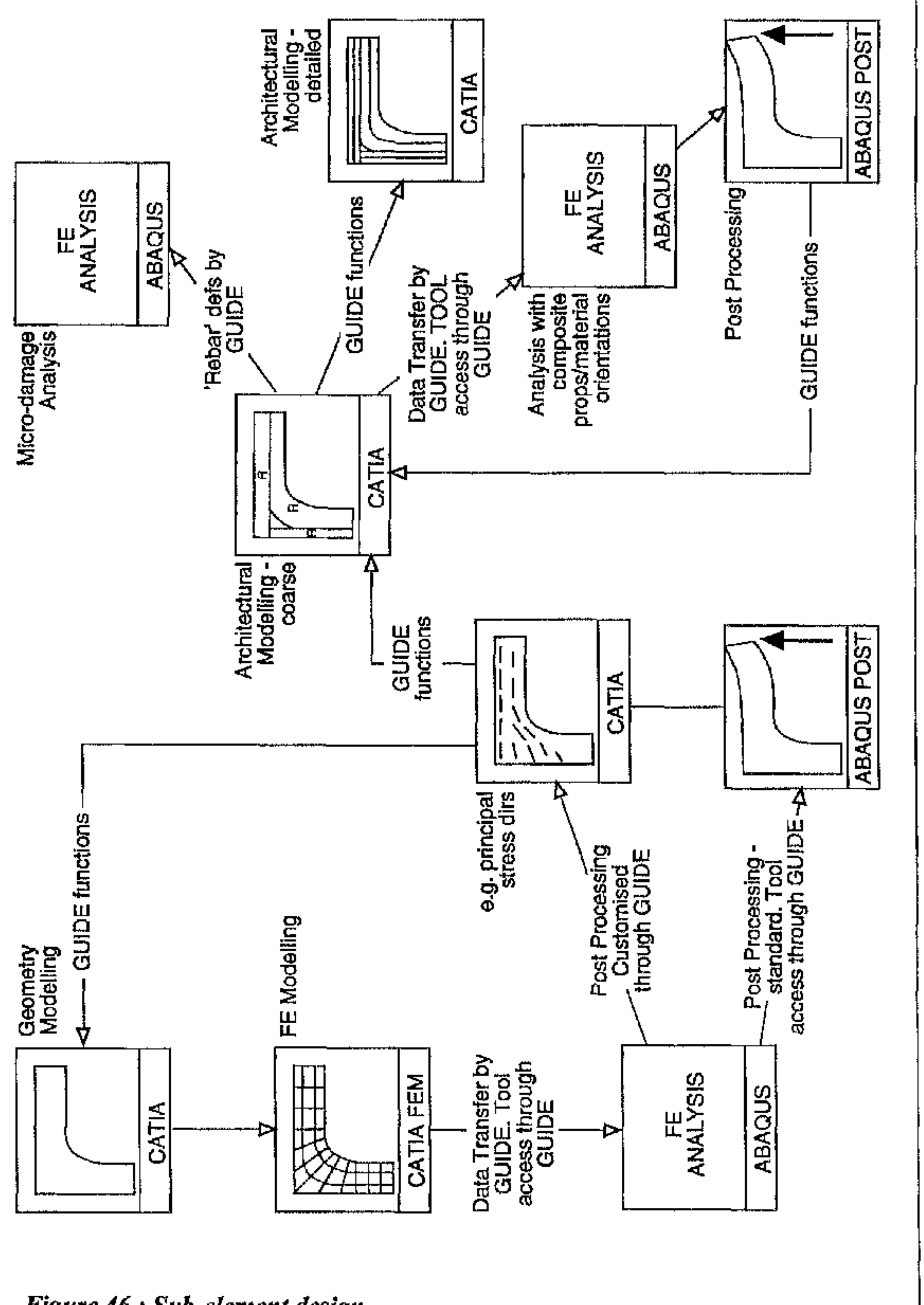

Geometry Modelling
CATIA
GUIDE functions
FE Modelling
CATIA FEM
Data Transfer by GUIDE. Tool access through GUIDE
FE ANALYSIS
ABAQUS
Post Processing Customised through GUIDE
Post Processing - standard. Tool access through GUIDE
e.g. principal stress dirs
CATIA
ABAQUS POST
GUIDE functions
Architectural Modelling - coarse
CATIA
Micro-damage Analysis
FE ANALYSIS
ABAQUS
'Rebar' defs by GUIDE
GUIDE functions
Architectural Modelling - detailed
CATIA
Data Transfer by GUIDE. TOOL access through GUIDE
Analysis with composite props/material orientations
FE ANALYSIS
ABAQUS
Post Processing
ABAQUS POST
GUIDE functions

Figure 46 : Sub-element design

3. Second Design Proposal

   Based upon the results of the 3 region T-beam sub element, the component is remodelled as 5 regions (illustrated in plate 18). The inter-laminar stresses (shown in plate 19) are calculated and the results are used to refine the regions to reduce internal stresses in the component. The process is iterated and leads to the sub-segmentation of the original regions until an acceptable solution has been reached.

The overall design record of the sub-element design activity is shown in plate 20. Plate 21 provides a detail of a T-beam sub-element design sub-activity.

The customised tools and processes yielded by the Ceramics Design System impress a discipline - the implicit design method - on the designer, but the CDS does not prescribe the design path. GUIDE provides **[Hopper et al, 1993]**, through the CDS, a method to: represent components manufactured from composites; fracture mechanics models and the analytical service and data requirements of these models; data base facilities including, for example, access to materials data bases; a record of the design path which has been followed in a form which permits its re-use and editing; on-line information to aid the designer.

A design team pursuing several design issues can construct a consistent set of models of a component or of an element of composite material, with the characteristics of its members being chosen to meet the needs of the various design tasks being engaged.

Closure

7.1 Conclusions

Design in engineering companies is turning to the use by the design authority of integrated, multi discipline teams and product analysis in terms of component technologies. The focus of company resources is on products rather than on specialist functions. Operationally the need is to: acquire resources which are functionally and geographically separated in the business routine of the company; provide designers with formal methods presented as integrated technologies; communicate data and deliver methods and procedures to design teams, the members of which may be distributed; maintain access to and sustain the integrity of existing product data. For this model to evolve effectively companies should employ:

- **a formally agreed and structured product model which includes, explicitly, manifestations of the products' evolution - i.e. knowledge of the data generation processes and of the design constraints which were applied, as well as of the data - and which can remain active throughout the products' lifetime, providing facilities for product development, configuration control, etc. and essential communications.**

GUIDE, the utility described in this work, supports the development, evolution and communication and maintains the integrity of a single product description. It employs structured representations of design entities and processes and the links between them, using combinations of structures, atoms and methods and a control mechanism which disciplines design. Thus:

- **GUIDE establishes a framework for the operation of distributed teams and the analysis of design into tasks which can be addressed by team elements working concurrently.**

GUIDE provides for:

- **the construction of structures containing atoms which point to other structures and methods to create aggregations of complex design entities and to form representations of the contributing technologies;**

- **the ordered representation of relationships as structures to describe links formed between its elements;**

- **the separation of relationships from the entities they link.**

- **the linking of pre-existing data, methods and facilities which designers have employed routinely with the GUIDE supported representations employed.**

Constraints are represented as methods which can evolve in time and may influence the application of GUIDE elements generally. The associated control mechanism manages conflict: the constraints upon an instance and its dependants must be satisfied before the setting or modification of a parameter can be finalised. Unsetting of an instance causes its dependants to be reset consistently with constraints affecting them. Relaxation of a design constraint is equivalent, in GUIDE, to a modification of the instance(s) to which the constraint applies and is made subject to the satisfaction of checks on any instances affected. This feature:

- **localises conflict and guards against the generation of design solutions in which constraints remain unsatisfied.**

GUIDE employs, as far as is possible, standard, commercially available software and applications. The elements it defines are implemented using a relational database and attributes characteristics to elements through relational links. Elements are linked using external relationships held within the relational schema. The GUIDE control mechanism thereby causes the relational schema to exhibit object oriented characteristics which facilitate the storage of knowledge representations but within the context of a distributed database. This facility:

- **secures the integrity of the representations and their availability for sharing across the organisation and its external contractors.**

GUIDE will benefit from advances made in the relational database architecture as these become available in international standards.

Data access is represented by GUIDE methods which supply atoms with values and cause data transformations. This facility sustains the communication with distributed engineering and product centred data bases and provides for the integration of design with company and external resources. This management of data access by GUIDE :

- **enables the association of disparate data forms within the knowledge representations.**

GUIDE maintains a design record - the route followed and the actions taken during design - which constitutes an accumulation of knowledge and the generation of expertise. The design record provides the actual outputs of a design and information concerning the design tasks undertaken and their sequence. In essence:

- **the design record contains comprehensive meta-knowledge and enables GUIDE to contribute to and acquire design knowledge.**

GUIDE facilitates the construction and testing of design solutions and can be used to examine design methods and processes by allowing rapid prototyping. Because of its internal discipline, GUIDE allows representations of design entities and procedures to be managed centrally and communicated to the designers without loss of operational flexibility (the independence of designers).

7.2 Future prospects

Design functions and the contribution of designers are poorly defined and evaluated. Lacking good measures of efficiency and productivity company investments of resources in design are uncertain and often inadequate. The design record is a basis for independent audit of designer actions in real time, because it exposes tasks more quickly and precisely. The disciplined construction of the design record coupled with a method to analyse its content as design proceeds could provide a facility to measure design efficiency and productivity and define the metric required for auditing the design process in general. Hence:

- **the design record could enable design traceability, provide for audit during the design process, sustain status evaluations and provide for design regression.**

In inter-company communications, the provision of accurate product data translation services the conduct of engineering functions (e.g. manufacture) remotely, but does not secure a basis for design by a multidisciplinary team which may be geographically distributed. In GUIDE's design record, the influences and dependencies of the company's engineering functions on the product model identify which parts of the model need to be shared with the collaborating agents.

The design record is a basis for inter-company communication, provided it can be expressed in a standard form. The facilities available in the PDES/STEP standard suggest that this is possible.

The introduction of a product into service necessitates the development of procedures to guide non-experts in performing product maintenance or to cause the modifications required to suit particular markets. Maintenance procedures must change when products evolve. Companies adopt new techniques - e.g. facilities management - to resolve maintenance and related problems. The design record contains information on the constraints and concepts which influence the operation of mechanisms and products. With suitable advances in modular computing, an engine capable of interpreting the record while a product is in service might be developed which would make products self-referencing and self-monitoring.

GUIDE offers designers with a range of design functions through a single interface and without prescription on operations. It provides access to distributed data, communication facilities and generates a record of the entire design activity. Some of the applications to which GUIDE can be put have been illustrated in this work.

References

Akagi, S. (1991). Expert system for engineering design based on object-oriented knowledge representation concept. In: *Artificial Intelligence in design*, edited by D.T. Pham. Springer-Verlag.

Andreasen, M.M. (1991). Design Methodology. In: *Journal of Engineering Design*, Volume 2, Number 4.

Andreasen, M.M. (1992). Designing on a designers workbench. In: *Proceedings of the 9th WDK workshop*, Rigi, CH.

Arnold, C.B. (1993). Data trends and directions. In: *DB2 Technical Conference Proceedings*, La Hulpe, Belgium.

Banares-Alcantara, R. (1991). Representing the engineering design process: two hypotheses. In: *Artificial Intelligence in Design*, edited by J.S. Gero. Butterworth-Heinemann.

Baral, C., Kraus, S. and Minker, J. (1991). Combining multiple knowledge bases. In: *IEEE Proceedings on Knowledge and Data Engineering*, Volume 3, Number 2.

Bauert, F. (1993). Creative and systematic team-work using abstract concepts and concrete entities. In: *Proceedings of ICED 93*, The Hague, Holland, Volume 1.

Bauert, F. and Ball, N.R. (1993). The integrated design framework: blackboards as a communication medium between designers and software modules. In: *Proceedings of ICED 93*, The Hague, Holland, Volume 3.

Beitz, W. and Feldhusen, J. (1991). Management systems and program concepts for an integrated CAD process. In: *Research in Engineering Design*. Springer-Verlag.

Borning, A. (1987). Constaint hierarchies. In: *Proceedings, European Conference on Object Oriented Programming*, Paris, France.

Bond, A.H. (1992). A predicate logic approach to CAD/CAM modelling. In: *Journal of AI for engineering design, analysis and manufacturing*, Volume 6, Number 1.

Bradley, D.A. and Buur, J. (1993). The representation of mechatronic systems. In: *Proceedings of ICED 93*, The Hague, Holland, Volume 1.

Brandenburg, W. (1992). Product specific modelling with an object-oriented approach. In: *European CATIA User's Association Conference Proceedings*.

Brooks, F.P. (1975). The mythical man month. In: *Essays on Software Engineering*. Addison-Wesley.

Brown, D. (1989). Using design history systems for technology transfer. In: *Proceedings of the MIT JSME workshop on cooperative product development*, Boston, USA.

Butterfield, W.R., Green, M.K., Scott, D.C. and Stoker W. (1985). Part features for process planning. In: *CAM-I Report: C-85PPP-03-1985.*

Cammarata, S.J. and Melkanoff, M.A. (1986). An interactive data dictionary facility for CAD/CAM databases. In: *First Workshop on Expert Database Systems*, pp. 423-440.

Carter, I.M. and MacCallum, K.J. (1991). A software architecture for design co-ordination. In: *Artificial Intelligence in Design*, edited by J.S. Gero. Butterworth-Heinemann.

Chandrasekaran, B. (1981). Natural and social system metaphors for distributed problem solving: introduction to the issue. In: *IEEE Transactions on Systems, Man and Cybernetics*, Volume SMC-11, Number 1.

Dasgupta, D. and McGregor, D.R. (1991). A structured Genetic Algorithm. In: *IKBS-2-92*, University of Strathclyde, 1991.

Dayal, U. and Smith, J.M. (1986). PROBE: A knowledge-oriented database management system. In: *On Knowledge Based Management Systems*, edited by M.L. Brodie and J. Mylopoulos. Springer-Verlag.

Dilts, D.M. and Wu, W. (1991). Using knowledge based technology to integrate CIM databases. In: *IEEE Transactions on Knowledge and Data Engineering*, Volume 3, Number 2.

Dittrich, K.R., Kotz, M. and Mulle, J.A. (1986). An Event/Trigger mechanism to enforce complex consistency constraints in design databases. In: *SIGMOD Record*, 15(3):22-36, September 1986.

Dixon, J.R. and Simmons M.K. (1983). Computers that design expert systems for mechanical engineers. In: *Proceedings, ASME International Conference on Computers in Mechanical Engineering*.

Eastman, C.M., Bond, A.H. and Chase S.C. (1991). A data model for designing databases. In: *Artificial Intelligence in Design*, edited by J.S. Gero. Butterworth-Heinemann.

Eastman, C.M. (1992). A data model analysis of modularity and extensibility in building databases. In: *Building and the Environment*, Volume 27, Number 2.

Ehrlenspiel, K. and Schaal, S. (1991). Ways to smarter CAD systems - Exemplified through design - concurrent calculation. In: *Proceedings, ICED '91*, Zurich, August 27-29 1991, pp. 609-617.

Eppinger, S.D. (1991). Model based approaches to managing concurrent engineering. In: *Journal of Engineering Design,* Volume 2, Number 4.

Erens, F., McKay, A. and Bloor, S. (1993). Shortcomings of today's design frameworks. In: *Proceedings of ICED 93,* The Hague, Holland, Volume 3.

Faux, I.D. (1986). Reconsiliation of design and manufacturing requirements for product description data, using functional primitive part features. In: *CAM-I Report: R-86-ANC/GM/PP-01.1-1986.*

Finger, S., Gardner, E. and Subrahmanian, E. (1993). Design support systems for concurrent engineering: a case study in large power transformer design. In: *Proceedings of ICED 93,* The Hague, Holland, Volume 3.

Garcia, A.C.B. and Howard, H.C. (1992). Acquiring design knowledge through design decision justification. In: *AI EDAM,* Volume 6, Number 1. Academic Press Ltd.

Giacometti, F. and Chang T.C. (1990). Object orientated design for modelling parts, assemblies and tolerances. In: *Proceedings, 2nd International Conference on Technology of Object Orientated Language and Systems,* Paris.

Gielingh, W.F. (1990). Computer integrated construction : A major STEP forward. In: *Computer Integrated Construction 1,* W. McLelland Ltd.

Glegg, G.L. (1969). The design of design. Cambridge University Press.

Hayes-Roth, B. (1983). The blackboard architecture: A general framework for problem solving. In: *Heuristic Programming Project Report HPP-83-30,* Stanford University.

Held H.J., Jager K.W., Kratz N. and Schneider M. (1991). Knowledge based engineering assistance. In: *Artificial Intelligence in Design,* edited by J.S. Gero. Butterworth-Heinemann.

Hopper, I., McCafferty, J. and Gibson, R. (1993). Ceramic composites for aerospace propulsion: Final report. September 1993, Glasgow University.

The Computer Application Committee of the IMechE (1988). A five year strategy programme for the process industries.

Inui, M. and Kimura, F. (1991). Design of machining processes with dynamic manipulation of product models. In: *Artificial Intelligence in design,* edited by D.T. Pham. Springer-Verlag.

Kroll, E., Lenz, E. and Wolberg, J.R. (1991). Intelligent analysis and synthesis tools for assembly-oriented design. In: *Artificial Intelligence in design,* edited by D.T. Pham. Springer-Verlag.

Lawson, B. (1980). How Designers Think. Architectural Press, London.

Logan, B., Millington, K. and Smithers T. (1991). Being economical with the truth: assumption based context management in the Edinburgh designer System. In: *Artificial Intelligence in Design*, edited by J.S. Gero. Butterworth-Heinemann.

MacKellar, B.K. (1991). A constraint-based model of design object versions. University of Connecticut.

Mantyla, M., Opas, J. and Puhakka J. (1987). A Prototype system for generating process planning of prismatic parts. In: *Proceedings of the IFIP.*

McGinnis, B.D. and Ullman, D.G. (1990). The evolution of commitments in the design of a component. In: *Proceedings of the International Conference on Engineering Design*, Harrogate, UK.

McMahon, C.A., Sims Williams, J.H. and Brown, K.N. (1993). A transformation model for the integration of design computing. In: *Proceedings of ICED 93,* The Hague, Holland, Volume 3.

Medland, A.J. (1993). The dynamic reconfiguration of rules within a constraint modelling process to address evolving design problems. In: *Proceedings of ICED 93,* The Hague, Holland, Volume 3.

Minsky, M. (1975). A Framework for representing knowledge. In: *The Psychology of Computer Vision*, edited by P.H. Winston. McGraw-Hill.

Moynihan, G.P. (1993). Application of expert systems to engineering design. In: *Concurrent engineering: Contemporary issues and modern design tools.* Edited by H.R. Parsaei and W.G. Sullivan. Chapman-Hall 1993. ISBN 0-14246-510-8.

Myers, L., Snyder, J. and Chirica, L. (1992). Database usage in a knowledge based environment for building design. In: *Building and Environment*, Volume 27, Number 2, Pergamon Press Ltd.

Nagy, R.L., Ullman, D.G. and Dietterich, T.G. (1992). A data representation for collaborative mechanical design. In: *Research in Engineering Design.* Springer-Verlag.

Nguyen, G.T. and Rieu, D. (1991). Representing design objects. In: *Artificial Intelligence in Design*, edited by J.S. Gero. Butterworth-Heinemann.

Pham, D.T. and Tacgin, E. (1991). Techniques for inteliigent computer aided design. In: *Artificial Intelligence in Design*, edited by D.T. Pham. Springer-Verlag.

Raczkowsky, J., Rembold, U. and Dillman, R. (1990). A blackboard system for the diagnosis of robot assembly tasks. In: *Proceedings of the 1st conference on Artificial Intelligence and Expert Systems in Manufacturing,* ISBN 1-85423-075-1.

Reddy, M. and O'Hare, G.M.P. (1991). The blackboard model: a survey of its application. In: *Artificial Intelligence Review*, Volume 5.

Sackett, P.J. and Brophy, G. (1992). Feature Modelling in CATIA. In: *European CATIA User's Association Conference Proceedings*, 1992.

Scott, B.F. (1988). Creation of an infrastructure for engineering design. In: *Proposal to the SERC for a University Research Centre for Engineering Design*, November 1988, Gilmorehill, Glasgow.

Sheth, S. (1991). Product data management and supporting infrastructure for an enterprise. In: *Engineering Databases : An Engineering Resource*, ASME .

Simons, H.A. (1970). The sciences of the artificial. MIT Press.

Simons, H.A. (1973). The structure of ill structured problems. In: *Artificial Intelligence 4,* pp. 181-201.

Smith, B. (1986). International efforts in product data exchange. In: *Proceedings of the MICAD86 Conference*, Hermes, Paris.

Smith, R.G. and Davis, R. (1981). Frameworks for cooperation in distributed problem solving. In: *IEEE Transactions on Systems, Man and Cybergenetics,* Volume SMC-11 , Number 1.

Sriram, D. (1986). DESTINY: A model for integrated structural design. In: *International Journal for AI in Engineering*, Volume 1, Number 2.

Stauffer, L.A. and Slaughterback-Hyde, R.A. (1990). The nature of constraints and their effect on quality and satisfying. Department of Mechanical Engineering, University of Idaho, Idaho.

Thornton, A.C. and Johnson, A. (1993). Constraint specification and satisfaction in embodiment design. In: *Proceedings of ICED 93,* The Hague, Holland, Volume 3.

Trousse, B. (1993). Integration of constraint-based systems into design support tools. In: *Proceedings of ICED 93,* The Hague, Holland, Volume 3.

Tsiotsias, A.S., Muir, C.D.R and Crombie, A.M. (1994). Product development and design methodology using PDES : Annual report. Document No. 94136GUM01144 , Department of Mechanical Engineering, Glasgow University.

Ullman, D.G. (1991). Design histories: archiving the evolution of products. In: *Proceedings of the DARPA workshop on manufacturing*, Salt Lake, USA .

Visser, W. (1993). Collective design: a cognitive analysis of co-operation in practice. In: *Proceedings of ICED 93,* The Hague, Holland, Volume 1.

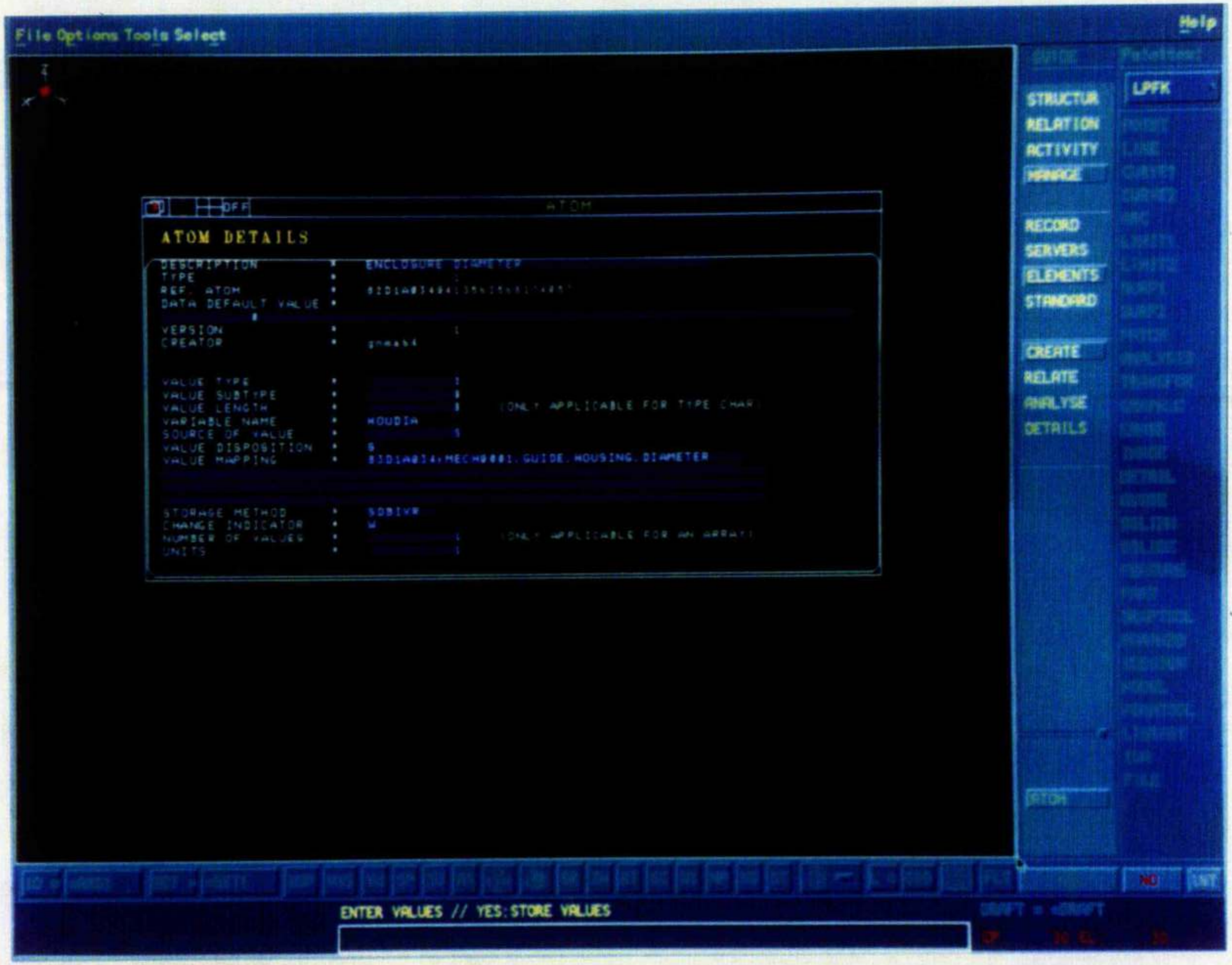
Vujosevic, R. and Kusiak, A. (1991). Data base requirements for concurrent design systems. In: *Engineering Databases: An Engineering Resource, ASME 1991.*

Yeh, R.T. (1992). Notes on Concurrent Engineering. In: *IEEE Transactions on Knowledge and Data Engineering*, Volume 4, Number 5.

Young, R.E., Greef, A. and O'Grady, P. (1991). SPARK: an artificial intelligence constraint network system for concurrent engineering. In: *Artificial Intelligence in Design*, edited by J.S. Gero. Butterworth-Heinemann.

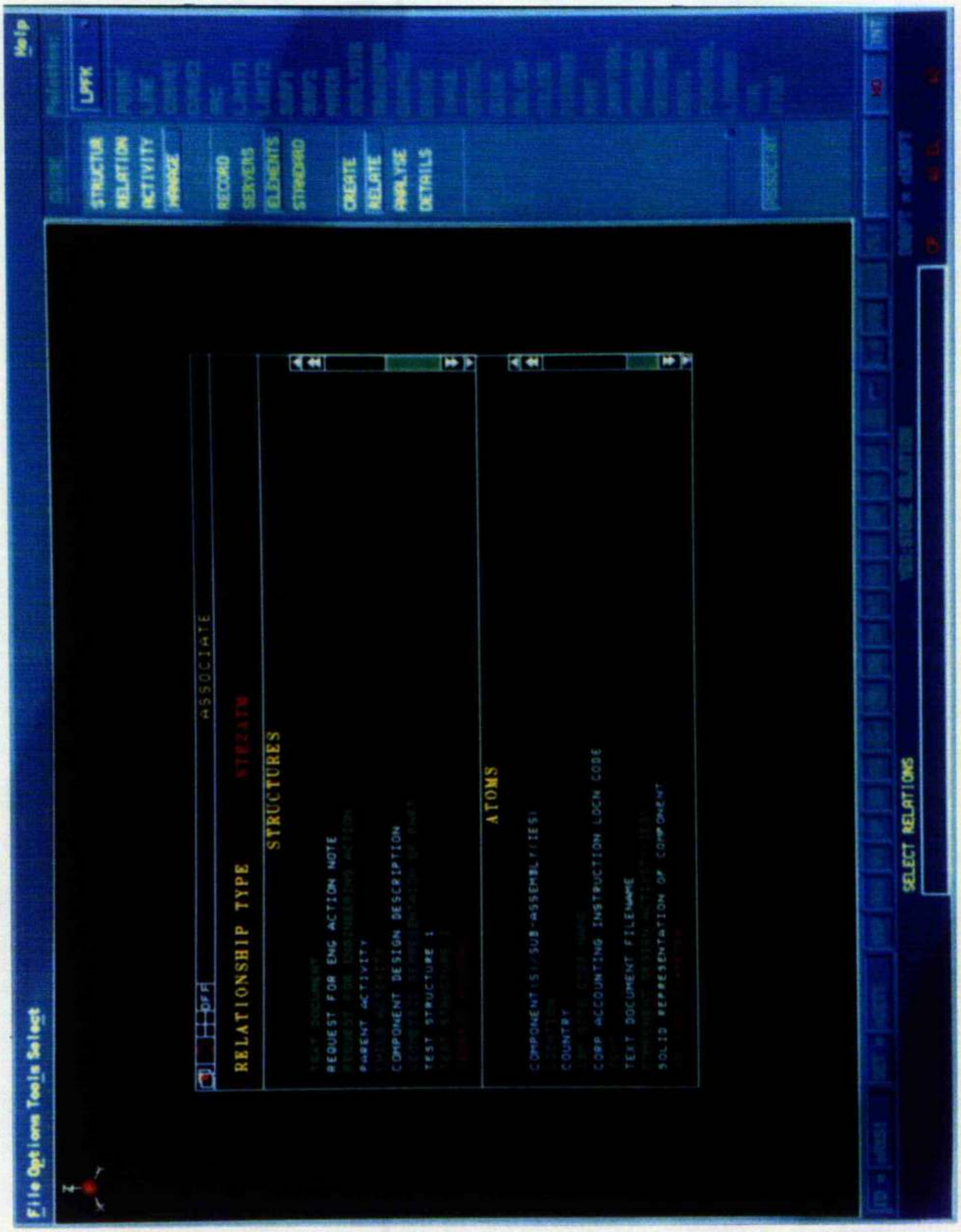
Zhu, G., Gao, J., Hu, W., Zhou, J. and Yu, J. (1993). A product definition model for concurrent design. In: *Proceedings of ICED 93*, The Hague, Holland, Volume 3.

Plates



```
ATOM DETAILS

DESCRIPTION          :   ENCLOSURE DIAMETER
TYPE                 :
REF. ATOM            :   $ID1A$I494:[illegible]
DATA DEFAULT VALUE   :

VERSION              :              1
CREATOR              :   gnma64

VALUE TYPE           :
VALUE SUBTYPE        :
VALUE LENGTH         :               (ONLY APPLICABLE FOR TYPE CHAR)
VARIABLE NAME        :   HOUDIA
SOURCE OF VALUE      :              5
VALUE DISPOSITION    :   6
VALUE MAPPING        :   $ID1A$I4:MECH$001.GUIDE.HOUSING.DIAMETER

STORAGE METHOD       :   SDBIVR
CHANGE INDICATOR     :   W
NUMBER OF VALUES     :              1  (ONLY APPLICABLE FOR AN ARRAY)
UNITS                :              1
```

Plate 1: Creation of an atom definition



ASSOCIATE

RELATIONSHIP TYPE

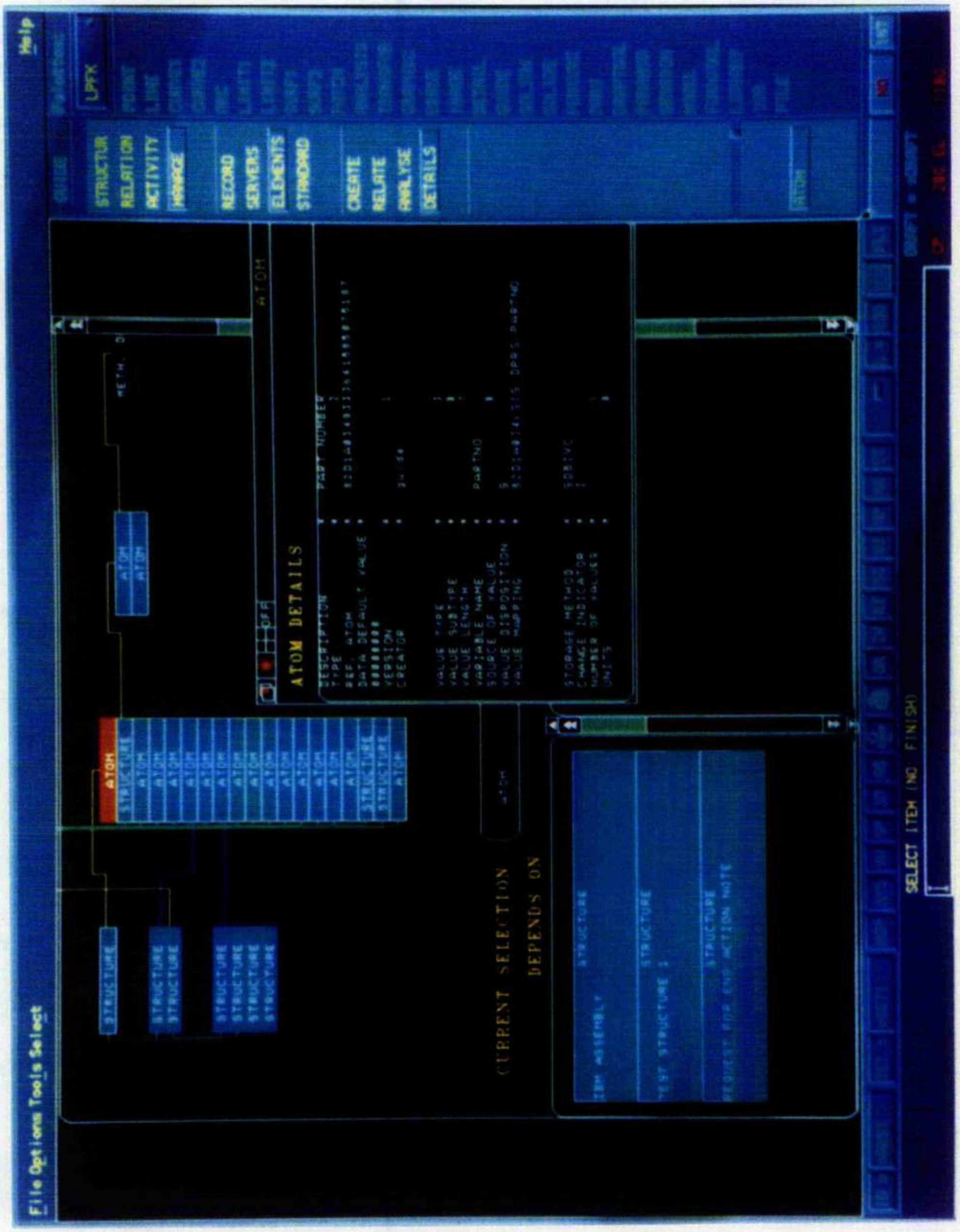
STRUCTURES

REQUEST FOR ENG ACTION NOTE
PARENT ACTIVITY
COMPONENT DESIGN DESCRIPTION
TEST STRUCTURE 1

ATOMS

COMPONENT(S)/SUB-ASSEMBLY(IES)
COUNTRY
CORP ACCOUNTING INSTRUCTION LOCN CODE
TEXT DOCUMENT FILENAME
SOLID REPRESENTATION OF COMPONENT

Plate 2: Association of atoms to structures



ATOM DETAILS

```
DESCRIPTION          :   PART NUMBER
TYPE                 :               I
REF. ATOM            :   82D1A0340333641585075187
DATA DEFAULT VALUE   :
########
VERSION              :               1
CREATOR              :   guide

VALUE TYPE           :               1
VALUE SUBTYPE        :               8
VALUE LENGTH         :               7
VARIABLE NAME        :   PARTNO
SOURCE OF VALUE      :               8
VALUE DISPOSITION    :   S
VALUE MAPPING        :   82D1A0341SIS DPRS PARTNO

STORAGE METHOD       :   SDBIVC
CHANGE INDICATOR     :   I
NUMBER OF VALUES     :               1
UNITS                :               8
```

STRUCTURE
IBM ASSEMBLY
TEST STRUCTURE 1
REQUEST FOR ENG ACTION NOTE

Plate 4: Analysis of an atom associated with a structure

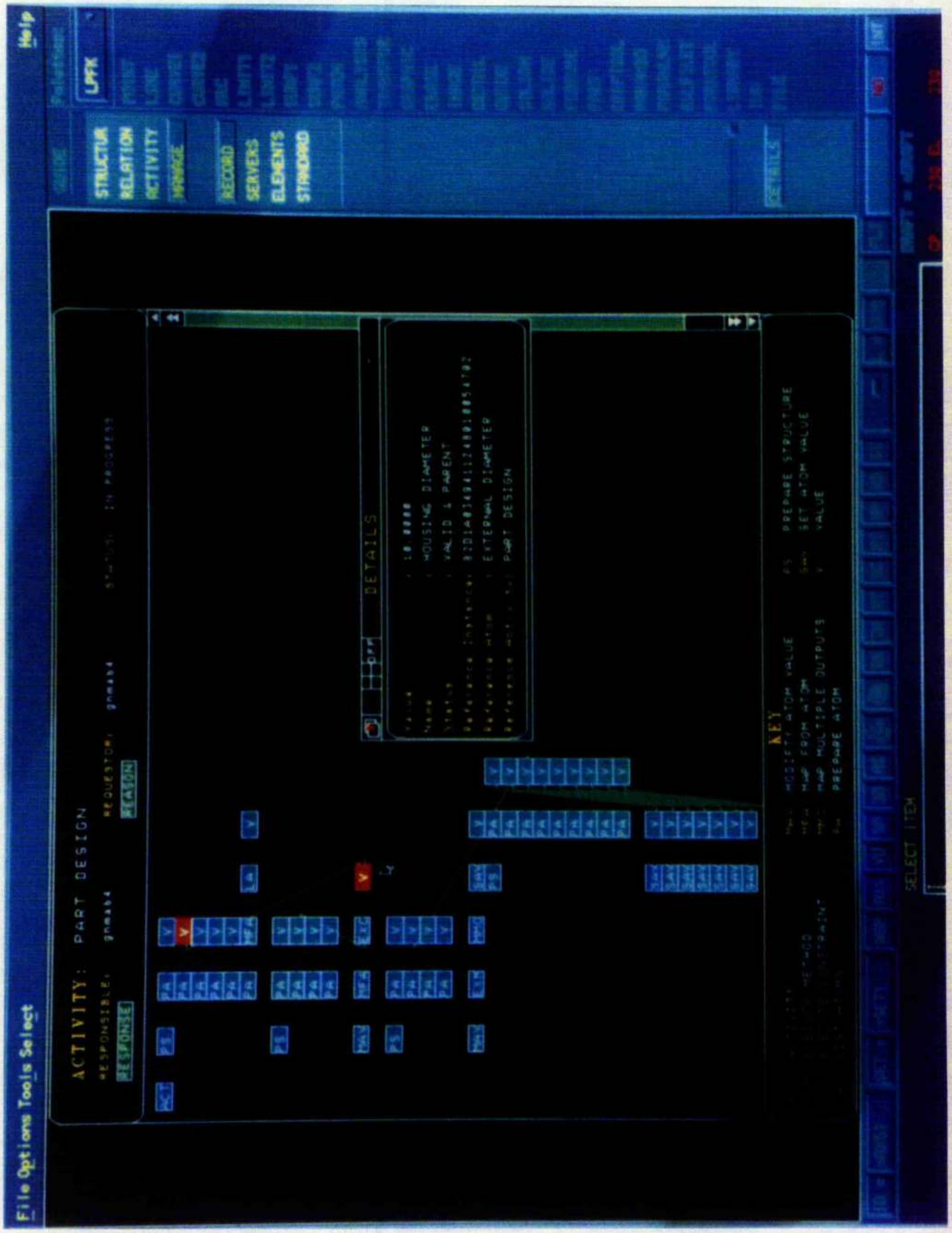
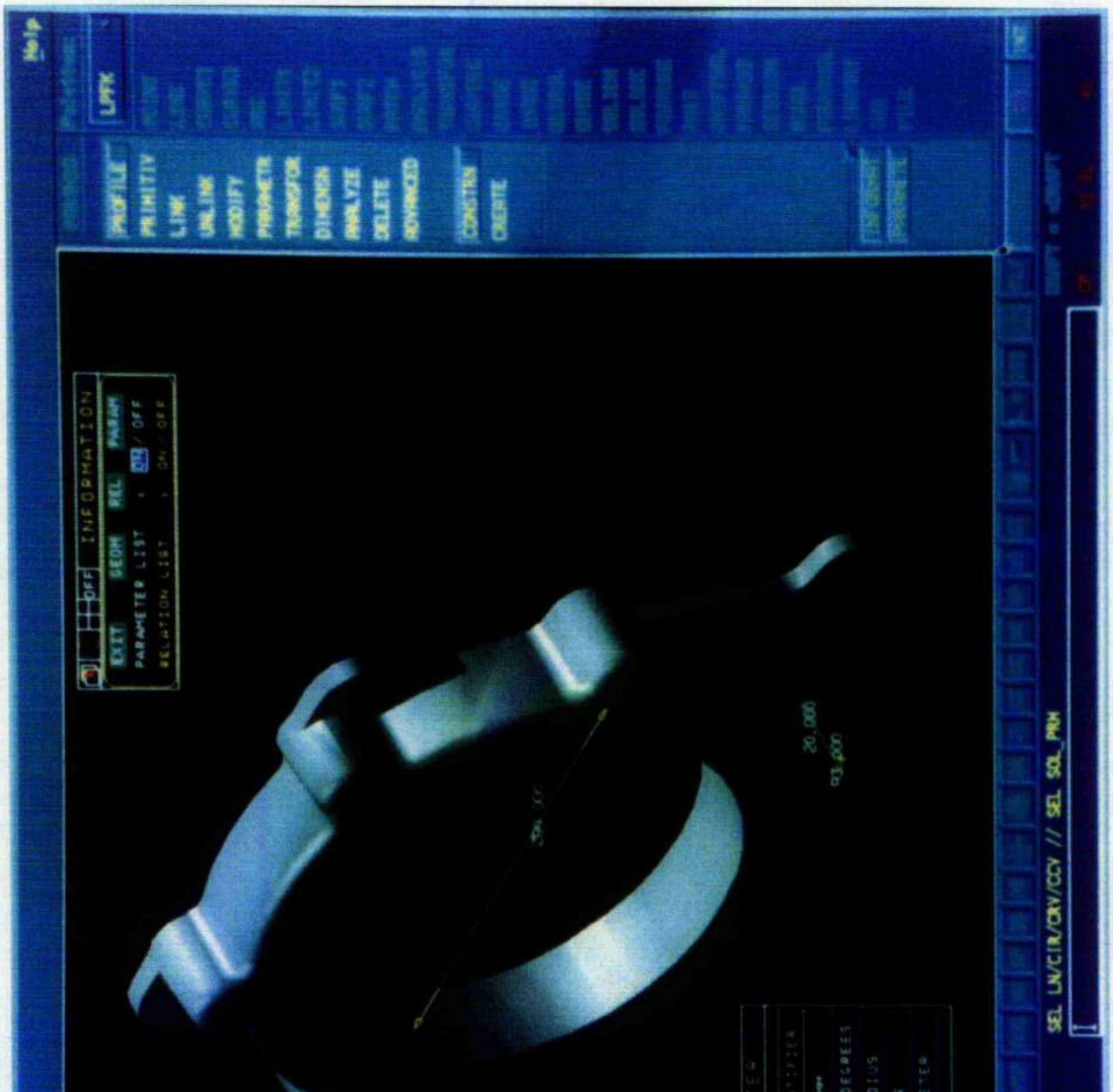


Plate 5: View of a design record incorporating instance details



INFORMATION

EXIT GEOM REL PARAM

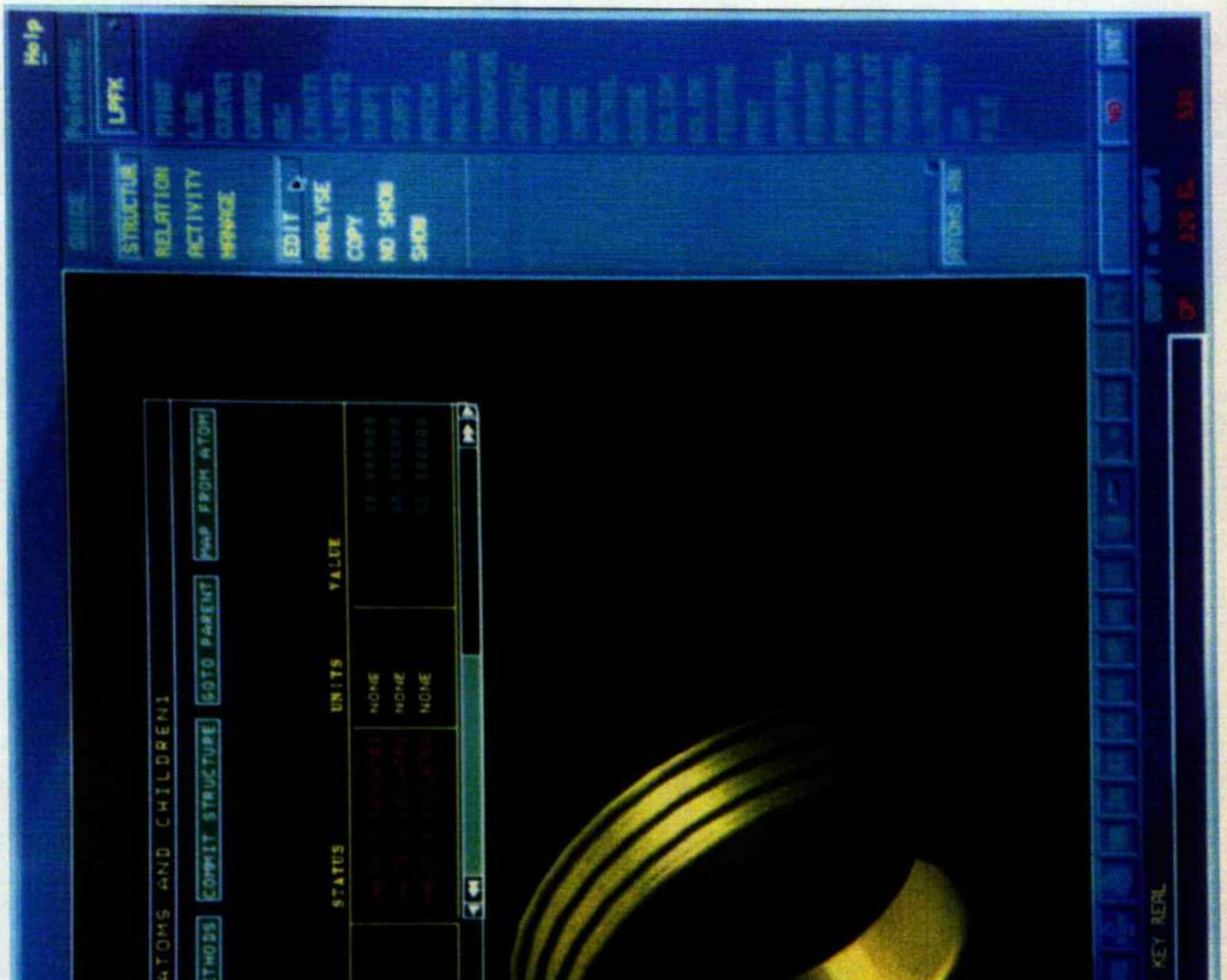
PARAMETER LIST : [✓ ON] / OFF

RELATION LIST : ON / OFF

PARAMETER

TYPE	VALUE	PARAM IDENTIFIER
INPUT	20.000	DEPTH IN mm
INPUT	1.000	DRAFT IN DEGREES
INPUT	1.000	FILLET RADIUS
INPUT	34.000	FACE ANGLE
INPUT	40.000	HOLE DIAMETER

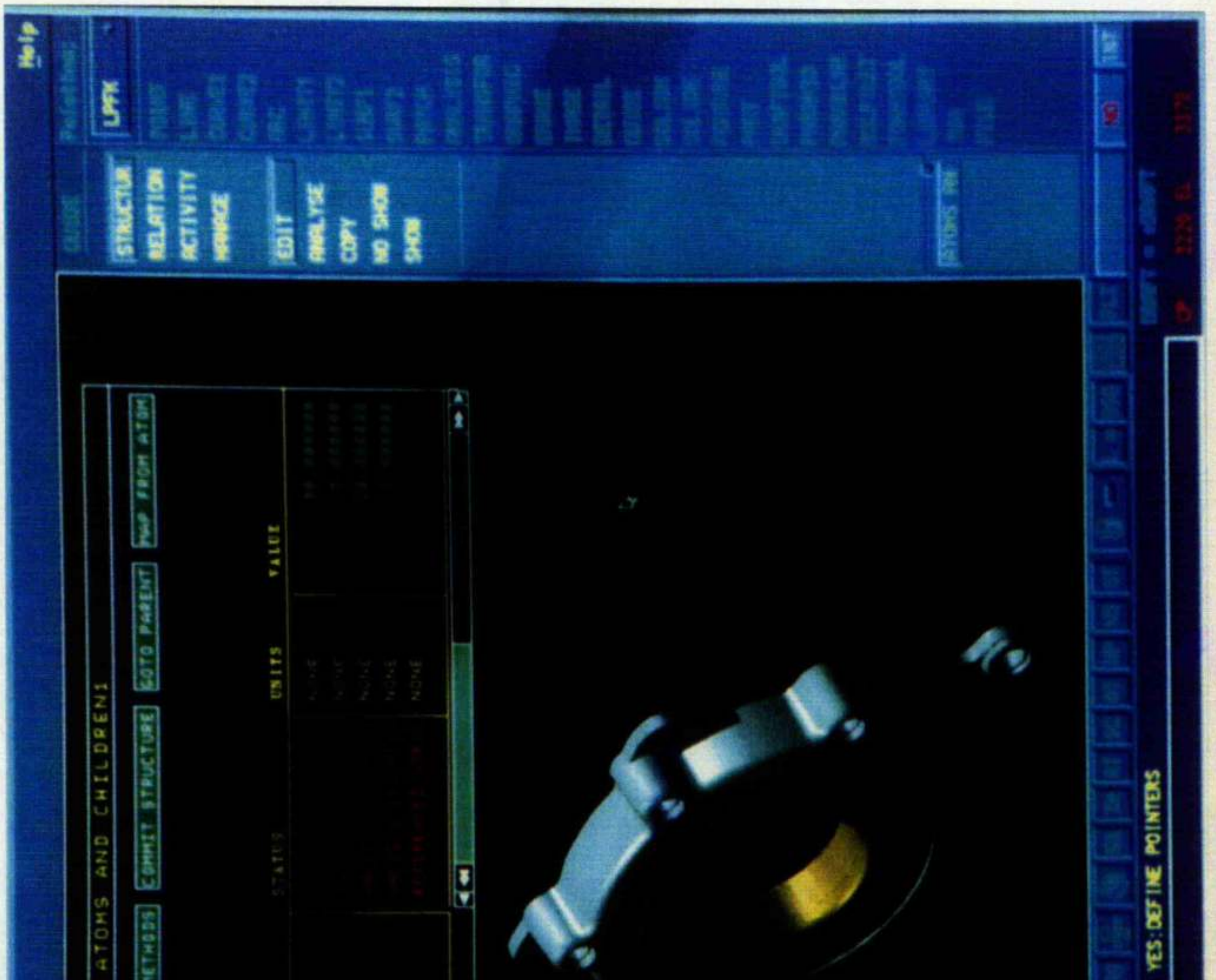
Plate 6: Parametrised crankshaft oil seal housing



Structure: CRANKSHAFT OIL SEAL
Family : PARTS
Status : PREPARED

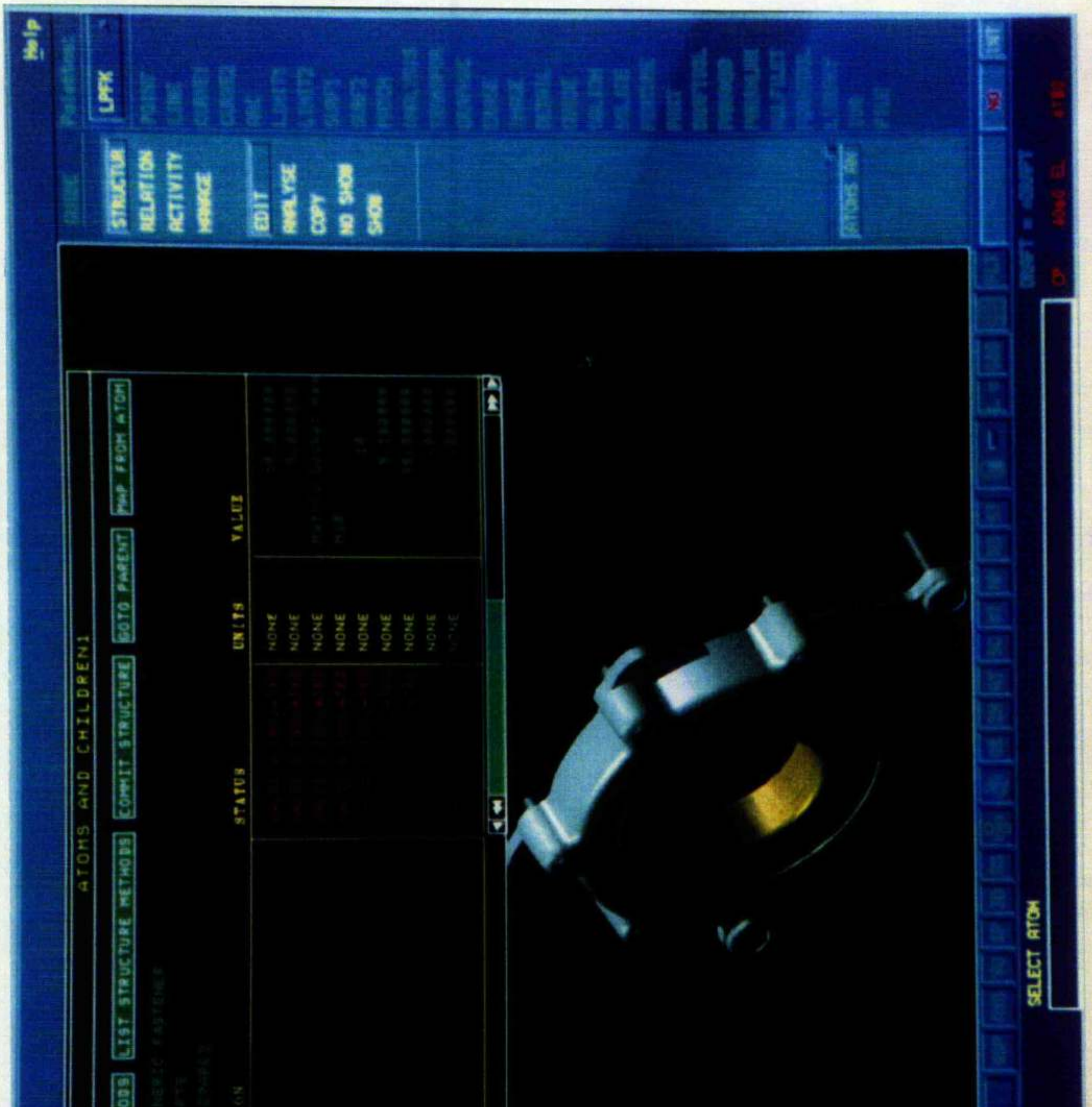
ATOM DESCRIPTION	STATUS	UNITS	VALUE
BODY THICKNESS	[illegible]	NONE	[illegible]
EXTERNAL DIAMETER	[illegible]	NONE	[illegible]
INTERNAL DIAMETER	[illegible]	NONE	[illegible]

Plate 7: GUIDE structure describing the oil seal



ATOM DESCRIPTION	STATUS	UNITS	VALUE
HOUSING DIAMETER	[illegible]	NONE	[illegible]
BODY DRAFT ANGLE	[illegible]	NONE	[illegible]
BODY THICKNESS	[illegible]	NONE	[illegible]
FACE FILLET RADIUS	[illegible]	NONE	[illegible]
[illegible]	[illegible]	NONE	

Plate 8: GUIDE structure describing the oil seal housing



ATOM DESCRIPTION	STATUS	UNITS	VALUE
BORE_DIAMETER	[illegible]	NONE	[illegible]
BORE_DEPTH	[illegible]	NONE	[illegible]
HEAD_TYPE	[illegible]	NONE	[illegible]
TYPE	[illegible]	NONE	[illegible]
THREAD_SIZE	[illegible]	NONE	[illegible]
BODY_DRILL	[illegible]	NONE	[illegible]
LENGTH	[illegible]	NONE	[illegible]
SINK_DEPTH	[illegible]	NONE	[illegible]
SINK_ANGLE	[illegible]	NONE	[illegible]

Plate 9: Creation of housing fixing holes by instantiation of fastener structure with data retrieved from a data base search

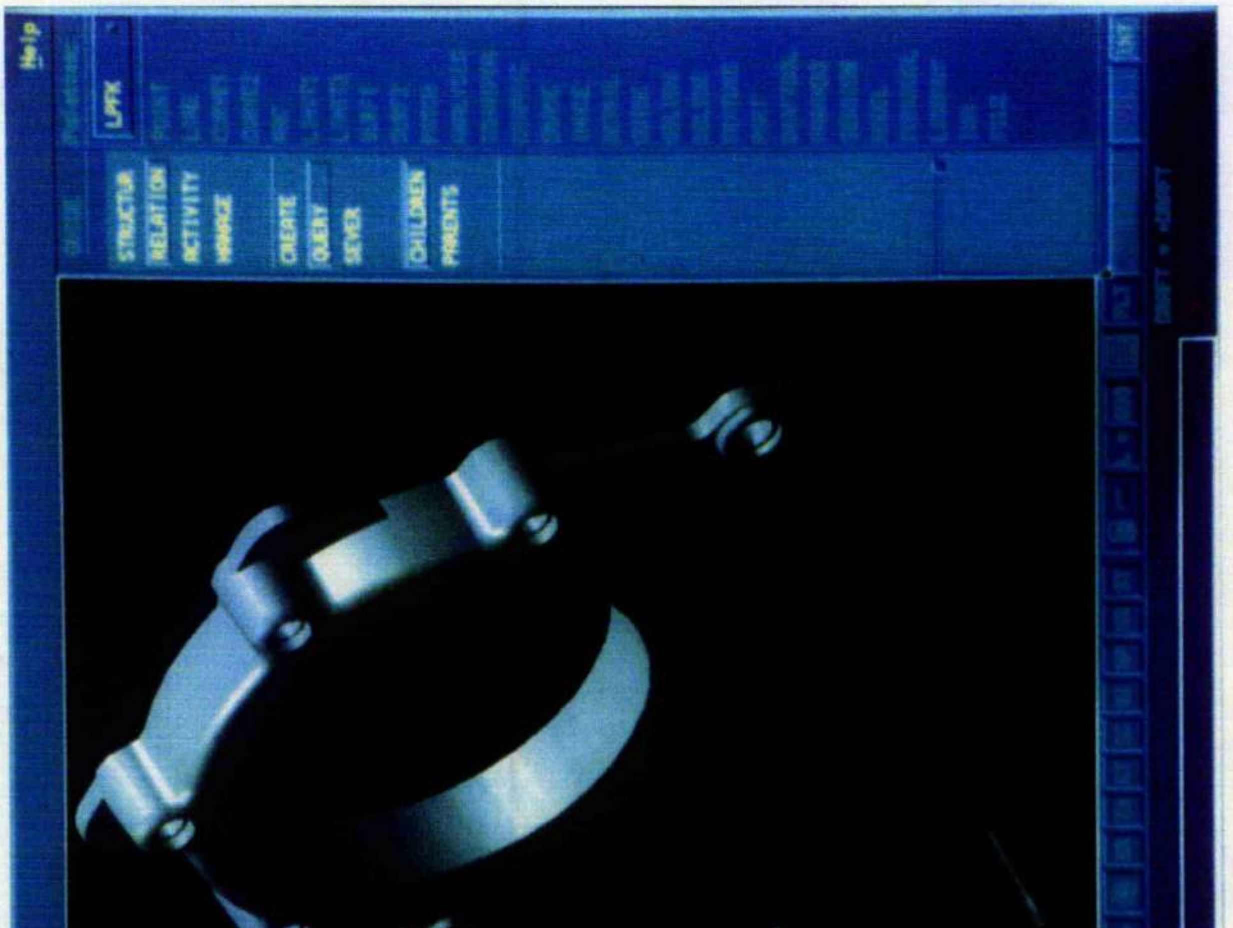
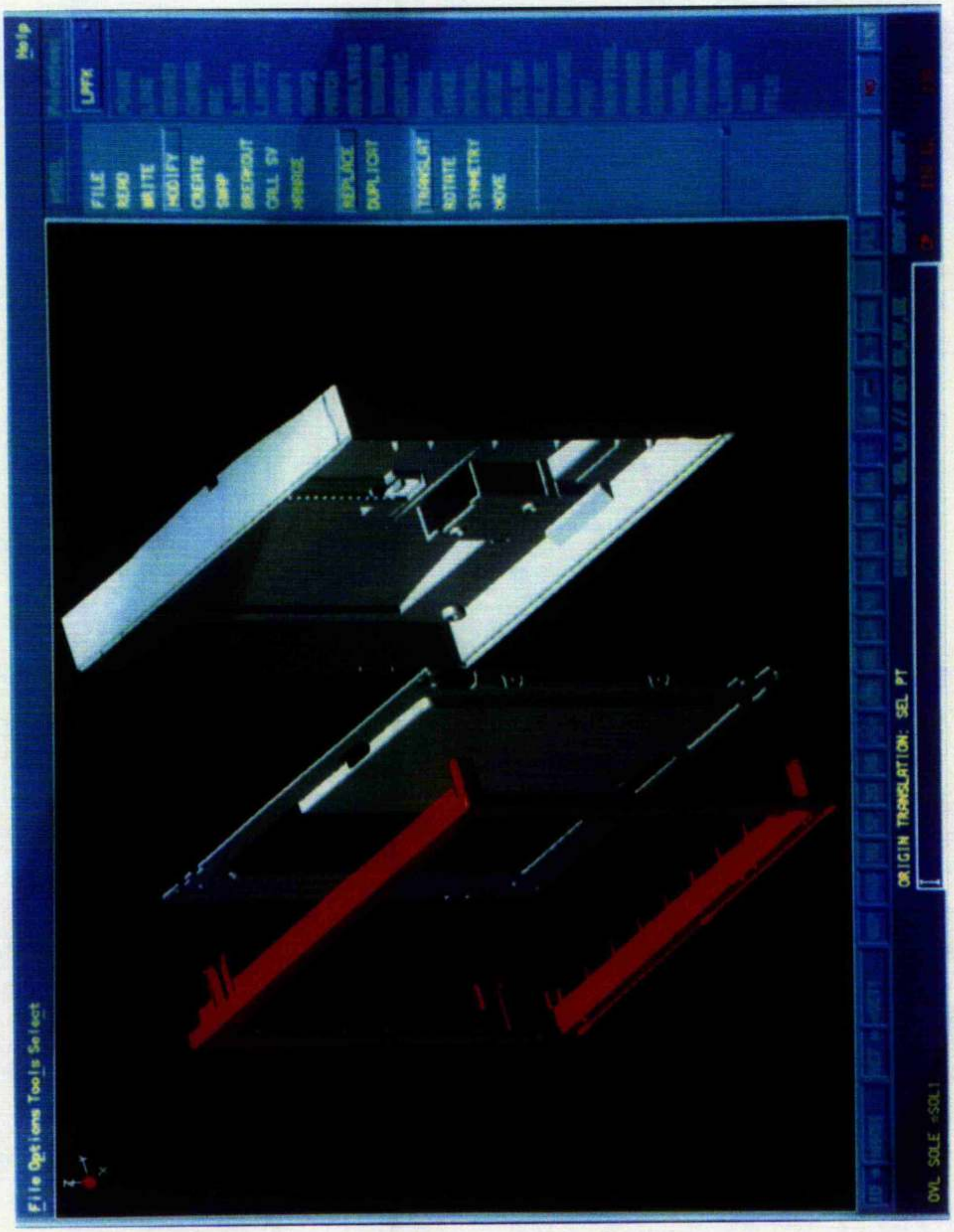


Plate 10: Finalised design of oil seal housing and associated components



ORIGIN TRANSLATION: SEL PT

OVL SOLE =SOL1

Plate 11: IBM 9507 Display monitor assembly

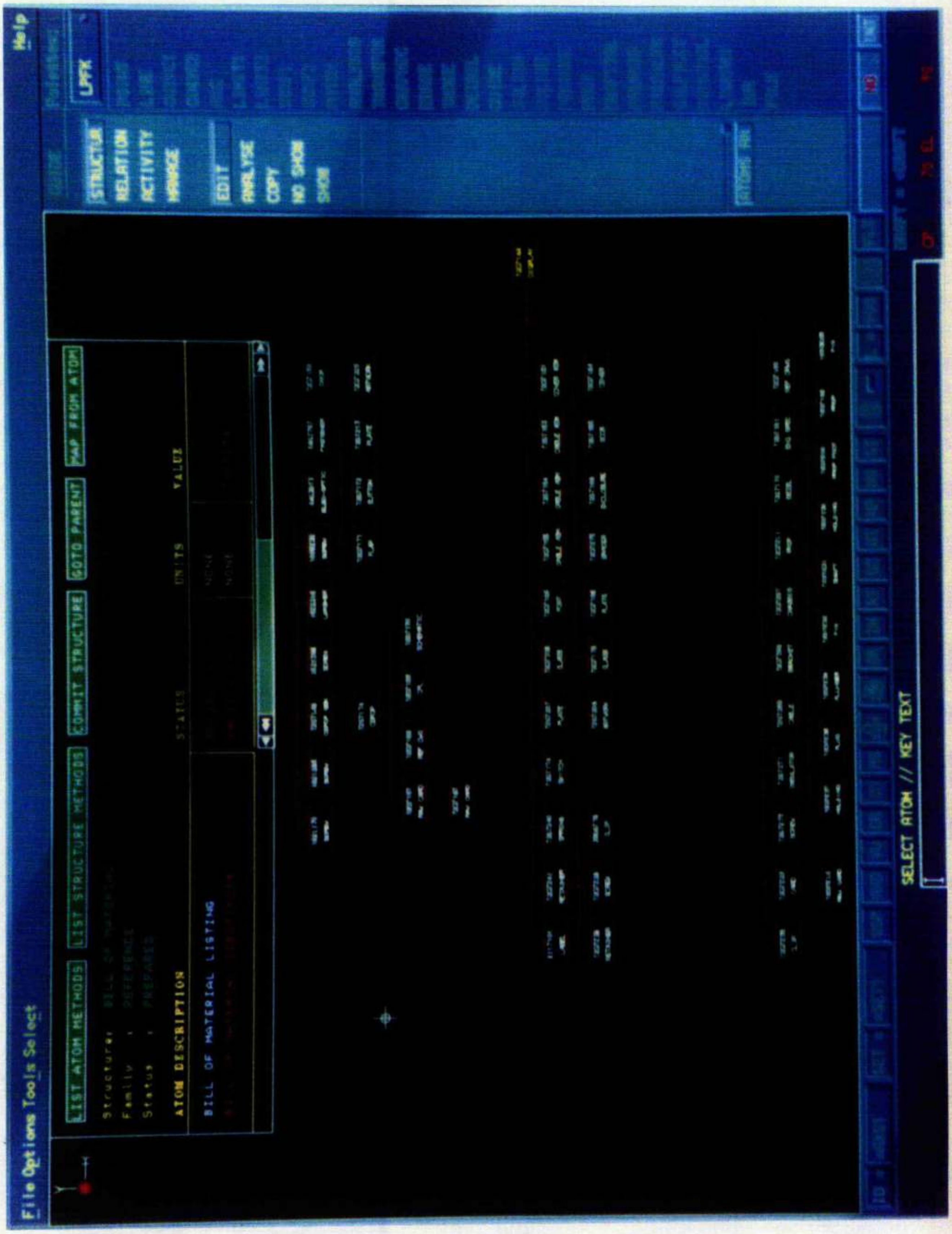
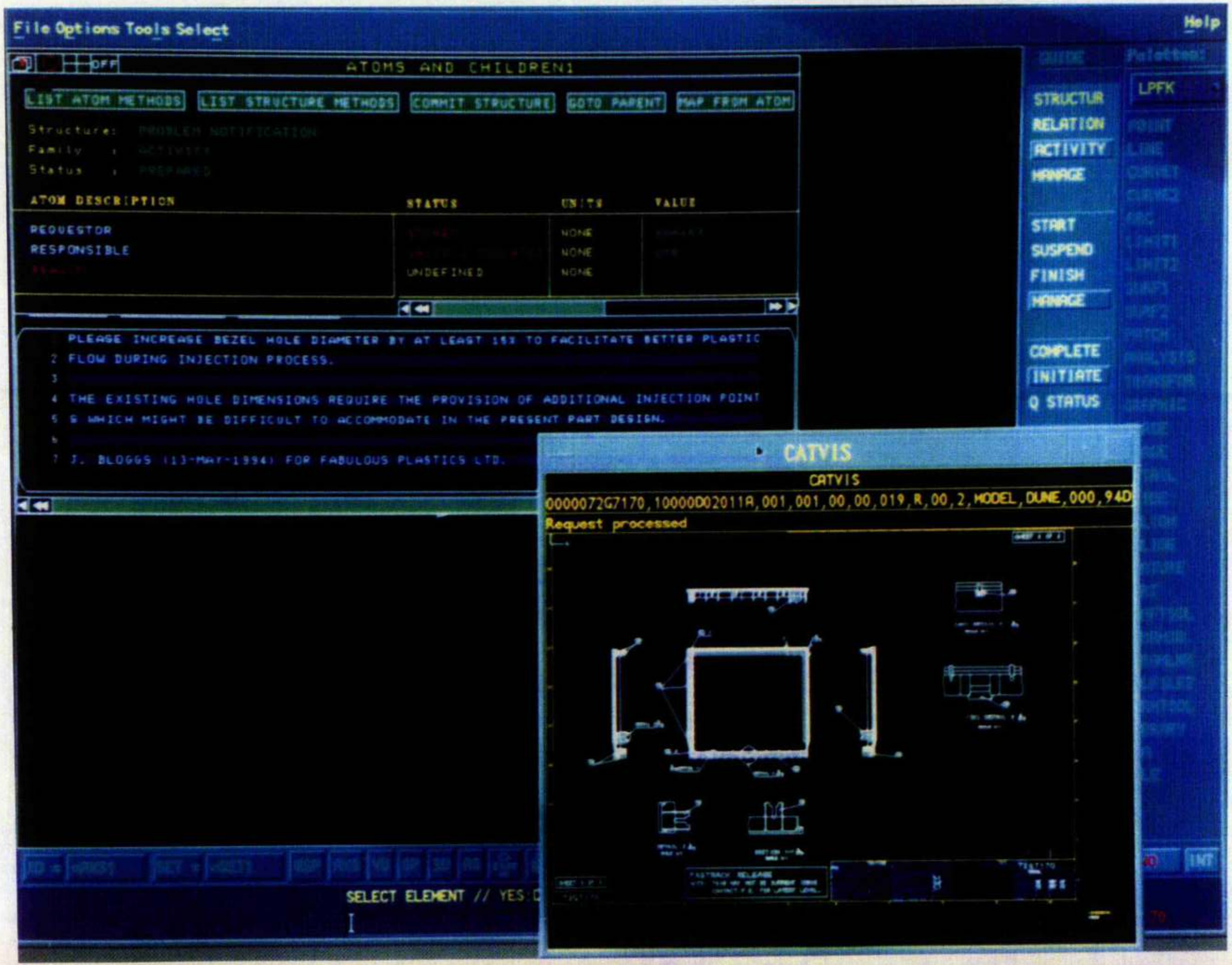


Plate 12: IBM 9507 Display monitor Bill of Material



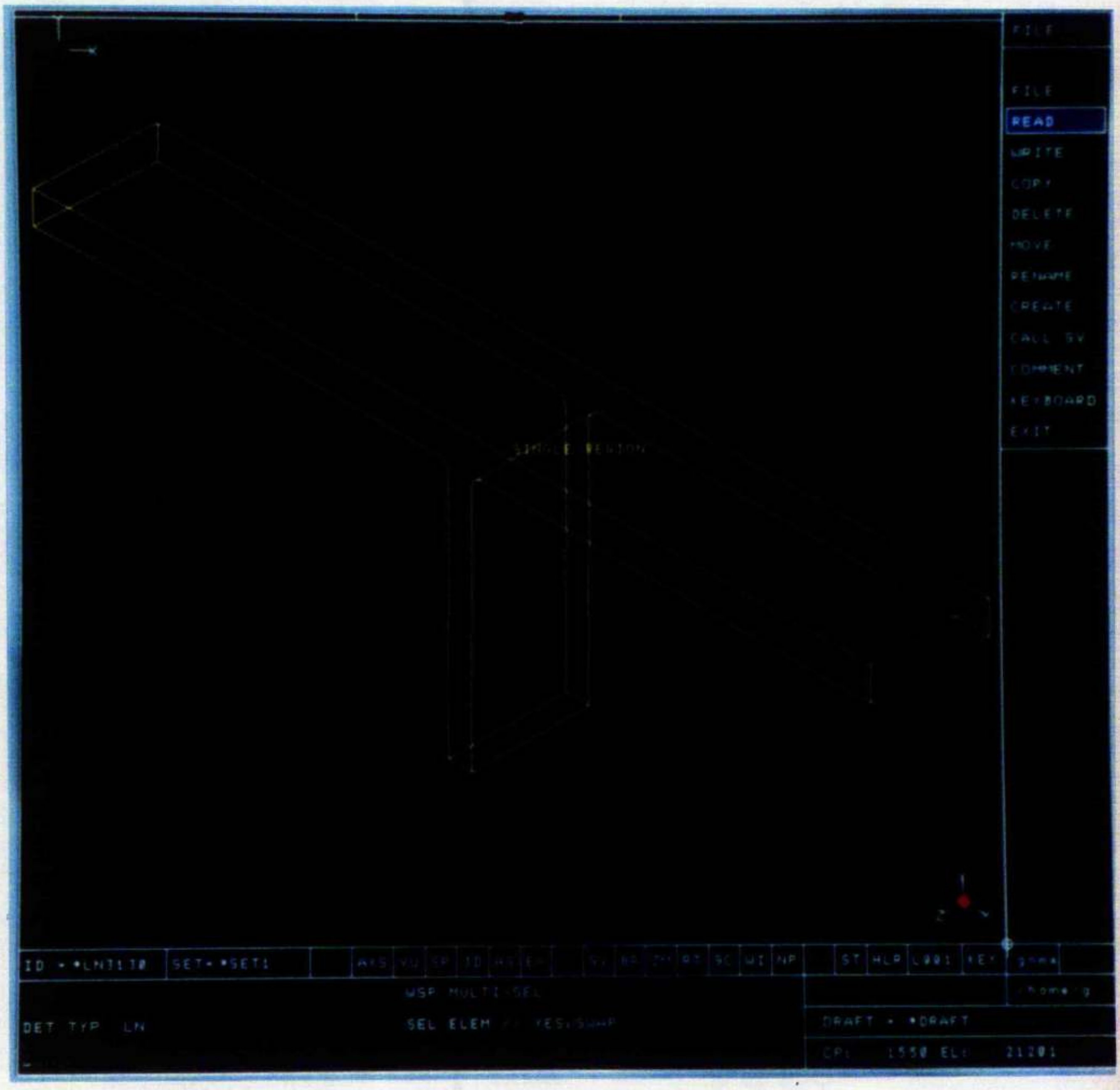
```
Structure:  PROBLEM NOTIFICATION
Family   :  ACTIVITY
Status   :  PREPARED
```

ATOM DESCRIPTION	STATUS	UNITS	VALUE
REQUESTOR	[illegible]	NONE	[illegible]
RESPONSIBLE	[illegible]	NONE	[illegible]
[illegible]	UNDEFINED	NONE	

```
PLEASE INCREASE BEZEL HOLE DIAMETER BY AT LEAST 15% TO FACILITATE BETTER PLASTIC
FLOW DURING INJECTION PROCESS.

THE EXISTING HOLE DIMENSIONS REQUIRE THE PROVISION OF ADDITIONAL INJECTION POINT
S WHICH MIGHT BE DIFFICULT TO ACCOMMODATE IN THE PRESENT PART DESIGN.

J. BLOGGS (13-MAY-1994) FOR FABULOUS PLASTICS LTD.
```

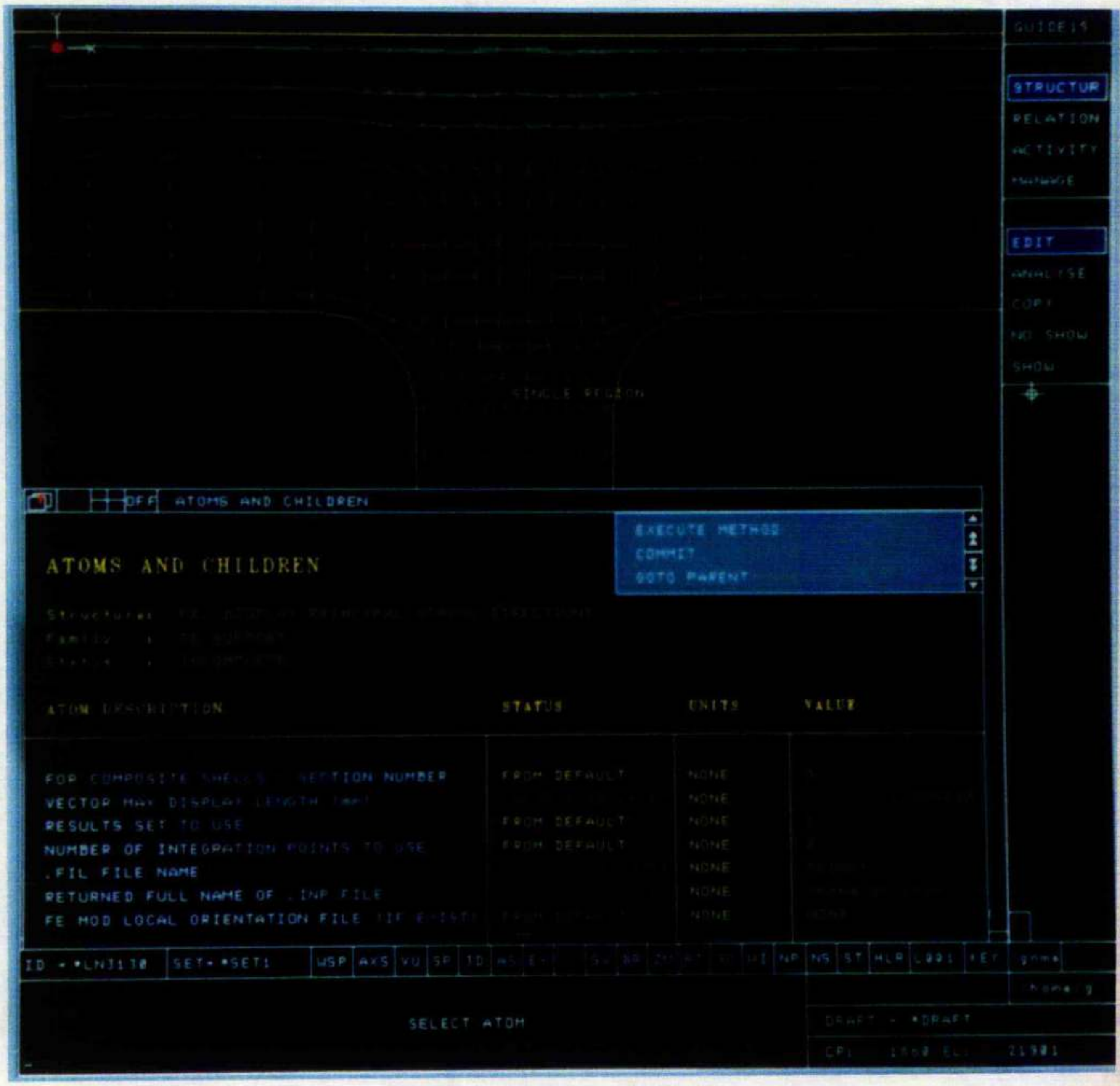
```
CATVIS
0000072G7170,10000D02011A,001,001,00,00,019,R,00,2,MODEL,DUNE,000,94D
Request processed
```

Plate 13: Manufacturing problem notification sent by a vendor to the product engineer



```
ID = *LN3110    SET= *SET1
                          WSP MULTI-SEL
DET TYP  LN               SEL ELEM   YES/SWAP        DRAFT = *DRAFT
                                                     CPU  1550 EL:  21201
```

Plate 14: Architectural model of single region T-Beam sub-element



ATOMS AND CHILDREN

Structure : [illegible]
Family : [illegible]
Status : [illegible]

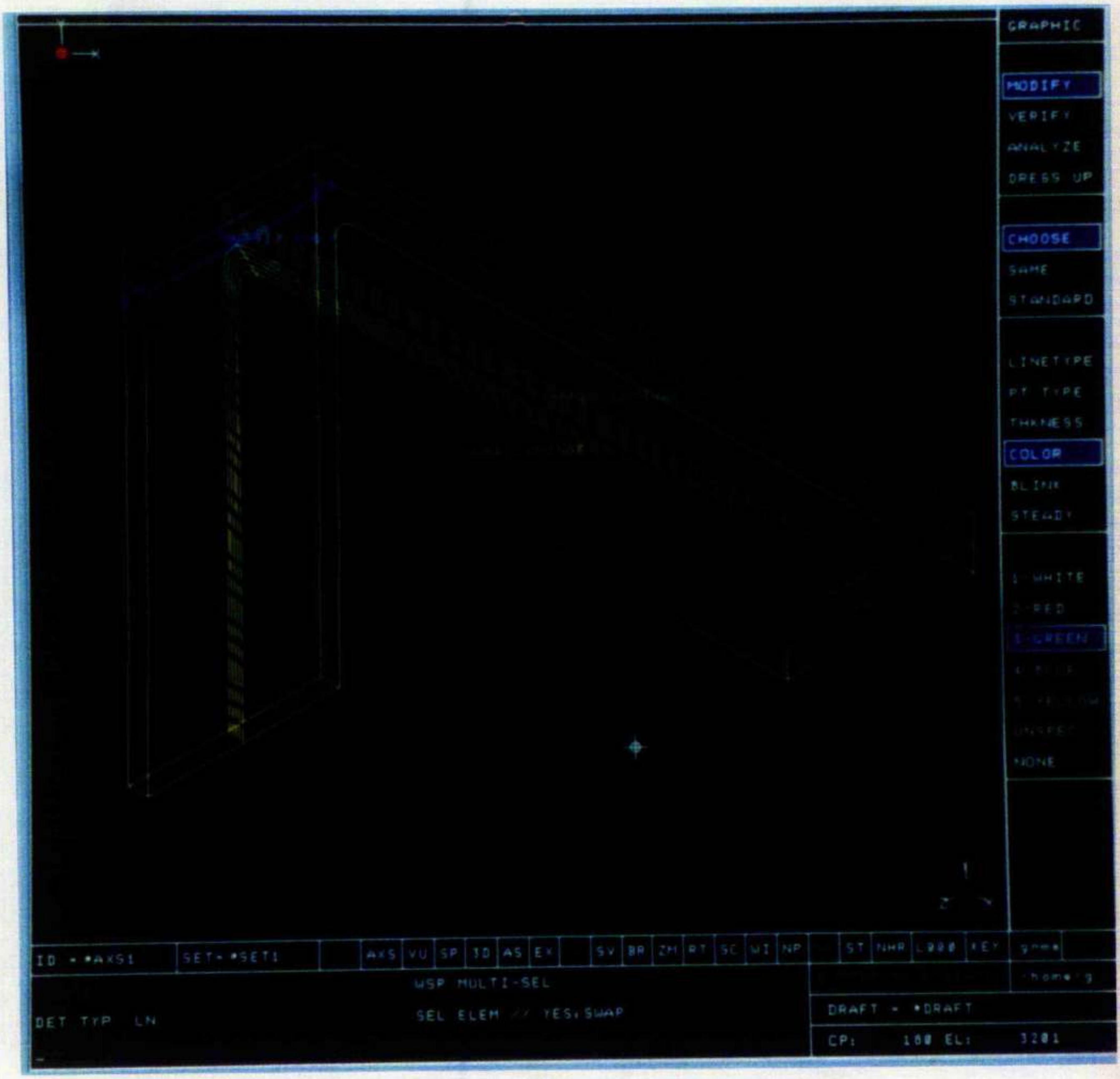
ATOM DESCRIPTION	STATUS	UNITS	VALUE
FOR COMPOSITE SHELLS - SECTION NUMBER	FROM DEFAULT	NONE	[illegible]
VECTOR MAX DISPLAY LENGTH (MM)	[illegible]	NONE	[illegible]
RESULTS SET TO USE	FROM DEFAULT	NONE	[illegible]
NUMBER OF INTEGRATION POINTS TO USE	FROM DEFAULT	NONE	[illegible]
.FIL FILE NAME	[illegible]	NONE	[illegible]
RETURNED FULL NAME OF .INP FILE		NONE	[illegible]
FE MOD LOCAL ORIENTATION FILE (IF EXISTS)	[illegible]	NONE	NONE

SELECT ATOM

DRAFT - *DRAFT

CPU 1560 ELA 21901

Plate 15: Principal stress directions imported from finite element analysis using a GUIDE method



WSP MULTI-SEL

SEL ELEM // YES;SWAP

Plate 16: Architectural model of a 3 region T-Beam sub-element

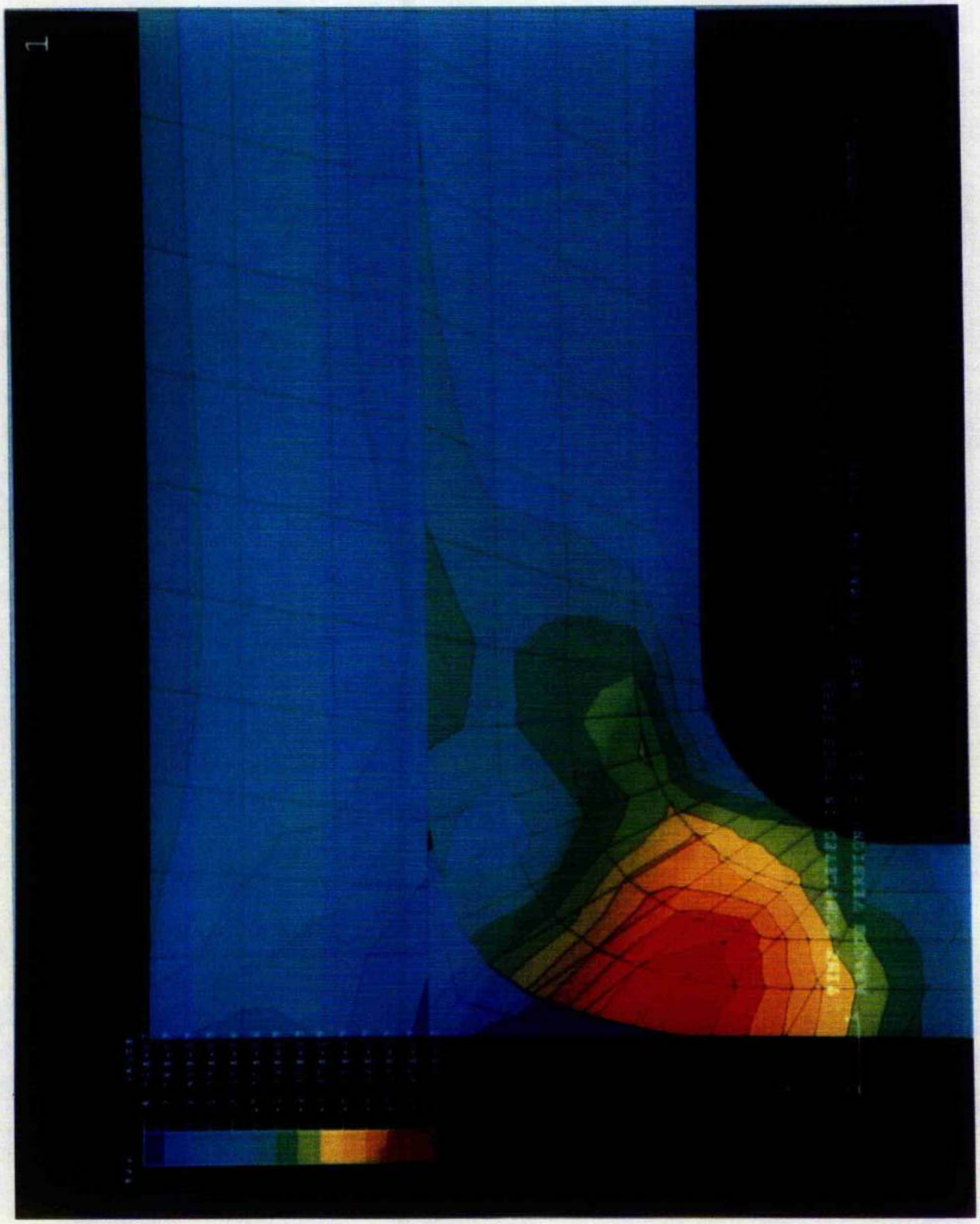
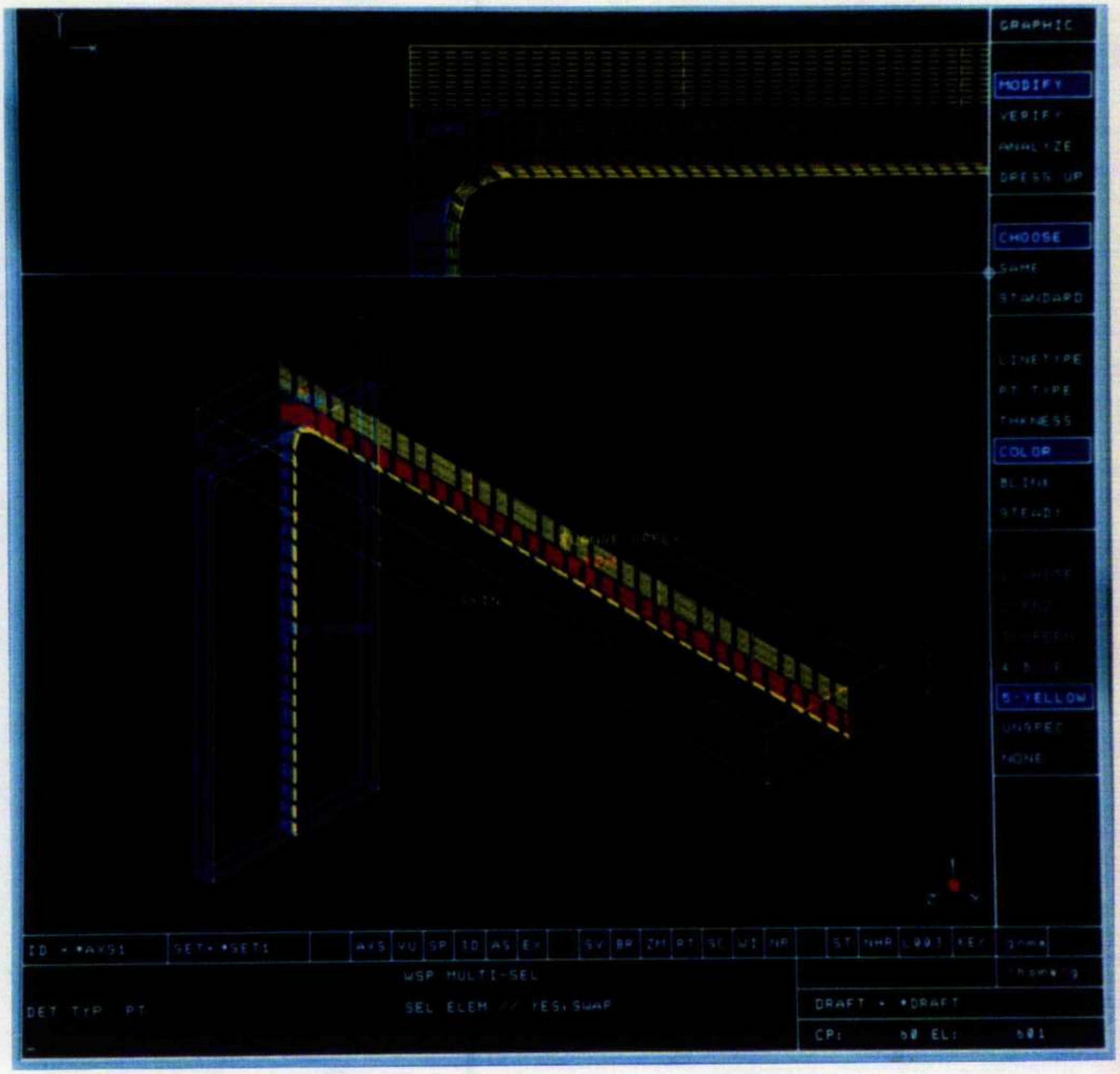


Plate 17: Contour plot of inter-laminar direct stresses for 3 region T-Beam sub-element



ID - *AXS1
SET - *SET1
WSP MULTI-SEL
DET TYP PT
SEL ELEM // YES, SWAP
DRAFT - *DRAFT
CP: 60 EL: 601

Plate 18: Architectural model of a 5 region T-Beam sub-element

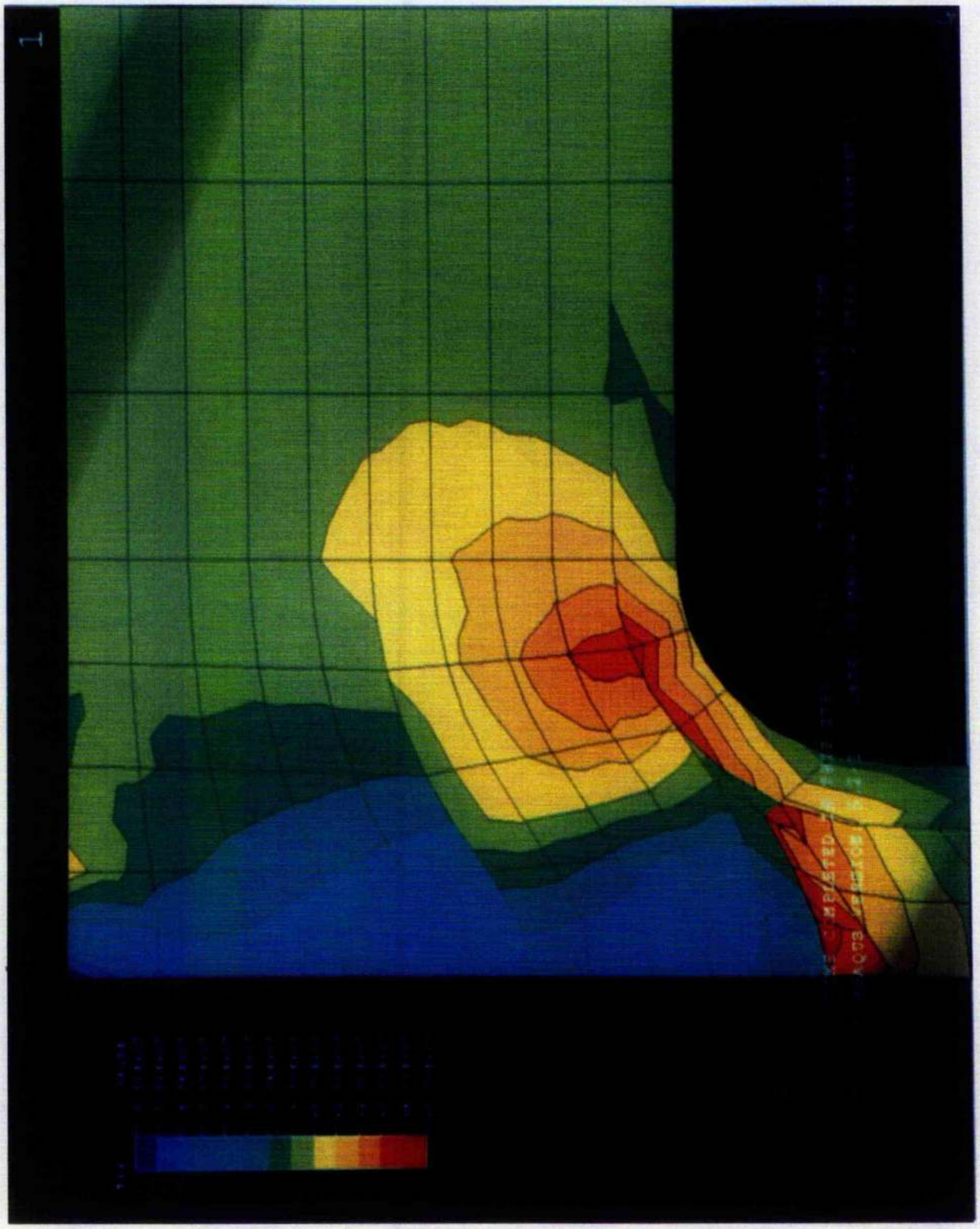
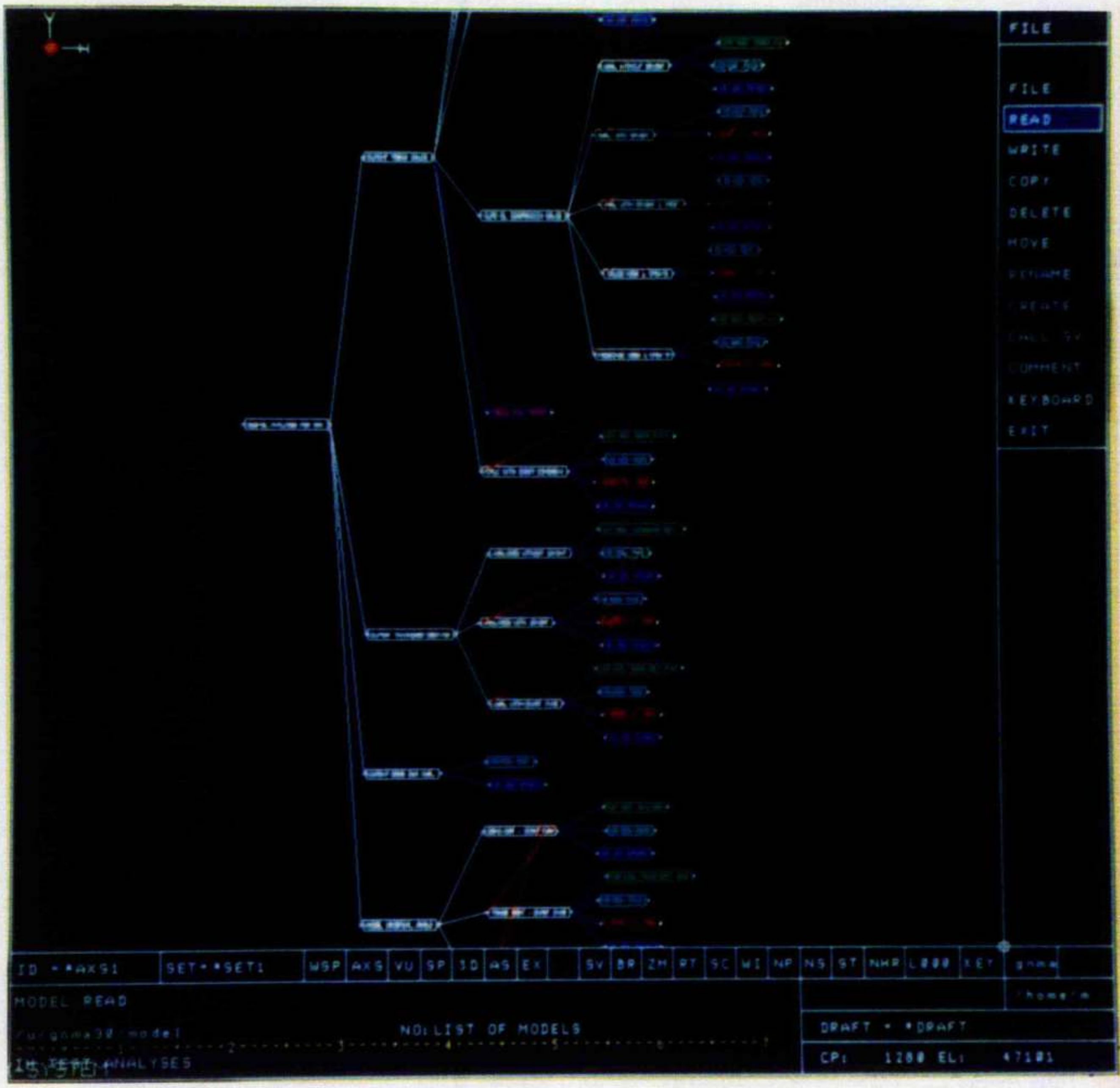
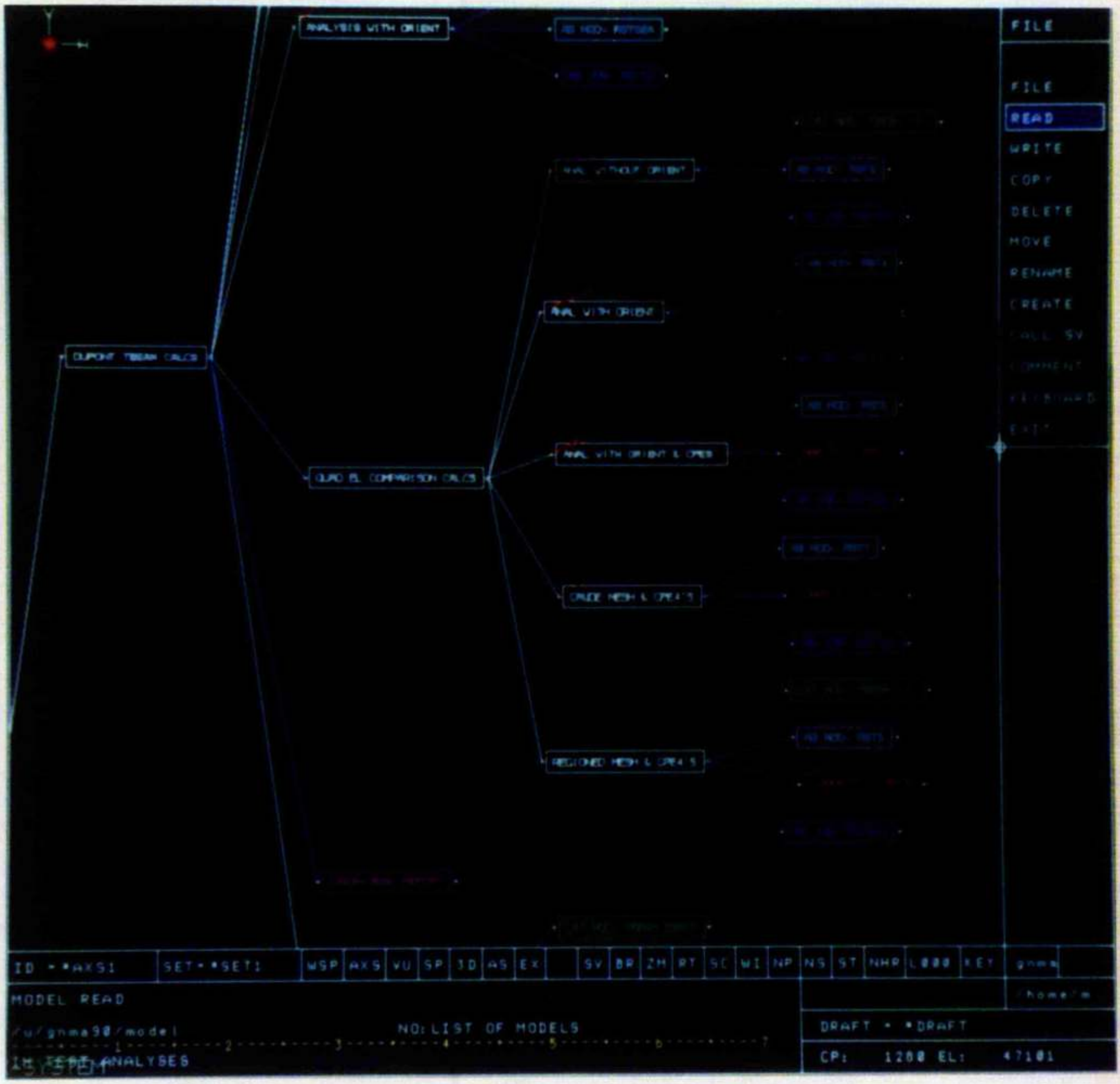


Plate 19: Contour plot of inter-laminar direct stresses for 5 region T-Beam sub-element



```
ID = *AXS1      SET = *SET1
MODEL READ
/u/gnma30/model              NO: LIST OF MODELS
IN TEXT ANALYSES
DRAFT = *DRAFT
CP:   1200 EL:   47101
```

Plate 20: Design record (post processed) of T-Beam sub-element design activity



```
MODEL READ
/u/gnma90/model                    NO:LIST OF MODELS
IM TEST ANALYSES
DRAFT - *DRAFT
CP:    1200 EL:    47101
```

Plate 21: Detail of T-Beam sub-element design sub-activity