MARCH 1987 VOL. 10 NO. 1

a quarterly bulletin of the Computer Society of the IEEE technical committee on

Database Engineering

CONTENTS

SPECIAL ISSUE ON INTEGRATED SOFTWARE ENGINEERING SYSTEMS AND DATABASE REQUIREMENTS

Database Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Database Engineering. Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unrefereed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Database Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in the Database Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. A member of the IEEE Computer Society may join the TC as a full member. A non-member of the Computer Society may join as a participating member, with approval from at least one officer of the TC. Both full members and participating members of the TC are entitled to receive the quarterly bulletin of the TC free of charge, until further notice.

Letter from the Editor

This issue is on Integrated Software Engineering Systems and Database Requirements. The increasing complexity in software systems and increasing cost in developing and maintaining them have motivated a number of major research and development projects, both in the industrial research laboratories and the universities, to find ways to significantly enhance software development productivity and quality of software. One approach is to develop a system which will integrate software engineering tools under a friendly user interface and around a shared database of software components, information about them, and knowledge about software design process. In an attempt to foster a better understanding of the role of a database system in such an integrated software engineering environment, I invited a fairly diverse mix of papers for this issue. Three (Lehman, Akima, and Dittrich, et al) of the seven papers included are from outside of the United States; five (Lehman, Akima, Biggerstaff, et al, Welch and Konrad, and Delisle and Schwartz) provide industrial perspectives; four papers (Lehman, Habermann, Dittrich, et al, and Delisle and Schwartz) discuss operational systems; and five (Akima, Biggerstaff, et al, Dittrich, et al, Welch and Konrad, and Delisle and Schwartz) attempt to bring out database requirements in software engineering environments.

I would like to thank the authors who contributed their valuable time to this issue. I enjoyed working with them. I hope the readers will find this issue informative and stimulating.

Mike Carey is presently organizing the June issue on extensible database systems. Sunil Sarin will edit the September issue on federated database systems. C. Mohan is tentatively scheduled to do the December issue on bridging database theory and practice.

Won Kim

Won Kim

Editor-in-Chief
March, 1987

MODEL BASED APPROACH TO IPSE ARCHITECTURE AND DESIGN

- The IST ISTAR Project as an Instantiation -

MM Lehman
Department of Computing
Imperial College of Science and Technology
London SW7 2BZ

1 Introduction - The Nature of Software

Software (by definition) contains no physical components. A program is constructed exclusively of *linguistic* elements, character strings, words, phrases, sentences, graphical elements, expressed in some formal or natural *language*. These are associated to produce representations that model concepts and properties of the problem to be solved by a computer system, of the domains or environments in which the problem and its solution are defined and of the solution which is to be implemented and used. Even where an end product takes a physical form such as a VLSI chip, its functional properties will have been defined by a formal text (program).

Since all that is being produced during the development of software is purely linguistic the development *process* is also non-physical. Definition, design, realization and continuing adaptation of a computing application and of the system that implements it is obtained by a sequence of *interpretation, manipulation* and *transformation steps* applied to a textual representation to *refine* it [WIR71]. The process starts with initial verbalization of an abstract (mental) concept and proceeds down to the production of executable code and its documentation.

Because of the non-physical nature of software, the software development *process* and its *products* are not disciplined, limited or controlled by natural laws or physical constraints. As a multi-person activity extensive in time, discipline *must* be applied to that process if chaos is to be avoided. On-time delivery of products accurately and reliably satisfying the needs for which the process was undertaken can only be achieved through *precise* representation of appropriate concepts and their rigorous application. Linguistic and process constraints replace the controlling factors that dominate other design and development processes. It is, therefore, vital for concepts, languages, methods and procedures used in software development to be precisely formulated and specified. If this is not done the process and its products will continue to suffer all of the problems that have plagued them for the last three decades.

The dominant role of *evolution* in software and systems development must also be accepted [LEH85]. Evolution arises from a variety of related and interlinked factors [LEH86b]. It is experienced as iteration and backtracking in the development process and as release sequences of its products. Control and direction of resultant pressures for change is a key concern of software technology. Mastery of the evolution phenomenon is crucial to management of each stage of the software development proces. Process management must be viewed as the direction and control of change.

Evolution is not, primarily, a consequence of ignorance, incompetence, sloppy thinking or incomplete planning. It is intrinsic to the very being of computing applications and, therefore, software. Defining, developing, installing and using a computerised application clarifies the need, the problems and means for their solution. All these activities raise the sights and ambitions of both users and implementers. Installation and exploitation also changes the application domain and, therefore, both the application and perception of it. And coupling between user and system is tight. Inadequacies, imperfections, mismatches with a changing environment are all personally experienced and rapidly become intolerable. Pressure for change mounts and cannot be resisted for long, particularly in the face of ever present competition. System attributes, its components and the overall system all evolve.

As mankind placés ever more reliance on computers, incorrect, unreliable or unavailable software will represent an increasing threat to society. Improvement of the software development process [SPW84, 86, 87, LEH87c], significant advances in its reliability and timeliness, must be accepted as an urgent societal priority. The facts summarised above typify the environment within and for which the technology must be advanced. It is these perceptions and facts that underlie much of the current work in formal software development methods [JON86, MAI87], in data modelling, in studies of the software development process [as above], in software evolution [LEH85] and in the development of Integrated Project Support Environments (IPSEs) and their tools [STE85, LEH85b, 87, PDS87].

As observed above, disciplining the products of software systems development and evolution requires a substitute for the natural laws and physical constraints that operate in other engineering fields. Through their semantics and grammatical rules natural languages provide a limited degree of control. Unfortunately, and as so succinctly expressed by Koestler*, their use also limits creativity. Moreover, the meaning of natural language representations in the developing system will be obscured by the use of primitives that convey meaning by association, *hidden persuaders*. in Koestler's words. Furthermore, natural language is neither precise nor unambiguous but both are vital to success of the development process. Finally, the basic operation of transformation by which progress is achieved must also be precise and correct. It should be calculable, not based on human interpretation and manipulation. The use of formal languages, whose primitives are logical rather than phenomonological and with operations that are precisely defined, largely solves all these problems.

Formalisation is, however, not sufficient [LEH87b]. Humans play an essential, creative, role in software development. In general, development can only be achieved through the creative activity and collaboration of many people over a long period. In this they continuously require full and ready access to information involved in earlier activity. It is, therefore, necessary to capture and make available full records of all information and data involved in the development process. Furthermore, the process by which process steps are defined, sequenced and executed, the management of human and other resources, controlled association and integration of the components developed over the lifetime of the product in its many instantiations, are also crucial to success. Process design and control is essential to achieve quality products since the quality of these will be no better than that of the process producing them. Thus there emerges the concept of process models [LEH84, 85, SPW84, 86, 87]. These provide the starting point for achievement of the goals of software engineering and hence to the concept of IPSEs as an encapsulation of advanced systems and software technology. IPSEs can be the realization and embodiment of software technology and an instrument for its transfer and introduction into industrial practice [LEH86]. Achievement, depends heavily on their design.

In what has been said above *software* has not been defined. The relevance of the observations is not confined to code. It applies equally to *problem* and *requirements statements, specifications, databases, documentation* etc. All contain information vital to the success of the application over its lifetime. All must be unambiguous, precise and correct. All must be preserved to be readily accessible in all their variations and versions if satisfactory system development, build and evolution is to be achieved and maintained. The crucial property of a software system is its *dynamic* correctness as the world changes.

The remainder of this paper concentrates on discussion of the development process, on relevant models and on IPSE architectural concepts. The IST ISTAR environment, developed at Imperial Software Technology Ltd (IST) in collaboration with British Telecom (BT), is used as a vehicle for the discussion. This system is particularly appropriate as the basis for such a discussion. Its design concepts were directly derived from a specific view of the programming process [LEH84]. The process model that resulted from this collaboration provided the initial conceptual basis and framework for ISTAR.

2 ISTAR Design Goals

The principal goals of the ISTAR project would appear to apply equally to almost all IPSE development. They included the provision of full support for software development teams rather than

* *The prejudices and impurities which have become incorporated into the verbal concepts of a given 'universe of discourse' cannot be undone by any amount of discourse within the frame of reference of that universe. The rules of the game, however absurd, cannot be altered by playing that game. Among all forms of mentation, verbal thinking is the most articulate, the most complex, and the most vulnerable to infectious diseases. It is liable to absorb whispered suggestions, and to incorporate them as hidden persuaders into the code. Language can become a screen which stands between the thinker and reality. This is the reason why true creativity often starts where language ends'* [KOE64].

just for individual programmers; such teams being, in general, distributed over distinct geographical or organisational entities. Hence the target system, and in particular its databases, should be distributable. Moreover, support had to be effective over the full life cycle of each product or, more strictly, of each application. Since equipment must inevitably differ between locations and over time, the IPSE had also to be simple to port. Moreover, to ease transition to new development processes and to conserve resources, the target system should exploit existing methods and tools wherever reasonable.

More generally, since IPSE development must inevitably be costly, the resultant system should be suitable for use by a wide variety of users to reduce individual usage costs. Even if initially tailored to current or anticipated needs and desires of a specific client an IPSE should be readily adaptable to incorporate methods and tools of interest to other groups or organisations. Moreover, software technology is advancing rapidly and there is no universal or best method. Any process or environment locked into one particular development method or into the state of the art at some specific time would rapidly become obsolete. At the very least, new methods and tools relevant to the interests and activities of potential users must be integratable as they become recognised and available. Finally, organisations and their workforce cannot change their way of working or adopt new methods or tools overnight. An IPSE, when first installed, should make possible the continued use of existing methods and tools, with new ones introduced as the workforce is trained in their use. An IPSE should, therefore, be flexible and evolvable accommodating, at any one time, a range of approaches and methods; the design and construction must be open ended so that both process and IPSE evolution will be feasible as technology advances and as resources become available.

3 The Process of IPSE Development

An IPSE must provide support over the entire life-span of an application in a *coherent* and *integrated* fashion, not just for initial development. Moreover, programming, however broadly defined, is only one of many activities needed for the creation, maintenance and continuing evolution of a software based product. Specification and design of a software product; algorithm selection (technical development); planning, control and evaluation (system management); data capture, storage and retrieval; component and system build management; work planning; resource management; all are important activities. They are all intrinsically related and interdependant. When concern with an artifact ranges over its entire lifetime it is the retained process information that links different activities.

An adequate but parsimonious process *model* involving a minimum number of concepts and paradigms is key to achieving integration. It must also be complete, spanning technical and management activities. This model must be complemented by a project model [STE85] which defines organisational and activity structure for the management and conduct of the process. In the case of ISTAR, the contractual (section 5) and process (section 4) models played a semenal role in the design.

Where the specification and design of an IPSE is developed from specific models, the system that results cannot be neutral with respect to processes and project organisations supported. Conversely, it is always beneficial for an IPSE end-user to adopt an approach compatible with process and project concepts that have influenced its design. Given sufficiently general underlying models, effective IPSE exploitation should be achievable in most circumstances. Nevertheless, the need to provide lifetime support will inevitably lead to an IPSE that is process sensitive. Anyone expecting to use an integrated environment effectively while following an ad hoc operational process is bound to be disappointed.

Models providing the basis of an IPSE architecture and design may, in turn, be used to guide selection or development of methods to be supported for use during the various stages of system development. Existing methods should be adopted whenever possible, by adaptation if need be. In the absence of already available methods one may have to support new development whilst temporarily using ad hoc approaches to the satisfaction of immediate needs.

Once process and project models have been defined and methods selected, one may develop relevant data models. Such models must reflect the structure of process and product information; the nature of specifications and constraints defining project goals and product properties; information flow; projected relationships between components and between activities; accountability during process execution and in the project organisation; the need for management data and so on.

Identification of (classes of) methods and associated data models, in turn, facilitates definition of tool attributes and the specification of tools, technical and management, to achieve them. Interplay between the emerging data models and tool set will provide information for their refinement. The full set of models may then be applied to guide the development of a *framework* IPSE subsystem that will eventually house selected tools and tie them into an integrated whole. This subsystem will normally include global process facilities such as an information repository and database, interconnection and communication facilities, work station control, tool building and packaging tools and so on. The approach to IPSE development described is illustrated by figure 1.

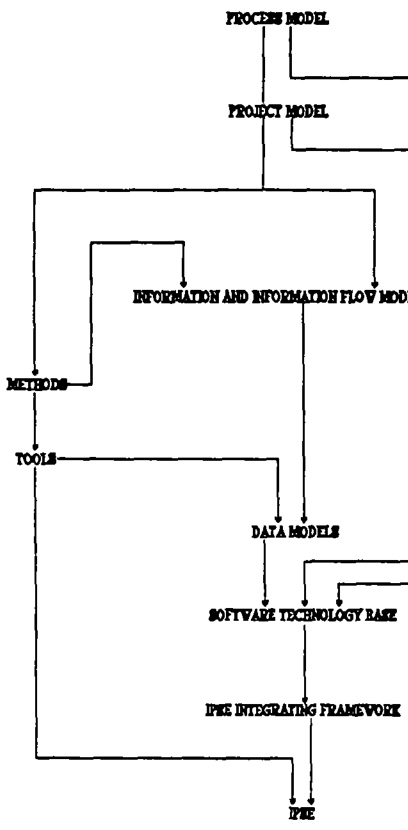


Figure 1 The IPSE Development Process

4 The Process of Program Development

4.1 Program Development

The term *Software Development* is generally interpreted as referring to creation of a program that realises a new application. In fact, it should include adaptation of an existing program; activities arising from changes in the operational environment, changed perceptions of that environment, revised objectives, new technology. There is no need to differentiate between these various situations. Indeed there is every reason to treat them as variations on a common theme; *the further development of a program or system from its current state (which may be empty) to a new state.*

At a high level of abstraction, the process whereby this is achieved may be described as *the transformation of an application concept into an operational system or of a change to an existing concept into a change to an existing system.* The transformation cannot, in general, be attained in a single step. It is achieved as a sequence of transformations, the many steps yielding representations R_1, R_2, ... R_i, ... TS. These span the space from initial verbalization of the application concept, AC, to realization, TS, of the target system.

4.2 The Program Step

In the classical process model, the waterfall for example [BOE76], individual steps are seen as very specific. They are normally referred too descriptively by terms such as *specification, coding, testing* and so on. Unfortunately, this model is not consistent with the needs of process integration, coherence, evolution and so on. Nor will it permit development of the global, parsimonious, models that scientifically based technology demands.

One such model has been described in detail elsewhere [LEH84]. It is based on the observation that despite the fact that languages and methods used for step activities at different stages will vary, they satisfy a common paradigm expressed in terms of a base language, a representation in that language, a target language, a transformation-derived representation in the target language and obligations of verification and validation. The paradigm also includes provision for decomposition through partitioning of a base representation and for the recursive introduction of intermediate steps. In the *canonical step* as discussed in the source reference, definition (selection) of the target linguistic system is the creative act in each step. This is the decision that sets the direction and degree of refinement desired in that step. Its creation permits derivation of the relationship between base and target languages. Other forms of creativity can, however, also be visualized.

This paradigm addresses only what is to be achieved, not how. At the highest level of abstraction the objective of each step is *refinement*, progressing the dual processes of *abstraction* from the application universe and *reification* to the system universe. It applies equally to early stages of concept development which might well be conducted in non-formal linguistic systems and to design, that is stages which should be formalised. Conceptually the paradigm implies a top down analytic approach. At each and every step the process (as distinct from system) designer will be able to identify methods and tools that best fit the specified technical and managerial objectives and constraints. He must only be certain how to fit them into the overall process. The model does not restrict the selection of available methods and tools. It merely guides that selection.

4.3 Verification

Execution of each transformation step includes an obligation for *verification*. This must demonstrate that the target representation is self-consistent and a consequence of the base representation, logical and extra-logical axioms in the base language, the specified inference rules and the derived relationship between base and target languages. Verification is automatically achieved when a verified compiler or other transformation mechanism is used. Alternatively, a sub-step may have been created using the approaches of constructive correctness. Or a correctness proof may have been generated, perhaps using a theorem prover. Whatever the approach, the adoption of formal development methods will simplify the provision of associated verification techniques and tools to support them.

4.4 Validation

Validation is not, in general, a precise, calculable act like verification but involves judgement. It seeks to confirm that the intent of the original application concept has been correctly interpreted and preserved during the development process. This judgement will, of course, be made in the light of improved understanding gained while following that process. Beauty lies in the eyes of the beholder. His judgement at any stage must include a view of the eventual properties of the operational system. He will attempt to satisfy himself that, as far as can be determined from current perspectives, the target system when reached will satisfy the need of the application at the time when the program is invoked. Such judgement should be made at each step in terms of current observations and knowledge of the further process. It will seek to demonstrate that, in terms of the linguistic level in which it is expressed, the derived representation displays or reflects the desired properties of the operational system. It must also search for implied undesirable properties of the final system. Finally, the representation should be shown to represent progress towards achievement of the target system; it must be suitable as a base representation for further development. Such validation may be based on a comparison of the representation with an image in the validator's mind. Alternatively, the image may be explicitly represented in some more understandable form. However represented, the image is a projection, a viewpoint, a *domain sub-model* of the application concept.

Semantic properties of the representation being validated must be evident to those involved in the validation process. To obtain maximum benefit from validation it may be necessary to recast the current representation into a form in which interpretation is simplified. Great care must, however, be exercised that such interpretation neither adds nor removes properties. Disciplined methods and tools to support this and other aspects of validation at each process step are most desirable [BAR84].

Validation can be interpreted in two ways. One may regard it as a semantic interpretation of the current representation compared mentally with and judged against one's expectations of and from the target system. If the representation is executable one may be able to base this judgement, in part at least, on the results of execution [BAR84] treating it as a prototype. In addition, when the representation is formal, the systematic application of, for example, theorem proving techniques allows one to examine consequences of the representations to ensure that such consequences correspond to or imply desired system properties. Equally, one will wish to ensure that there are no undesirable consequences. In practice one can, of course, only state that no undesirable properties have been found.

An apparently alternative view sees the intent of the original application concept expressed in some domain model. For each process step the current representation must be compared with appropriate portions of the domain model. One must seek correspondence, in some sense, between interpretations of the representation and documented expectations, the domain sub-model. That is, validation seeks to assess the relation of the developing system to some *preconceived expression* of the objective. Consider, for example, a well defined problem where the properties of a solution have been be precisely documented, an earlier program, for example. Being precise, however, the domain model represents R_1. The situation reverts to that of the previous viewpoint with the concept of a domain model reducing to an image in the human mind. This may be important in the process but lies outside the reach of the software engineer.

In summary, validation has two aspects. *Behavioural validation* establishes that each representation is acceptable as a satisfactory model of the application concept in its operational domain and that a target system developed from it is likely to present satisfactory attributes and behaviour. *Process Validation* must demonstrate that the current representation represents progress towards achieving the goal TS. It constitutes a useful base from which the development and refinement *process* can proceed.

4.5 Backtracking

From time to time the results of verification or validation procedures will indicate that an error has been made or a less than fully satisfactory result achieved. Perception or expression of the problem, of methods for its solution, of assumptions or of relationships may have been incorrect. Alternatively, they may have been incomplete or unwise. An error may have been made in transformation. Whatever the cause, it becomes necessary to backtrack, at least, to that representation in which the error or omission occurred, perhaps to the linguistic level at which related notions can first be expressed. The appropriate changes or additions must then be made where appropriate and propagated forward. The target representation is achieved by iteration around the transformation path.

4.6 Overall Process Structure

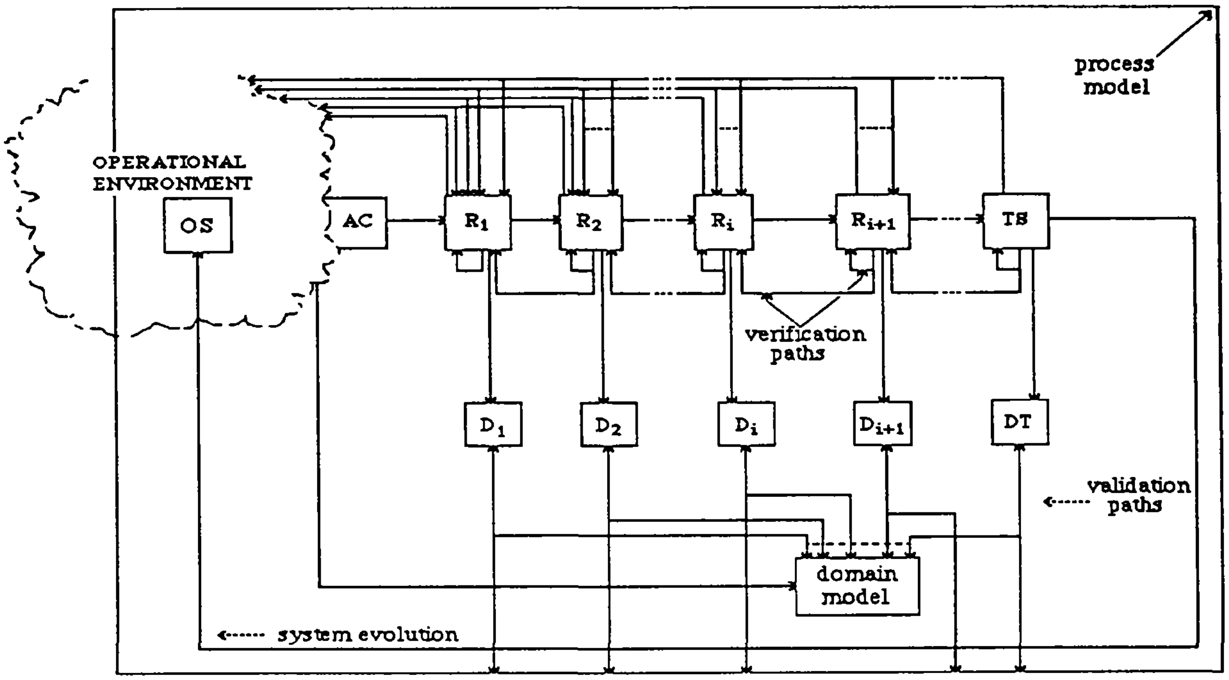
As a consequence of backtracking one obtains a new sequence of representations. The linear sequence of representations implied in the foregoing discussion is, in reality, multi-dimensional. Each act of decomposition, for example, initiates a new sequence. Each backtracking path overlays an earlier sequence. The information repository that must retain process-generated information will have to preserve all generated and accepted representations once they have been released from control by an individual participant. When variation and evolution is fully taken into account, the conceptually simple model and its implementation in a database becomes a complex structure. This fact points clearly to the need for automated configuration control. This must control storage, association and selection of information and representations for future reference.

4.7 Feedback Driven Evolution

Once the target system representation is attained and accepted, it can be installed and made operational. With that it becomes an integral part of the application environment, *a part of the application.*

Installation and operation physically changes the application domain. However well the impact of operation may have been foreseen, feedback via intelligent human operators, users and so on will make that foresight less than perfect. Operational experience will inevitably change the users' understanding of the problem. Hence it modifies their expectation of system properties. Equally, users will become frustrated with newly perceived inadequacies of the system. The application concept changes. Its initial verbalization must be modified or adapted and the development process repeated. In general, usage feedback is inherent in computer application development. It causes the intrinsic rate of system evolution to be much more rapid than that of other artificial systems [SIM69]. As observed above programs must undergo continuing evolution. Else they become ever more unsatisfactory as a consequence of increasing mismatch with an evolving operational environment.

A more detailed of the development process than has been possible here, and as illustrated in figure 2, is given elsewhere [LEH85c].



AC - Application Concept, D_i - Domain Sub-model, OS - Operational System, R_i - Representation TS - Target System

Figure 2 The ISTAR Process Model

The total process depicted in the figure involves people. These apply disciplined, preferably formal, methods which may be supported by tools. The implementation activity is organised as a project. Thus one must next consider how that project is defined and structured.

5 The Project Model

As observed in section 1, a multi-person effort spread over a period of time consists, by its very nature, of many sub-activities. The planning, generation, direction and control of such activities, individually and collectively, constitutes a major technical and managerial challenge whose successful pursuit may well determine the value of the immediate product and the success of follow on activities. The preliminary discussion has already recognised the critical nature of project organisation, its reflection on system structure. The process model must, therefore, be paralleled by a compatible project model.

It is vital to the success of a project that each sub-activity be defined in terms of the nature and characteristics of its inputs, outputs and operational environment. These must be agreed before related activity is initiated; as must any changes subsequently. When work is to be undertaken for a client by an implementer, the agreement must be recorded in some way, enshrined in a *contract* for example. More generally one may model a project and its activity as a structured set of contracts. This records, inter alia, the agreed view of the externally visible consequences and products of individual activities as well as the relationships and dependencies amongst them. Such agreement must inevitably change with time, leading to a dynamic structure, a growing family of contracts, *conceived, defined, active, completed, abandoned* and so on. That is, the structure will change as attributes of contracts are modified in response to external changes, operational experience or process experience; as activities are completed; as new activities are identified, agreed and initiated.

Full information retention is essential to any multi-contract activity, the more so when the product of that activity is to be long-lived and continuously evolving. It is, in general, necessary to capture, record, structure and catalogue most, if not all, information generated during development and to store and update the data that represents that information. Given the contract-based structure envisaged above, it is natural to organise information retention and the associated data storage on a *per contract* basis, with further separation distinguishing between data that is internal and private to the contract and data that must be selectively available to other project activity.

Effective and reliable communication channels are equally crucial to success in a many person activity. They play a critical role in the control of tasks, in conveying information required before initiation and during execution of each task, in the submission of reports on progress in each activity, in the transmission of textual products of the process, in liaison with the client or end user, in management of resources and so on. Once again, the individual contract provides a logical base for the location of communication channel terminals with contract dependencies indicating prime communication links.

In general, organisational structure provides a valuable guide to the appropriate location and interconnection of support for the above facilities. In any IPSE based on the models presented, the contract structure determines the relationship between activities, as determined by individual contract definitions. That same structure largely determines the structure of the information repository, the site of database elements and the location of formal communication interfaces.

The activity that identifies and specifies the set of contracts that define and control execution of a project may be a precursor to that project. It is itself, at least notionally, the subject of a contract. The latter must then include provision for establishing the relationship between contracts, a relationship that provides strong guidelines for design of the contract hierarchy. It is, however, more likely that the need for, at least some, contracts is recognised only as the project progresses. Individual contractual activity may, therefore, include definition, assignment or management of *sub-contracts*.

The core of any contract is its specification; a specification as precise as can be achieved at the time of definition. Notice that 'precise' does not necessarily mean detailed. In fact the process of programming should be viewed as a process which elucidates and formally expresses detail by successive refinement [WIR71]. Thus a contract could require the development of detail in a specified context or, for example, the clarification and/or formalisation of a requirement informally stated as a contract input. Moreover, the process of validation may reveal errors, weaknesses in or omissions from original specifications. Specifications will change as the process proceeds, independently of changes induced by changes in the application or operational environment.

A contract specification should contain, at least, a technical specification, management constraints (on process and products) and product acceptance criteria. It must also include identification of deliverables and, in particular, the contract product and any required reports. Once project activity has started there may be a more general flow of reports into and out of each sub-project. Some of these will have been anticipated and required, included in the deliverables or the management constraints. Others will be *ad hoc* but may nevertheless represent a significant contribution to the contract product. Finally, as mentioned above, a contract may give rise to other contracts. A contract hierarchy emerges. As previously observed the *active* hierarchy will grow and shrink as tasks are initiated or completed. The associated databases, however, will continuously grow since the information must, in general, be retained, at least, for the life time of the product for reference during system build and evolution.

At any one time, the active hierarchy defines the structure of current activity. A sequence of snapshots provides the total project history. In ISTAR the hierarchical structures also define the physical database structure, formal communication paths, reporting relationships and accountability, instantaneously and as they change over project life-time. Other database structures can be envisaged, although at the present time the project-model-related structure appears optimal.

6 The Information Repository (Databases) and IPSE (Project) Communication

In ISTAR, each active contract has and controls its own database. It is the collection of all such bases (and of the media in which the bases are to be found) that constitutes the project information repository. Their accumulation over time provides the product database, a multi-dimensional version preserving structure. By its very nature this database is distributed. It and its underlying facilities were developed from an earlier ECE (Pilot Esprit) study. As a consequence of flexibility considerations the first implementation was based on the binary relationship (BR) model, facilitating mechanical reasoning as well as direct information retrieval. For certain purposes, however, it is appropriate to build an ERA model on top of BR but tsuch a complex structure can lead to performance problems. A new, direct, ERA implementation has therefore been developed.

The distributed nature of project definition on a *per contract* basis provides a logical structure for communication facilities that can be individually matched to each activity, to the relationship of each to other activities, to the probable volume and frequency of communication and to the degree of protection required. The facilities may range from a boy on a bicycle through telecommunication links to use of shared devices. The nature of the link need not be predetermined by the process and project designer or visible to the user. It can be changed as required by needs and circumstances.

7 The Role of the Project Model and its Implementation

The ISTAR project model, *the Contractural Model*, provides a conceptual basis for IPSE architecture, its information and data management and project communication. It also provides a framework for the design and operation of tools in areas such as project management, data management, system and quality control. Specifically, the model and its reflection in system structure and functional content provide an environment for support of management responsibilities such as project planning, work breakdown and allocation, resource distribution and management, activity control. In general, the project model plays a crucial role in the provision and organisation of tools for management support. Equally, it provides conceptual and practical mechanisms for information management as required, for example, for variant and version control, system building and configuration control. Instances of its exploitation for quality assurance are provided by methods and tools relating to the management and implementation of work delivery, defect reporting and their analysis, correction and distribution. Finally it may be observed that integration of third party tools and tool development must also be supported so that the IPSE may be adapted to different environments and may evolve.

These examples suggest that a project model may have a direct, as well as a conceptual, role in IPSE development - both for initial architecture and design and for their evolution. The ISTAR experience, to date, has fully confirmed the benefit to be obtained from explicit adoption of a project model. The proposed models and structure provides the logical potential and the framework to meet the needs of each situation. Whether and how this is exploited in use of the IPSE depends on the process and project designer and its implementers. They, in turn, will be strongly influenced by methods (technical development and management) supported by the IPSE and by the availability of tools.

9 Methods and Tools

The objective of this paper has been to present the use of models in IPSE development. This has been illustrated by description of the top down approach to ISTAR development. Detailed discussion of ISTAR functional capability, (methods supported, tools provided) is not directly relevant to these objectives. It is, nevertheless, of interest to outline the next steps of the development process; functional decomposition of a target-system. Note that the resultant classification is not necessarily of practical value to a user. In the case of ISTAR, for example, a user's concern will be with the properties of a *Workbench* that groups and links the tools he wishes to use at any particular time, not with their common global functionality. This dichotomy provides an example of the partial orthogonality (in a multi-dimentional space) of design and usage concerns and structures.

Brief reference has been made in the paper to two functional categories, framework and tool set. The conceptual placement of certain capabilities is clearly associated with one or other of these categories. The logic that is used to assign others is less clear. But then does it really matter?

Framework function could include:

- Work Station Control
- Project Communication
- Information Repository and database
- Management Monitoring, Control and Communication
- Tool Packaging and Building Tools
- Utilities

A four class categorisation is appropriate for tools:

- Project Management
- Technical (Product) Development
- Tool Adaptation, Integration and Development
- Information and Data Management

Further detail of the specific methods and procedures supported in ISTAR, of tools included and of the workbench facility will not be provided here. To include them would not further clarify the approach. It would merely divert the readers' attention from the wider perspective presented and from future trends. It is, however, of interest that the enveloping facility that permits the ready integration of third party tools has proved to be one of ISTAR's most important facilities.

10 Conclusion

The notions presented in this paper are philosophical in nature but their partial realization in ISTAR is very real. As a consequence of practical experience and of an earlier study, British Telecom, IST's collaborator in the project, had developed a view of software development and IPSE requirements fully consistent with the concepts that IST wished to implement. Thus the development team received full support to follow the approach described to the point at which the above categorisations had been made. When, however, it came to the selection of specific methods and tools the choice had to be guided, at least in part, by the client's desire to move to disciplined, controlled and tool supported management procedures, to the initial continued use of their current practices and tools and to evolve to a fully supported development process, increasingly slanted to the use of formal methods. In these ways it was hoped to achieve greater reliability, increased productivity and a significant improvement in the capability for responsive but correct system adaptation in a changing operational environment.

In addition to the introduction of disciplined methods and their support, provision of staff familiarisation and training is an essential component of a strategy to achieve such goals. A change of attitude on the part of technical and managerial staff is also necessary. None of this can be achieved over night. *Evolution* is, therefore, the only practical route to the increasing exploitation of advanced software technology in an established development environment. The *revolutionary* introduction of formality, new methods and tools is neither technically feasible nor commercially viable. Introduction of new methods, installation of new tools, staff education and achievement of acceptance requires effort, time and patience. Meanwhile development must go continue.

Any organisation that has recognised the need for change will wish to proceed as speedily as possible. It will, therefore install development support and gradually enforce usage and enlarge the repertoire, while recognizing that in a rapidly advancing but, as yet, untested field any installed support will need itself to be tested in use, evaluated and evolved over a series of releases or versions. In this respect, at least, IPSEs are no different to any other large, software dependent, system [LEH85]. Practical experience with ISTAR is showing this to be a viable transition strategy to advanced software technology. That is, methods supported by ISTAR in its first releases and tools provided, reflect the best of currently applicable technology. Emerging technologies such as stepwise formal development, mechanical verification, systematic validation at each step, the use of mechanised reasoning and of expert system technology, are all likely to be candidates for application in the process that has been visualized. That is for the future. In the meanwhile, the likely directions of future expansion and evolution of IPSEs in general, and of ISTAR in particular, are well understood. As a consequence of the model based approach adopted their exploitation, in ISTAR at least, is believed to be within reach.

Software that is in continuing use must be repeatedly adapted to changes in its operational environment, to changes in the perception of that environment, to advances in technology. Moreover, the system must remain correct under such adaptation. Dynamic correctness and changeability of a software system, the ability for the maintenance and system evolution activity to respond correctly and with the necessary speed to needed adaptation, are all a consequence of the development process. As computers penetrate ever more deeply into every aspect of society, as mankind becomes more and more dependant on computers, these attributes become critical for the survival of mankind. It is thus no wonder that increasing attention is being given to discipline, formality and process support, to the concept and realization of truly integrated project support environments.

An IPSE is a complex software system. The process by which it is developed will largely determine the nature and quality of its further evolution, in particular with respect to effort required, correctness and responsiveness. Its own function and structure will, in turn, influence the characteristics of programs and systems developed with its use. Thus, if processes and projects it supports are to be effective, understanding of their desired properties must dominate IPSE design.

It has been the thesis of this paper, reflecting the technical philosophy of Imperial Software Technology Ltd., that an IPSE design process must begin with adequate models of the program development process and of project structures that implement it. This was the case for the approach taken in the development of the IST ISTAR IPSE used here to illustrate an approach to software development in general. Prototypes of ISTAR have been running since April 1985. The first production release became available on schedule in the spring of 1986. The system is about to be installed in a number of major sites. All the experience to date suggests that project goals have been met and that the approach taken to IPSE design and evolution is effective.

11 Acknowledgements

I make grateful acknowledgement to all my colleagues at IST who have been infinitely patient in educating me and providing the information needed to put this paper together. Most especially I must express my gratitude to Vic Stenning, Technical Director of IST and the inspiration behind all those who have worked on this project. My deep appreciation and thanks also to Wlad Turski who has so patiently guided me over these past few years.

12 References

[BAR84] Bartlett A J, Cherrie B H, Lehman M M, Maclean R I and Potts C, 'The Role of Executable Metric Models in the Programming Process', Final Report to DARPA, order no. B-2-3288, under contract no. F49620-82-C-0098, Apr. 1984

[BOE76] Boehm BW, 'Software Engineering', IEEE Trans. on Comp., vol. C-5, no. 12, Dec. 1976, pp. 1226-1241

[IST83] 'Requirements for Software Engineering Databases'. Final report to the CEC under the Preliminary Esprit Program, July 1983, Imperial Software Technology Ltd., 60 Albert Court, Prince Consort Rd., London SW7 2BH

[JON86] Jones CB, 'Rigorous Program Development Using VDM', Prentice Hall Int., 1986

[KOE64] Koestler A, 'The Act of Creation', 1970 edition, Pan Books Ltd, London, pp. 176-177

[LEH84] Lehman MM, Stenning V and Turski WM, 'Another Look at Software Design Methodology', ACM Software Eng. Notes, vol. 9, no. 2, Apr. 1984, pp. 38-53

[LEH85] Lehman MM and Belady LA, 'Program Evolution - Processes of Software Change', Academic Press 1985

[LEH85b] Lehman MM and Stenning NV, 'Concepts of an Integrated Support Environment', Data Proc., Butterworth & Co (Pubs) Ltd., London, vol. 27, no. 3, April 1985, pp. 8-10

[LEH86] idem., 'Advanced Software Technology - Development and Introduction to Practice', Invited Lecture, Information Processing '86, Proceedings IFIP Congress 1986, Dublin, Sept. 1-5, Publ. by Elsev[illegible] Sci, Pub. (North Holland), 1986, pp.605-611

[LEH86b] idem., 'Modes of Evolution', see [SPW87]

[LEH87] Lehman MM and Turski WM, 'Essential Properties of IPSEs', SE. Notes, vol. 12, no.1, Jan. 1987

[LEH87b] Lehman MM, ' Process Models, Process Programs, Programming Support - Invited Response To An ICSE9 Keynote Address By Lee Osterweil', Proc.9th Int. Conf. on Software Eng., Monterey, CA, 30 March - 2 Apr. 1987. To be publ. by IEEE Comp. Soc.

[LEH87c] idem., 'Desirable Properties of the Software Development Process', sub. to SE. Notes

[MAI87] Maibaum T and Turski WM, 'The Specification of Computer Programs', Addison Wesley, to be published, Spring 1987

[PDS87] Proc. of the 2nd ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Development Support Environments, ACM SIGPLAN Notices, vol. 2, no. 1, Jan. 1987

[SIM69] Simon HA, 'The Sciences of the Artificial', M.I.T. Press, Cambridge, MA. 1969

[SPW84] Potts C (ed), 'Proceeding of the Software. Process. Wrkshop', Egham, Surrey, U.K., Feb. 1984. IEEE, cat. no. 84CH2044-6 Comp. Soc., Washington D.C., order no. 587

[SPW86] Wileden JC and Dowson M (eds), SE Notes Special Issue on the 2nd International Workshop on the Software Process and Software Environments, Coto de Caza, Cal., 27-29 March 1985, vol. 11, no. 4, Aug. 1986

[SPW87] Dowson M (ed), 'Iteration in the Software Process', Proceedings of the 3rd International Process Workshop, IEEE Comp. Soc. Press, March 1987

[STE85] Stenning V, 'Software Engineering: Present and Future' in The Corporate Database, State of the Art Report 13:3, D Iggulden (ed) 1985, published by Pergamon Infotech Ltd, Maidenhead, England

[WIR71] Wirth N, 'Program Development by Stepwise Refinement', Comm. ACM, vol.14, no.4, Apr. 1971, pp. 221-227.

A Short History of the Gandalf Project

A. N. Habermann
Carnegie Mellon University
Pittsburgh, PA 15213

6 February 1987

Abstract

The Gandalf project is concerned with the generation of interactive special-purpose programming environments. The project went through three phases which each lasted approximately two years. A fourth phase is planned for the period of 1986 -1988. The subjects of study and experimentation during the first three phases were respectively: tool integration, automatic environment generation and active-object environments. The fourth phase will focus on building expertise into environments. One of the tangible results of the project is the Gandalf System which provides elaborate facilities for designers to specify and automatically generate target programming environments. The Gandalf system is itself an example of a special-purpose programming environment and serves as a workbench for the design and implementation of other special-purpose programming environments. The products of the Gandalf system provide an integrated tool set that interacts with its users in terms of task-specific structured objects instead of characters and text. This paper describes the development of the Gandalf facilities during the three completed phases and indicates the project's long-range research and development program.

The Gandalf project is supported in part by Siemens Corp. ZTI, Munchen/P

1. Introduction

The object of study and experimentation in the Gandalf project is the design and implementation of interactive special-purpose programming environments that provide task-specific assistance to their users. Some examples of the type of programming environments we have in mind are: an interactive electronic mail system, a software development environment for system configuration management, an interactive environment for teaching introductory programming, etc. Each of these environments includes facilities that are specifically designed to help the user to do a particular task.

Taking the special-purpose approach, one must face the fact that many slightly different environments are needed in cases where a single general-purpose environment suffices. The larger the numbers, the more important it is to find ways of reducing the effort of generating a particular instance of an environment. The special-purpose approach won't work if creating such an instance requires the same effort as creating a general purpose environment. The Gandalf project has therefore focused on the development of tools and facilities that reduce the effort of generating an environment to a fraction of the time it takes to do the same job by hand. The reduction is largely obtained by a high degree of code sharing between environments and by automatic code generation.

The tools and facilities available for environment generation are assembled in a system that we call the "Gandalf system". This system is a workbench for implementors to design and automatically generate target environments. It builds on the general ideas of the Gandalf project and is itself an example of a special-purpose environment to support the generation of environments. In order to distinguish the user of the Gandalf system from users of programming environments generated with the Gandalf system, we refer to a person in the first category as "designer" or "implementor" and to a person in the second category as "user". The special-purpose environments generated with the Gandalf system are from here onwards referred to as Gandalf environments. Using this terminology, a designer uses the Gandalf system to generate a Gandalf environment.

The Gandalf project went through several phases in which tools and techniques were developed that facilitate the specification and automatic generation of special-purpose environments with the characteristics described here. The first phase involved the integration of the tools in a user environment into a coherent collection that can support a specific user task. This phase started out with integrating tools for program construction and was continued with the integration of tools for system version control and system configuration management.

The second phase, which partially ran in parallel with the first, concerned the generic approach to specifying and generating special-purpose environments. It included the design and implementation of the common parts of all Gandalf environments and of a description mechanism for designers to specify the particular facilities of a target user environment. A large part of the work went into the design and implementation of program generators that transform a designer's formal description into programs and tables that can be processed by a traditional compiler. The third phase involved the design and implementation of a mechanism for a designer to describe semantics and runtime support. The most important aspect of this phase is the development of the *Action Routine* mechanism which allows a designer to define so-called *daemons* that are attached to objects in the user environment for the purpose of interacting with the user and with other objects in the environment. All Gandalf environments share a runtime Action Routine support mechanism that triggers daemons depending on the occurrence of specific events such as the creation or deletion of an object.

2. A Historic Overview

The first phase, tool integration, consisted primarily of three subprojects, SDC, SVCE and LOIPE, in which we focussed respectively on tools for project management, for system configuration control and for the edit-compile-execute cycle. The results of these three subprojects were later merged into a single environment, the Gandalf Prototype, which provides a complete set of tools for software system development.

It was pretty clear from the beginning of the project that building a variety of task-specific programming environments is not a viable proposition without mechanizing the generation process. It would take too long to build one and it would be very hard to maintain a number of them. The second phase of the project concentrated therefore on designing a description formalism with a collection of support tools that allow a designer to specify a target environment and automatically generate it from the description. First, we focussed on a structured database design and on the representation of objects in the database. This work was done initially in three subprojects, ALOEGEN, DBGEN and SMILE, dealing respectively with database editing, environment generation and the working environment for the implementor. These tools and environments were used to build a number of programming language environments. The most successful one of these environments is GNOME (for Pascal), which has now been used for a number of years in the introductory programming course at CMU.

The design of structure and representation in the second phase was followed by a third phase in which we concentrated on semantics and runtime support. This phase consisted primarily of three subprojects, the design of the Action Routine Language, ARL, an editing environment for ARL and a system kernel, IPE, which provides the runtime support for every target environment that is built with the Gandalf tools. At the end of the third phase, the tools were integrated into a single implementor's environment, the Gandalf System. A short description of each of these subprojects follows in subsequent sections.

3. Software Development Control

The SDC environment controls the access of programmers to shared modules and to the common pool in which these shared modules are kept. The necessary protection of the shared modules in the common pool is obtained by defining a module as an abstract datatype which can be manipulated only through a well-defined set of operations.

Changes to the common pool are made through pairs of *reserve* and *deposit* operations. A reservation is not successful if the common module in question is already reserved. A deposit is only successful if the programmer performing the deposit is the same as the one holding the reservation. This arrangement assures that the modifications by two or more programmers to the same module cannot overlap in time. An alternative way of terminating a reservation is by using the *release* operation which is used when the programmer wants to give up his reservation without making a new version public.

In addition to the reservation field, the module datatype contains two lists that respectively describe the access right of programmers and the modification history. SDC distinguishes between three levels of access rights. The lowest level consists of programmers not listed, who have read-only rights. The middle level consists of the project programmers who have source modification rights and can add to logfiles in which the modification history is kept. The higher level consists of the project managers who have all the rights of the others but can also edit and purge logfiles and access lists.

The logfile of a module contains one message for every deposit. When a programmer deposits a new version of a module, he is prompted for a description of the modification, while the deposit operation automatically enters the date and the programmer's name to the message. Logfiles can be read in their entirety or selectively by date and/or programmer name.

4. System Version Control

SVCE allows a system designer to specify the functionality of a module and its dependency on other modules through an export and an import list. Each specification can be realized by a multitude of implementations, both of the variant and of the successive version type.

The most elementary device for building systems is the ability to write a *system model* that lists a collection of module versions that together can be used to generate a subsystem. The elements of a system model are in essence values that represent particular implementation versions of a program module. Subsystem descriptions in SVCE may contain such specific values, but more common is the use of variables that represent modules or subsystems. The advantage of using variables is that the logical structure of a subsystem can be defined without referring to a specific implementation. These variables represent module and subsystem specifications, but not their implementations. A choice as to which version or variant to use can then be made each time a subsystem is instantiated.

5. Incremental Program Construction

Lisp environments have the nice characteristic that programs can be debugged in the source language, because the Lisp interpreter allows one to execute a function in the same environment in which it is defined. The purpose of the LOIPE subproject was to design a similar uniform interface for procedural languages based on incremental compilation. LOIPE integrates program editing, compilation, execution and debugging. All operations in LOIPE are cast in terms of language constructs such as statements, expressions, functions, etc. LOIPE represents programs as trees that correspond to the intermediate representation generated by the parser of a traditional compiler. The editor maps the tree representation into user readable text, while the code generator maps it into object code. Both text and object code can be viewed as derived objects. The debugger and the runtime system map the state of the executed object code back into the tree.

LOIPE provides a uniform editing interface to its users. Debugging takes place through inserting trace and break statements into the tree with the use of the editor. The main difference between LOIPE and the traditional programming interface is that in LOIPE the editing units are programming language construct instead of characters and text. One benefit is that no syntax errors can occur, because programs are not generated as text. Another benefit is that the editor is able to perform static semantic analysis on the fly and warn the user, for example, when procedure calls don't agree with their declarations or when names remain undeclared.

LOIPE provides an editing command for executing programs. The execution command does not have to call the compiler or linking-loader because of LOIPE's incremental compilation and incremental linking mechanisms. The code generator is automatically invoked each time the user has completed the declaration of a function or procedure. If the user writes short subprograms (on the order of one to two pages), he will hardly notice the delay caused by the code generator.

Incremental linking is based on generating indirect procedure calls. A module that contains a number of subprograms starts off with an *entry vector* in which the locations of these subprograms are listed. Access to these subprograms from outside of that module is obtained solely through the entry vector. If the location of a subprogram changes because of code modifications, all that changes is the value of the corresponding entry vector element, while access from outside still refers to that element without change.

6. A Language-Oriented Editor Generator

The ALOE subproject started out with the idea to explore the pros and cons of syntax-directed editing. This work complemented the LOIPE effort in providing the necessary mechanisms for editing the syntax tree and for mapping the syntax tree into user-readable text[1]. Soon it became clear that having the user think in terms of syntactic constructs is not the important issue, but having the user interact with the environment in terms of structured objects instead of uninterpreted text. We therefore gradually changed our use of terminology from syntax-directed editing to structure-oriented editing or structure editing.

ALOE distinguishes between *abstract syntax* and *concrete syntax*. The abstract syntax describes the logical structure of objects, while the concrete syntax describes their representation. For instance, the abstract syntax description of electronic mail messages defines a message as a composition of sender, receiver, date, subject and text. The concrete syntax defines the keywords and the layout of messages when printed on paper or when displayed on a screen.

The ALOE design and implementation resulted in an environment generation scheme that is still in use in the Gandalf system. This scheme is depicted in Fig. 1.

The designer's description of the abstract and concrete syntax is represented in the upper left corner. This description is transformed by the Program Generator into C-programs and tables that implement the objects that represent the task-specific nature of the target environment. These specific facilities are compiled and linked with the common facilities that are shared by all target environments in which the user can generate and manipulate objects defined by the designer's syntax description.

The common facilities serve three purposes: input/output, long-range storage and database management. Input/Output and file storage procedures are provided by the ALOE library and database management in the target environment by the *ALOE kernel*. The kernel controls cursor movement, modifications to the user's database and interpretation of the concrete syntax descriptions that determine the output formats.

[1]This mapping from tree to text is called unparsing, since it operates in the opposite direction of a parser which maps program text into a syntax tree.

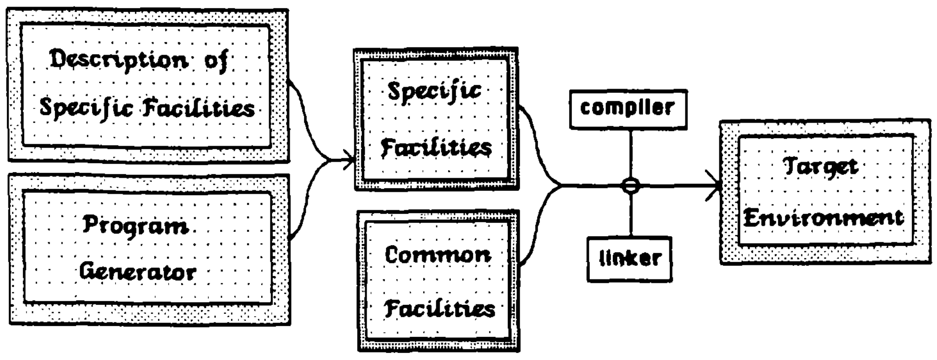


Fig. 1: The ALOE Generation Scheme

7. Action Routines

Semantics are handled by *Action Routines* which can be attached to object types described in ALOE. An Action Routine might be considered as a procedural field of a record. However, an Action Routine field is not activated directly by the user, but indirectly as a result of events that are caused by the user's actions. This approach makes the Action Routine mechanism suitable for dealing with issues such as memory management, semantic checking and window management. Action Routines were initially written in C and later in ARL.

8. SMILE

The SMILE system was designed as a software development environment that combines the features of SDC and SVCE. The SMILE environment consists of a common pool of shared modules that can be reserved in the user's local workspace. Modular interfaces are defined by export and import lists that are checked when a system is generated. The facilities of SMILE are not as elaborate as those of SDE and SVCE combined, but are designed to provide a smooth support for the most frequently applied operations on modules. SMILE is used intensively in the Gandalf project as a practical environment for tool development.

The environment designer starts with a SMILE database that serves as the common pool for all project modules. The designer enters a particular subenvironment depending on the type of module he creates or modifies. For instance, if he writes a syntax module, he automatically enters the ALOEGEN subenvironment. When he is finished with a module, the designer automatically enters the enclosing SMILE environment in which modular interface checking and automatic (re)compilation take place.

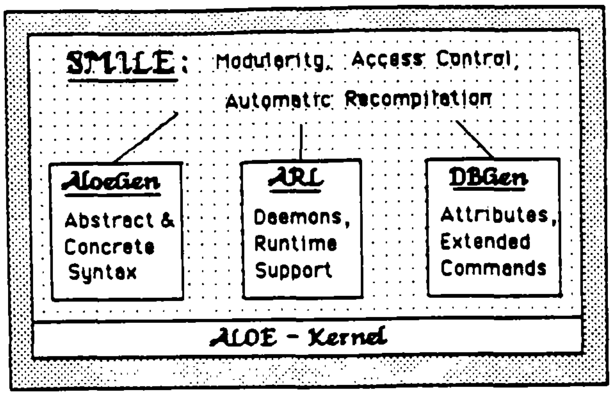

SMILE: Modularity, Access Control, Automatic Recompilation
AloeGen
Abstract & Concrete Syntax
ARL
Daemons, Runtime Support
DBGen
Attributes, Extended Commands
ALOE - Kernel

Fig. 2: SMILE and its Subenvironments

9. The Gandalf Prototype

The Gandalf Prototype is an implementation of an experimental environment that combines SDC, SVC and LOIPE. The purpose of implementing this prototype was to synthesize the ideas and mechanisms designed for the various tasks of program construction (programming-in-the-small), system version control (programming-in-the-large) and project management (programming-in-the-many). The prototype environment has been used to demonstrate the flexibility of Gandalf environments with respect to the implementation of program management rules. The rules implemented in the prototype are just an example that can easily be modified or extended. The importance of implementing management rules in the environment is that the environment can enforce these rules.

10. Editors and Gnome

A structure editor can be useful even without an adapted code generator. One can use the text that is generated by the unparser as ordinary source code. This idea was first applied when we generated a syntax-oriented editor for Pascal that we wanted to use for teaching programming. The initial experience was encouraging enough to start a specific project, the Gnome project, which aimed for a production-quality Pascal environment for novice programmers. The initial Gnome environment was gradually transformed into an integrated system in which both editor and codegenerator operate on the tree representation of a program. The current Gnome system does incremental compilation on the fly.

The Gnome system has been used for a number of years and has improved considerably since its inception. A major software engineering effort went into providing a smooth user interface. During recent years a high quality version of the Gnome environment has been created for the Macintosh, which makes use of mouse control and elaborate window facilities. This version is appropriately called MacGnome.

11. The Action Routine Language

The syntax description that the designer generates with the help of ALOEGEN determines the structure and the representation of the objects in the target environment. This description does not determine the *behavior* of these objects, because no operations on these objects are included in the syntax description. The first step in the direction of describing the behavior of objects in the target environment was the introduction of the *Action Routine* concept described earlier in this paper.

The original design had the drawback that Action Routines had no state and depended entirely on deriving conditions and values from existing user objects. This problem was resolved by the introduction of *attributes* that can be attached to objects in addition to Action Routines. The main distinction between attributes and other object components is that attributes are not visible to the user and cannot be directly operated upon by the user. Attributes are accessible to Action Routines and to Extended Commands.

Attributes can be of scalar type such as integer or character, but can also be of type tree. Attribute trees differ in no way from ALOE trees that a designer can describe. It is thus possible for a designer to define a separate grammar for an attribute tree, or even for a number of attribute trees. The structure description, then, consists no longer of a single grammar, but of a number of grammars describing the structure of the visible parts of objects and of the associated tree attributes.

12. The ARL Environment

The ARL environment is an integrated environment in the style of LOIPE. It provides a language-oriented editor for writing Action Routines, it gives access to the extensive ARL library, and it performs incremental compilation. The ARL library consists primarily of useful tree-traversal routines that are used for collecting data from objects and attributes. Semantic checking in ARL is limited to static semantics, which corresponds to type and parameter checking in other procedural languages. However, even with this limitation, the ARL environment has proven to be extremely useful. It has made it much easier to describe the dynamic behavior of objects in the target environment. The combination of language, library and environment is a tremendous improvement over writing C procedures by hand. Action Routines written in ARL are called *daemons*.

13. The IPE Kernel

The runtime environment of all target environments is determined by the *IPE kernel*, also known by its old name, the ALOE kernel. The kernel is part of the common facilities (cf. Fig. 1). The kernel is able to handle predefined events, such as create or delete an object, and also designer-defined events. The designer can, for instance, define an event that is triggered in one daemon with the effect of activating another daemon that is waiting for that event.

The IPE kernel is a completely redesigned version of the ALOE kernel. It contains the necessary mechanism for activating daemons depending on the occurrence of events. This version of the kernel gives the database in the target environment its event-driven character. The IPE kernel partitions each step into three phases: a permission phase, a pre-operation phase and a post-operation phase. Daemons can make use of these phases and act in different ways depending on whether an operation is attempted, is going to take place, or has been completed.

14. The Gandalf System

The Gandalf System consists of the SMILE environment and the three subenvironments, ALOEGEN, DBGEN and ARL (cf. Fig. 2). The SMILE environment assists the implementor in the design of modular interfaces, in system configuration and version control and in automatic system generation. It also provides a small set of useful project management rules that prevent race conditions and maintain development history. The ALOEGEN environment is used for describing syntax and for connecting grammars with daemons, DBGEN for connecting objects with attributes and the ARL environment for writing daemons.

The system has been released in January 1986 and is available in the public domain. A small fee is charged for shipping and maintenance. Usage of the system is subject to the normal restrictions on products of non-profit organizations. The system consists of approximately 40.000 lines of kernel code and 50.000 lines of code for the four development environments. The ARL language is comparable in size to Pascal.

15. The Gandalf Programme

The Gandalf project does not have a particular target other than improving the generation of high quality programming environments. The project is therefore open-ended and will evolve with the development of the technology. The next phase that is planned for the period 1986-1988 concerns the issue of building expertise into environments. We believe that the daemon model offers an interesting alternative to production system implementations of expert rules. A potential advantage of the daemon model is that expertise can be associated directly with individual objects in the user's environment which has the beneficial effect of introducing a natural localization of rule applications. We also believe that an important element for building expertise into environments is the observation of the user's behavior and, in general, of the history of interactions. These types of issues have been adopted as the starting point of the next research phase.

Further development will take place in parallel with basic research. An important part of the development effort will go into an implementation of the VIEW mechanism. A mechanism for specifying static and dynamic views was designed during the past year which allows a designer to generate an environment as a synthesis of various complementary views.

References

[a] Hansen, W. J.
"Creation of Hierarchic Text with a Computer Display"
Ph.D. Thesis, Stanford University (June 1971)

[b] Habermann, A. N. and D. Notkin
"Gandalf: Software Development Environments"
IEEE Transactions on Software Engineering, Vol. SE-12, No. 12 (Dec 86)

[c] Teitelbaum, R. and T. Reps
"The Cornell Synthesizer: A Syntax-Directed Programming Environment"
Comm. ACM, 24, nog. (Sept 1981)

[d] Donzeau-Gouge, V. et al.
"Programming Environments Based on Structure Editors: the MentorExperience"
INRIA, Technical Report, No. 26 (May 1980)
Also in "Interactive Programming Environments", pp 128 - 140
D. R. Barstow et al. Eds. McGraw Hill, New York (1984)

[e] Reiss, S.P.
"Graphical Program Development with PECAN Program Development Systems"
Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Programming Environments
Pittsburgh, Pennsylvania (April 1984)

[f] Henhapl, H.
"PSG, a Programming System Generator"
Technical Report, TH Darmstadt (September 1985)

Information Management Challenges in the Software Design Process

T. Biggerstaff, C. Ellis, F. Halasz, C. Kellogg, C. Richter, D. Webster

Microelectronics and Computer Technology Corp. Austin, TX. 78759

Abstract: The *upstream* software design *process* challenges database technology via its requirements for effective handling of diverse data -- both unstructured and structured, fuzzy and concrete, incomplete and fully specified, informal and formal, complex and primitive. Commercial databases provide robust commercial products that deal well with the "easy" (e.g., complete, regular, precise, formal, consistent) problems. Data, knowledge and information based technologies must be developed further and merged to deal with the "hard" ones.

1. Introduction

The MCC Software Technology Program (STP) is dedicated to long-term research in the *upstream* of software system design and integration with emerging technologies so as to achieve an extraordinary improvement in software productivity. Many of the challenges STP faces are peculiar to software, especially upstream, design (e.g., partial execution, the intangibility of the final product and the need to support fuzziness, deferral of commitments and exploration). Much of STP's effort is directed toward producing an environment, christened *Leonardo*, that supports team development of very large, complex, distributed, real-time systems. The rationale behind this research *programme* has been discussed elsewhere [3; 11; 17]. Here we wish to emphasize the information management aspects of *Leonardo*, to see where STP can or must take advantage of current and projected database techniques, and where *Leonardo* will place stringent requirements on developing new information management technology. After a brief discussion of *Leonardo*, we will specify a number of *Leonardo's* requirements that seem related to database management and then analyze database concepts and technologies in terms of their abilities to meet such requirements.

2. Leonardo — The Designer's Environment

STP is committed to *upstream* research which significantly improves productivity in *large-scale, coordinated* system development. Whereas the downstream of software development is formal and concerned with artifacts such as code, modules and documentation, the upstream is informal, often fuzzy, and concerned with the early phases of the development process -- requirements, issues, high-level concepts, in short, with *abstractions*. The fuzziness of the upstream process is amplified by difficulties in technical and managerial information flow. The emphasis on large-scale systems is more specifically directed at distributed systems, as well as organizationally complex systems, such as those developed by large teams. In fact, an important implication of this emphasis on large, complex systems is that upstream abstractions must be communicated and explored within an extended, *cooperating group* of people, with the aims of finding and representing appropriate abstractions for a problem, and analyzing and refining the representation to a point sufficiently concrete to feed the downstream.

Much of the effort in *Leonardo* is concerned with the capture, representation, elaboration, reuse and presentation of design information, i.e., information that is generated during the design process to define and discuss the product that is being developed. However, *Leonardo* must also be aware of the design process itself, since it cannot be decoupled from (though it may certainly influence) the policies and procedures within the organizational, social and educational structure in which it is embedded. Some of the research within STP addresses issues such as reusing design abstractions and artifacts [4], designing and programming large distributed systems [10], representing and manipulating coordinated systems, visually and dynamically [13], validating a hypermedia representation in the design environment [7] and managing a large base of persistent, complex objects [9].

3. Leonardo's Design Information Space

One of the most striking features of *Leonardo* is the amount, complexity and diversity of the data that are encountered in the design environment. With such a vast array of data, management becomes a key factor. A major part of the *Leonardo* effort lies in defining its so-called *Design Information Space*, deciding what it is and how to manage it.

We envisage the Design Information Space as a large network of diverse data relevant to the design *process*, e.g., the *issues* that arise during the design process (including postulated *alternatives*, their *resolution*, through *decisons* and *rationale*, ultimately leading to *commitments* and *impacts*), representations of the various *stakeholders* and their *views* and *interests* (including *work procedures* and *working relationships*), *notes* and *comments* (including *to-be-decided's* and *questions*), and *artifacts* (including *code*, *documentation* and even the *design* itself). It is difficult to specify precisely what data structures are required for the vast diversity of data we see in use today, especially since those structures are themselves dynamic, evolving entities. Furthermore, there is more to supporting the design process than merely capturing the information. The information must be represented *effectively*, so it will be appropriately accessible, useful and reusable to the design team and other stakeholders. The aim is to decrease the intellectual burden on the designer by shifting it onto tools and support systems, e.g., by managing the data and their interdependencies as much as possible, identifying potential problems and consequences, retrieving and even recognizing similarities and analogies. Consequently, we see the following aspects as being critical in determining both the structure of the Design Information Space and its requisite support tools:

- **Diverse, Non-Canonical Structures:** We have given ample indications of this aspect already. The information elements can be highly irregular or unstructured and not easily described in canonical data models.
- **Mixed Formality and Informality:** A software design contains highly diverse elements ranging from structures as informal as natural language text to structures as formal as predicate calculus expressions. Each must be dealt with in its natural mode. For example, we must be able to perform associative searches on natural language text and logical inferences on predicate calculus expressions.
- **Fuzzy and Inconsistent Information:** Fuzziness arises in the design arena both in terms of incomplete information or deferred commitments, and plausibilty or probability. In the earliest phases of design, concepts are often poorly (incompletely or incorrectly) ~~defined; furthermore,~~ different concepts may be inconsistent with one another. Deferring binding until quite late in the design cycle provides the designer with more flexibility and simplifies the task of managing and coordinating details. However, such fuzzy, partially specified data must not only coexist with precise, completely defined data, but in general must evolve through degrees of precision. A designer should be able to create a symbolic representation for something that will eventually develop into a very complex structure with perhaps far reaching effects, yet can remain fuzzily specified in the interim. The essence of software design is fuzzy representation evolving into more concrete representation.
- **Factored Designs:** Designs consist of factored and compartmentalized information elements, with local attibutes and behaviors that are independent of the specific application programs which use the elements. Design objects are complex and locally intelligent; they can be decomposed into multilevel subobjects, each of which can act according to its own pattern of behavior (e.g., producing descriptions and demonstrations of its functionality or usage). A trend toward highly factored designs can be seen in the concepts of structuring (e.g., subroutines, macros and remotely defined symbolic names for values), object-oriented programming, remotely defined program proto-

cols and parameterized packages. Factored structures appear to be a key concept in permitting deferred commitments and incremental development.

- **Rich with Dynamic, Unpredictable Relationships:** When designing a complex system, we find that most items are related in various ways to many other items in the system. Such complex webs of relationships must be represented explicitly. Relationships can arise from both logical associations among design objects (links) and pragmatic limitations and dependencies within the system (constraints). Symbolic representation of deferred commitments requires relationships to keep track of the logical connections between fuzzy or partial structures that will eventually evolve into a tightly defined and integrated structure. Moreover, in the upstream phase, designs change rapidly and unpredictably (e.g., as understanding of the problem increases, the stakeholders learn more about each other's concerns and the requirements or specifications are altered or renegotiated), creating the need for a system flexible enough to deal with evolving patterns of relationships. Finally, constraints often represent eventual goals, necessitating nonuniform integrity control throughout the various design stages.
- **Size, Scale:** *Leonardo* is concerned with large, complex systems; so the Design Information Space will contain a vast amount of information, both in terms of design elements and data. We estimate that the information for a single project will be at least an order of magnitude greater than the source code, and that the common, reusable information (i.e., world knowldege) accumulated over a variety of projects will be several orders of magnitude beyond the source size for any one application. The scale issue also emphasizes the likelihood that only some, not all, of the data can be captured.
- **Exploration and Navigation:** Much of the communication between the designer and the Information Space will be both interactive and exploratory. Typically, a designer selects parts of a design and performs dissimilar operations on them. Navigational or "focused browsing" support will be needed to allow him to view the objects he and *Leonardo* deem most relevant, as he attempts to refine, understand or reuse a design.
- **Teams and Coordination:** The Information Space will be embedded in a larger system context of individual workers, technical teams, projects, business units and customers. As such it serves as the primary coordination mechanism for the sharing of many types of information among the various stakeholders. While in the past teams have tended to divide system design into disjoint pieces (allowing everyone to proceed in parallel with a minimum of communication), the goal in *Leonardo* is to permit teams to work *together* as a more productive whole. This goal seems to require simultaneous access to the same data, or notification mechanisms to allow effective pseudo-concurrent access. For example, several individuals might need to truly share a file, with changes by each being visible in real time to everyone else, e.g., in a distributed design meeting in which the participants are in distant locations, but simultaneously viewing and manipulating the same information. Large software design projects involve significant planning and coordination efforts. *Leonardo* must record administrative and temporal information -- design plans and specifications, pert charts, module dependency lists, and code segment compilation timestamps, etc. -- and be able to monitor and act upon this information.
- **Complex, long-lived transactions:** If we view a design session as a transaction on a design database, then we may have many nested subtransactions within a main transaction that may last for several days. Within a "transaction", a designer will be exploring alternatives, trying one idea, then partially "undoing" it, and perhaps several other subtransactions, to try another.
- **Layered Abstraction and Inheritance:** Many information elements and their behaviors can be organized into layers of abstraction with each lower layer being differentiated from the layers above by the addition of more specific properties. The behavior associated with any specific information element may then be drawn selectively (i.e., inherited) from those associated with the element and its parents.

- **Contexts:** Within the Information Space, groups of related information elements will be clustered into contexts that articulate specific roles or reflect particular perspectives, e.g., the adaptable views required by different specialists on design teams, as well as individual members' hypothetical or exploratory endeavors. Large scale operators and actions generally will be restricted to such contexts. Moreover, design team members must be able to access public or team data, and incrementally merge or integrate their views, possibly retaining or modifying their individual perspectives. Of course, in the design process, a view may be fuzzy and may exist independently of the unified, global object supposedly being viewed.
- **Virtualization of the Information Space:** Ideally, the designer should not be aware of any difference in behavior between objects which are locally created and manipulated and objects which must be obtained from the global information base. The Information Space should appear to be a seamless, virtual extension of the designer's workstation environment. However, this ideal must be relaxed a little in the context of teams, sharing and lengthy design sessions, since designers must be informed of long delays or important modifications.
- **Temporal Data/Histories:** Permanent records are required for most Information Space interactions. *Leonardo* must record, mangage and manipulate design changes and alternatives, including such things as versions (revisions, releases and variants) and configurations. These management problems are complicated by the fact that design histories can develop along several axes, including temporal and abstraction axes.

4. Applicabilty of Database Technology

Each of *Leonardo's* modifiers -- coordinated, large-scale, complex, distributed, real-time, software and upstream -- places requirements and constraints on its underlying information management system. Having described a number of these requirements, we now wish to analyze them in terms of modern database features and concepts: data models; schemas; queries; views; protection and integrity; concurrency and transaction granularity, backup and recovery; performance.

- **Data Model:** The diversity, complexity and irregularity of the data in the Design Information Space provides a modeling challenge. A comprehensive data model, one which can cope with complex as well as simple objects and that can deal with the storage and retrieval requirements of multimedia/hypermedia environments, is required. Design activity is often concerned with the creation, modification, and retrieval of complex objects. Design objects can be grouped within a class hierarchy or lattice. The inheritance capabilities usually associated with such structures have proven useful in support of software design (Cf. [22].). Thus, in many respects our needs seem to naturally dictate a semantically enhanced object-oriented database model (Cf. [1; 16].).
- **Database Schema:** Relational schemas are relatively static, being defined at database creation time and then changed infrequently thereafter. Object-oriented and functional models may be more appropriate for the complex, rapidly evolving Information Space schema [2].
- **Database Query:** In contrast to typical data processing, where one isolates a few tuples prior to an update or performs the same operation on a large group of selected items, *Leonardo's* preferred access mode should be (composite) object-at-a-time, as the Design Information Space is explored or browsed (Cf. [16].). Still, the power provided by the completeness and efficiency of relational set-at-a-time access capabilities also seems to be essential to support some kinds of retrievals. Semantic relationships in functional and many object-oriented systems reputedly support the *meaningful* set-at-a-time accesses, but without a theoretical foundation or some of the query optimization and join techniques characteristic of modern relational databases. We have also seen that *Leonardo* requires more sophisticated query support, e.g., inferencing, pattern-recognition and analogy, than that provided by conventional database systems.
- **Database Views:** We have seen that *Leonardo* incorporates widely varying perspectives on the contents of the design database. Relational methodology provides a sound formal basis for the

construction of arbitrarily complex user views and their use in retrieval and display [25]. View update and merge mechanisms are not yet available, but would be highly beneficial in a team design environment.

- **Database Concurrency and Transaction Granularity:** Simultaneous access to varied forms of design information by groups implies a need for enhanced sharing. Transactions, the standard mechanism for maintaining consistency among concurrent activities (the atomicity property), may be quite long in *Leonardo*. Conventional databases assume that transactions last a few minutes at most, or that conflicts will arise infrequently, and thus use some form of locking or optimistic concurrency control. Standard locking mechanisms do not allow several individuals to simultaneously access an object in write mode. It is not viable to lock everybody out of the Design Information Space while very long transactions are completing. In the design environment, there is a much higher probability of access conflicts (or access overlaps, to be more accurate); so optimism wouldn't be effective. Hence, there is a need for more flexible concurrency control mechanisms such as soft locks and notification [9]. Alternately, a version of the database might be created for each long transaction, with full concurrency and integrity control available for the micro-transactions within it.

- **Database Protection and Integrity:** Standard protection facilities such as passwords and encryption are certainly needed in design databases. However, in the software design environment, there is much more need for protection against accidental information loss and inconsistency than against malicious destruction or intruders. Policies for the control and use of different versions of complex design objects are especially important [6]. Designers need forms of protection which make distinctions according to user profile (e.g., member of the system design team), and according to the nature of the information (e.g., execute-only access to system kernel). Integrity checks will be needed to insure that multiple design constraints are satisfied and maintained during modification. However, the concept of integrity or consistency control in the context of fuzziness and team coordination is, like concurrency control, an area for research.

- **Backup and Recovery:** Many standard backup and recovery techniques seem to apply to the Information Space. However, transactions provide the wrong granuarity for recovery via the all-or-nothing "commit" property. Since, in the design environment, transactions will frequently be long-lived and complex, it is not viable to simply undo transactions [26].

- **Performance:** Database is a performance-oriented technology, though the object-oriented and semantic models still extract a significant performance penalty. The anticipated size and complexity of *Leonardo's* Information Space should eventually make performance a major concern. The extent to which *Leonardo* will be used in a navigational mode reduces its performance requirements. However, the need for interactivity, sophisticated search and integrity mechanisms, and temporal or historical archiving pushes in the other direction. There is a lot of promising research along these lines (e.g., [8; 16; 23; 24]).

5. Toward a Comprehensive Information Management Framework

Figure 1 illustrates several areas of research that are expected to influence the development of software design information technology. Research in the area of information retrieval has produced auto-indexing [20] and full text search capabilities [8; 14] for retrieving text based design information. Recent work [18; 19] on extracting formatted data from text has started to fuse a link between information retrieval and data management technology, while recent efforts in hypertext [7] can be expected to contribute directly to the capture and retrieval of design information.

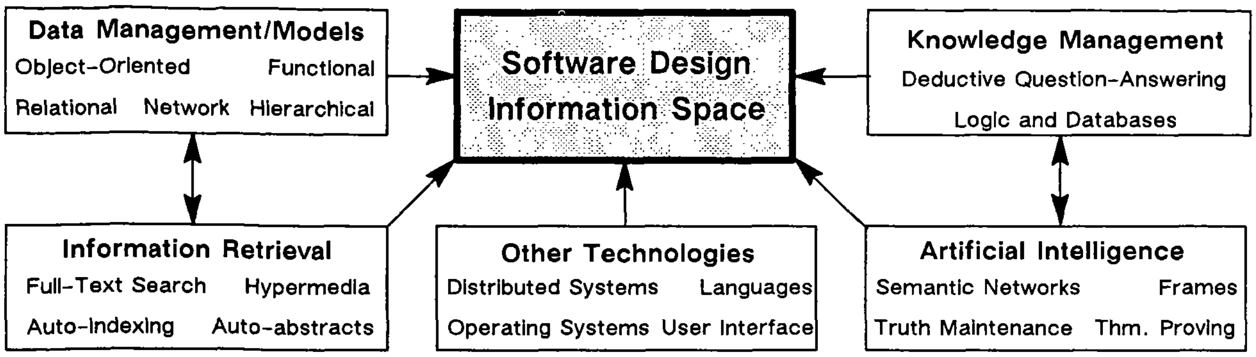

Data Management/Models
Object-Oriented Functional
Relational Network Hierarchical
Software Design Information Space
Knowledge Management
Deductive Question-Answering
Logic and Databases
Information Retrieval
Full-Text Search Hypermedia
Auto-Indexing Auto-abstracts
Other Technologies
Distributed Systems Languages
Operating Systems User Interface
Artificial Intelligence
Semantic Networks Frames
Truth Maintenance Thm. Proving

Figure 1. Software Design Information Technology

The important notion of "object" has evolved in the AI, database and programming worlds, essentially as an elegant packaging of modularization and encapsulation concepts. Research at MCC is attempting to extend the object-oriented paradigm to include much or all of the functionality associated with relational systems [1], and, reciprocally, to extend the relational model to include complex objects [27] and promote increased efficiency in rule-based inferencing over data [15]. This research strongly interacts with work on functional data models [21], knowledge representation [5] and logic and databases [12]. There are also a number of "peripheral" technologies, such as programming languages, operating systems, user interface management and distributed systems, which have had and will continue to have secondary influence on information management.

Thus we have a number of candidate technologies, which might together provide the complement of properties that we require. Each of these technologies -- hypermedia, relational, functional, and object-oriented databases, and knowledge representation systems -- has advantages and disadvantages for the Design Information Space and *Leonardo*.

Hypermedia systems provide the ability to deal with highly informal and irregular data structures, but alone, such systems provide little else, and are notably deficient in search mechanisms.

In contrast, relational database systems are strong in the area of search, in that they offer well-defined, understood search and query facilities over large-scale, but homogeneous and largely passive, databases. However, they are weak elsewhere, notably in representing the nonhomogeneous and active data elements that arise in the design process. They excel in dealing with simple, regular structures, or where simple, regular structures can be composed into large-scale structures to which regular operators can be applied in a "set-at-a-time" style of processing. In the Design Information Space, simple regular structures and large-scale regular operators are atypical.

One of the major strengths of a semantic or functional model would be its capability for dynamically representing numerous class and object dependencies and interrelationships. Such dynamics extends to the database schema level itself, or other settings where it is difficult to determine *a priori* the nature and number of relations that are associated with any particular object, as is typical during the design process.

Some object-oriented models (Cf. [1].) share this capability for dealing with irregular structures. However, object-oriented systems focus more upon information elements, their properties and their active behaviors, providing the layers of abstraction, deferral of commitment and the inheritance properties that we need. Unfortunately, object-oriented systems are also weak in the area of search.

Knowledge representation systems also deal well with irregular structures, and in addition, generally include facilites for complex pattern-based searches, inheritance and inference. These systems lack data management's concern with effective manipulation of vast quantities of data and the subsequent issues of efficiency, concurrency, persistence and robustness. The emphasis on real-world structure rather than real-world quantity has limited applications to moderate size and breadth.

In general we expect that ongoing research in AI, data and knowledge management and information retrieval will critically affect the overall architecture and development of the Design Information Space and hopefully lead to the unified model that *Leonardo* requires.

6. Conclusions

The challenge that software engineering, especially the upstream, poses to database is in the effective management of "imperfect" or "problematic" (e.g., dynamic, irregular, imprecise, informal, even inconsistent) data and the support of exploration or navigation as an aid to understanding, improving, using and reusing this data. It is virtually a truism in commercial data processing that any integrated information system is best built around a shared database managed by a proper DBMS. Obviously, we cannot restrict ourselves to standard database technology, but must also be concerned with other management technologies, such as those for knowledge and information bases, since no one of them seems to provide the needed flexibility to deal with the full scope of *Leonardo* and its Design Information Space.

Although current commercial DBMS's lack the ability to deal with the diversity, dynamics and complexity inherent in application areas such as *Leonardo*, they exhibit a robustness and attention to effectiveness that is absent in most of the other relevant technologies. However, AI, semantics-oriented, object-oriented and hypermedia technologies are beginning to mature. Moreover, in the future, it will be difficult to maintain a distinction between such technologies, as they will unite in most real applications. Further work is needed in merging these technologies and extending them to meet the needs of *Leonardo.*

References

[1] Banerjee, J., H. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou and H. Kim, *Data Model Issues for Object-Oriented Applications*, to appear in **ACM Trans. Office Info. Systems**, April 1987.

[2] Banerjee, J., H. Kim, W. Kim and H. Korth, *Schema Evolution in Object-Oriented Persistent Databases*, **Proc. 6th Advanced DB Symposium**, Tokyo, 1986, 23-31.

[3] Belady, L., *MCC: Planning the Revolution in Software*, **IEEE Software**, Nov. 1985, 68-73.

[4] Biggerstaff, T. and C. Richter, *Reusability Framework, Assessment, and Directions*, to appear in **IEEE Software**, Vol. 4, No. 2 (1987).

[5] Brodie, M. and J. Mylopoulos (eds.), **On Knowledge Base Management: Integrating Artificial Intelligence and Database Technologies**, Springer-Verlag, New York, 1986.

[6] Chou, C., W. Kim, *A Unifying Framework for Version Control in a CAD Environment*, **Proc. 12th Int'l Conf. on Very Large Data Bases**, Kyoto, 1986, 336-346.

[7] Conklin, J., **A Survey of Hypertext**, MCC Tech Report STP-356-86, October 1986.

[8] Christodoukalis, S. and C. Faloutsos, *Design and Performance Considerations for an Optical Disk-Based, Multimedia Object Server*, **IEEE Computer**, Vol. 19, No. 12 (1986), 45-56.

[9] Ege, A., and C. Ellis, *Design and Implementation of Gordion, an Object Base Management System*, to appear in **Proc. 3rd Int'l Conf. on Data Engineering**, Los Angeles, 1987.

[10] Forman, I., *On the Design of Large Distributed Systems*, **Proc. 1st Int'l Conf. on Computer Languages**, Miami, 1986, 84-95.

[11] Frankel, K., *Report on the MCC Conference*, **CACM**, 28 (1985), 808-813.

[12] Gallaire, H., J. Minker and J. Nicholas, *Logic and Databases: A Deductive Approach,* **ACM Computing Surveys**, Vol. 16, No. 2, (1984), pp. 153-186.

[13] Graf, M., *VERDI: A Visual Environment for Distributed Systems Design*, to be submitted to **3rd Int'l IEEE Workshop on Visual Languages**, Linkoping, 1987.

[14] Hollaar, L., *Text Retrieval Computers*, **IEEE Computer**, Vol. 12, No. 3, 762-772.

[15] Kellogg, C., A. O'Hare and L. Travis, *Optimizing the Rule-Data Interface in a KMS*, **Proc. 12th Int'l Conf. on Very Large Data Bases**, Kyoto, 1986, pp. 42-51.

[16] Maier, D., *Why Object-Oriented Databases Can Succeed Where Others Have Failed*, **Int'l Workshop on Object-Oriented Database Systems**, Pacific Grove, CA, 1986, 227.

[17] Marks, P., **What Is Leonardo?**, MCC Tech Report STP-141-86, April 1986.

[18] Pavlovic-Lazetic, G. and E. Wong, *Managing Text as Data*, **Proc. 12 th Int'l Conf. on Very Large Data Bases**, Kyoto, 1986, 111-116.

[19] Sager, N., **Natural Language Information Processing**, Addison-Wesley, 1981.

[20] Salton, G., *Another Look at Automatic Text-Retrieval Systems,* **CACM**, 29 (1986), 648-656.

[21] Shipman, D., *The Functional Data Model and the Data Language DAPLEX*, **ACM Trans. Database Systems**, 66 (1981), 140-177.

[22] Smith, R., R. Dinitz and P. Barth, *Impulse-86 A substrate for Object-Oriented Interface Design,* **Proc. OOPSLA'86**, 1986, 167-176.

[23] Stanfill, C. and B. Kahle, *Parallel Free-Text Search on the Connection Machine System*, **CACM**, 29 (1986), 1229-1239.

[24] Stanfill, C. and D. Waltz, *Toward Memory-Based reasoning*, **CACM**, 29 (1986), 1213-1228.

[25] Wiederhold, G., *Views, Objects and Databases*, **IEEE Computer**, Vol. 29, No. 12 (1986), 37-44.

[26] Yeh, S., C. Ellis, A. Ege and H. Korth, **Performance Analysis of Two Concurrency Control Mechanisms for Design Environments**, MCC Tech Report STP-036-87, February 1987.

[27] Zaniolo, C., *The Representation and Deductive Retrieval of Complex Objects*, **Proc. 11th Int'l Conf. on Very Large Data Bases**, Stockholm, 1985, 458-459.

THE SIGMA PROJECT AND DATABASE ISSUES

Noboru Akima
Planning Manager, SIGMA Project
Information-Technology Promotion Agency
5F Akihabara-Sanwa-Toyo Bldg.
3-16-8 Soto-Kanda, Chiyoda-ku
Tokyo, Japan

1. INTRODUCTION

SIGMA (Software Industrialized Generator and Maintenance Aids) is a project whose mission is to improve software development productivity and software quality. At present, approximately 150 companies (some are foreign based) participate in the project, including the Japanese government. The project is funded by investments from the participants, which are planned to be 25 billion Yen over five years between 1986 and 1990.

To achieve its objectives, SIGMA tries to establish and diffuse two things:

(1) standardized software development environment which is independent of target computers, and

(2) network system for retrieving and transferring programs and technical information.

The project is being undertaken by engineers sent from various companies such as computer manufacturers, common carriers, software houses, etc. The number of these engineers is nearly 50, and they are technically in charge of running the project; selecting the most suitable technologies, managing the jobs contracted to outside companies; and so forth. Besides these engineers, the SIGMA System Development Committee, consisting of professionals and representatives from various fields in industries, is in place to advise the project. Two other committees, the Steering Committee and Technical Committee, are to support the System Development Committee.

2. SIGMA NETWORK

One of the means to achieve the project's objective is to establish a computer network. This network provides services to SIGMA user's electronic mail, bulletin boards, electronic conversations, file transfer, remote job entry, virtual terminal functions, etc. These services are provided on DDX-P (Digital Data Exchange - Packet switching) by NTT, the largest common carrier in Japan. Within a company or an organization, LAN (IEEE802.3) will be used. Connecting service with public telephones is also considered.

3. SOFTWARE DEVELOPMENT ENVIRONMENT

Software development environment proposed by the SIGMA project is one which can be used to develop computer programs for various target machines. This environment consists of personal workstations, software development tools which enable software engineers to computerize their activities and eventually improve productivity and quality. To make the tools portable, all the tools will be written in C programming language.

4. SIGMA OPERATING SYSTEM

Standardizing programming language is not enough to assure portability of tools. Therefore, the common operating system interface is defined. Unix[1] is chosen as the base for this operating system. The common operating system has been named SIGMA O/S. Unix is chosen because of its superiority for software development and relative hardware-independence. The SIGMA O/S extensions, described below, are those necessary for software development and for assuring the portability of tools. At present, elementary functions have been defined for developing the first version of O/S and tools. A detailed specification for further extensions is still on-going.

(1) *Japanese language processing capability.*
This capability is essential for software development in Japan.

(2) *Network function.*
Network functions are enhanced and protocols are defined to enable all SIGMA users to communicate. The protocol basically uses TCP/IP.

(3) *Graphics.*
Recent software tools are supposed to have graphics capability to manipulate charts, graphs, tables, etc. The graphics interface is based on GKS.

(4) *Multi-window.*
Multi-window capability is also indispensable for the software development workstations.

5. DATABASE MANAGEMENT SYSTEM (DBMS)

The volume and types of data to be processed on SIGMA O/S will vary with the magnitude of the software developed and the development method used. In the SIGMA project, we expect roughly five types of data.

(1) *Data Dictionary.*
Data for managing data resources. This contains the names, attributes, and information related to software resources.

(2) *Software Data.*
Data for managing software development. This includes results or intermediate results generated during software development. These include project management data, system configuration data, program specification data, source/binary, document and test data.

(3) *Common Data.*
Data for re-use of software resources. This includes terminology dictionary, common code representation, and program modules.

(4) *Tool Data.*
Data on using the SIGMA system. This includes the tools available to SIGMA users and guides for their usage.

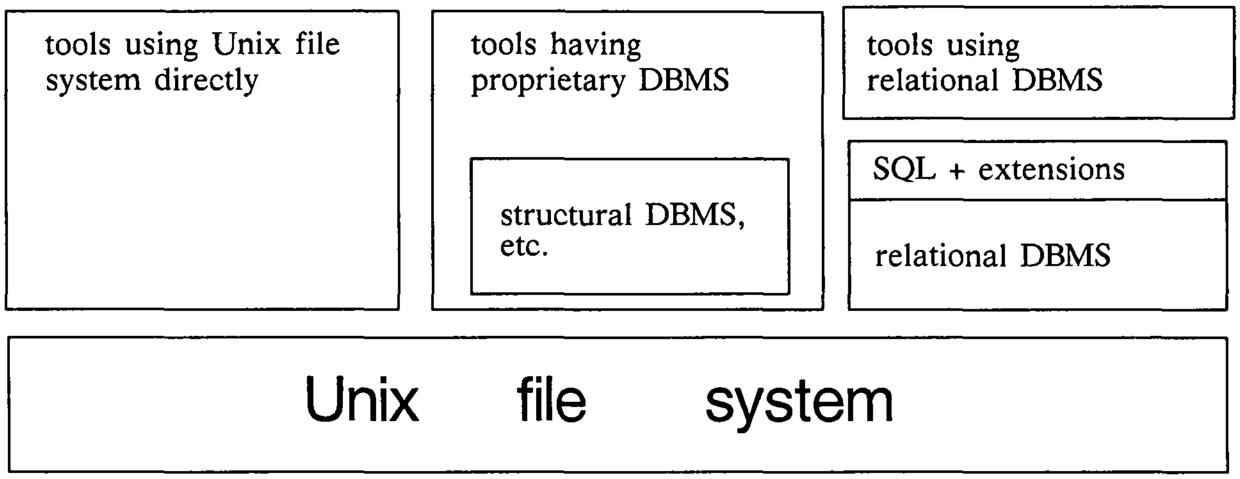
(5) *Fact Data.*
Data for productivity guidelines and future enhancements. This contains logging data, data of resources spent for development, productivity indexes, and data for quality control.

Today, there is no database system which can deal with all these different types of data. The current relational DBMSs are adequate for handling data dictionary, project management data, and fact data. Extensions to the current relational DBMSs are necessary for the common data, tools data, software

[1] Unix operating system is developed and licensed by AT&T.

development results and documents, which deal with very long unstructured data such as text and binary data.

Recommended DBMS specification for SIGMA is SQL specification of the ISO international standard and some SIGMA extensions which are necessary for SIGMA usage and SIGMA tools. However, there are many Unix tools which do not run with DBMSs. Since one of the most important objectives of the SIGMA project is to assure portability of tools on various workstations with different makes of SIGMA O/S, SIGMA allows tools to interface with different kinds of DBMSs, rather than specifying a single DBMS, as shown below. In particular, SIGMA leaves the choice of access methods and data models to tool developers.


tools using Unix file system directly
tools having proprietary DBMS
structural DBMS, etc.
tools using relational DBMS
SQL + extensions
relational DBMS
Unix file system

For the first version of SIGMA O/S, in order to assure the portability of SIGMA tools, the minimum extensions to SQL language specification have been defined, i.e., standardization of data types (including extra long character string and Japanese character string), and code assignment in interface area for the C host language. Additional extensions will be further considered for later versions of SIGMA O/S, depending on the progress of standardization of SQL/Addendum1 and SQL2.

6. SOFTWARE DATA BASE (SWDB)

The objective of improving software development productivity is expected to be achieved by tools on good O/S and workstations. Tools should work systematically and give the power to software engineers to raise productivity all through their work.

Activities in each phase of the software life cycle are a series of processing and modifying of various types of information associated with the software and software development. Supporting these activities through the use of computers requires integrity of this information, and they should be passed over through each phase without conflicts. To establish an integrated environment which supports the requirement analysis phase down to operation/maintenance, a mechanism is needed for managing the information generated, referred and modified as the life cycle proceeds through different phases. SWDB is to manage this information systematically and effectively. SWDB is to be used directly by software engineers or indirectly by tools. The first version of the SIGMA system, which is scheduled to be test-released to the participants in late 1987, will not provide this mechanism. The project has two and one-half more years to enhance and complete this task.

In the environment provided by SIGMA, software development is to be done on workstations connected through a LAN. Under this environment, we are evaluating the following candidate architectures for managing SWDB and personal files.

(1) *Central Server and Diskless Workstations*

All information exists on a central SWDB server. The workstations do not have local database support. Requests for information from workstations are sent to the SWDB server via LAN. This is good for data security, but will suffer from performance problems when the number of workstations and frequency of access increase.

(2) *Workstation Databases and Central Server with Shared Database*

Software development is done on workstations with local database support, and the SWDB server manages only data to be shared among the workstations. Workstations request the SWDB server for necessary information which is not on the local workstation. This architecture should be good for performance. However, updating of the Data Dictionary on the workstations and the server will require careful control, and the users will need to be aware of the distribution of versions of libraries and files.

(3) *Workstation Databases and Central Server with Full Database*

The SWDB server maintains all information. However, each workstation has its private DBMS and copying is done automatically, as in the virtual memory mechanism. When changes are made to the private database, they are written back to the master SWDB. This architecture allows a greater degree of sharability of data among workstations, since the server will have full copies of all workstation databases. However, it will require careful control to maintain replicated private and master databases. Further, performance will not be as good as (2).

7. LIBRARY MANAGEMENT TOOL AND DATA ARCHITECTURE

Until we implement SWDB, we will provide the Library Management Tool (LMT) and Data Architecture (DA). DA will integrate tools. It defines input and output (intermediate) of tools, enabling flexible combination of tools. DA consists of Control Information and Data Structure. Control Information is used to categorize and systematize the DA itself; while Data Structure consists of semantics information, plus the method for combining tools. DA makes combining tools easier, as well as maintaining flexibility for the tools provided by many tool vendors.

LMT will support the management of design documents, source programs, object programs, test data, JCL, intermediate results, etc., which are produced during software development. Basic functions of LMT are access control, version management, retrieval, allocation and back-up of libraries. It also has a function for configuration management. Load modules can be created using source file, macro file, subroutine libraries and the information of these relations. Engineers access these objects and information using logical names of program, module and/or type of programming language. Accessing by logical names, engineers are less annoyed where these objects and information physically reside.

LMT is further divided into some groups of tools:

(1) Library system management tools.
(2) User registration tools.
(3) Member registration tools.
(4) Member access tools.

(5) View registration tools.
(6) Member retrieval tools.
(7) Configuration control tools.
(8) Version control tools.
(9) Status control tools.
(10) Reporting tools.
(11) Target machine related tools.

LMT manages the following types of information:

(1) Library system management information
(2) Library system configuration information
(3) User information
    -user -group -manager
(4) Member information
    -member identification -access qualification
    -version -status -keyword
(5) Results management information
    -configuration management -version management
    -status management
(6) View information
(7) Retrieval information

8. CONCLUSION

At the present time, the SIGMA project has spent one and one-half years designing the first version of the system. This test-version will be completed within one year from now, and the next stage for modification and enhancement will follow. The proposed SWDB will be fully implemented during the next stage. The concrete efforts for implementing SWDB, including a preliminary design, have already started, and LMT, DA, and other tools, such as project management tools, will be integrated into this concept.

The fundamental policy of the SIGMA project is to integrate existing technologies into a practical system, as the system is expected to be in use by businesses after the five-year development phase. Therefore, the technologies introduced into the system should be the best ones, as well as those that would be widely accepted by engineers and management. The selection criteria are being carefully considered. The project is named so as to represent the "total sum" of everyone's cooperation.

DAMOKLES – the database system for the UNIBASE software engineering environment

Klaus R. Dittrich   Willi Gotthard   Peter C. Lockemann

Forschungszentrum Informatik an der Universität Karlsruhe
Haid-und-Neu-Straße 10-14, D-7500 Karlsruhe 1

Abstract

UNIBASE is a major joint project of a number of German companies and research institutions to produce an integrated software engineering environment. There is little dispute these days that powerful data management features are needed at the lower levels of systems like UNIBASE. Likewise, it has become clear by now that traditional database systems are in several respects inappropriate to do the job. The *DAMOKLES* database system used in UNIBASE therefore has to solve a number of problems differently than classical systems.

This paper gives a coarse overview of the UNIBASE project. It then provides a short rationale for database support in this environment, lists the most salient requirements and briefly describes some of the highlights of *DAMOKLES*. Finally, the current state of the system is reported.

1. The UNIBASE project

UNIBASE is one of four projects supported by the German Ministry for Research and Technology that aim at the development of integrated software engineering environments (SEEs). While different in their detailed emphasis, all projects are done in close cooperation between software companies and research institutions. UNIBASE includes four partners from each side. A total of some 60 professionals is involved over a four year period (1985 – 88).

The overall UNIBASE architecture (figure 1) consists of a uniform user interface, *DAMOKLES* as the common underlying database system, and a number of tools that together support the various phases of the software life cycle in between. A number of these tools are supposed to provide – together with the user interface and the database system – the infrastructure of a software engineering environment; they include basic facilities for

- document management (as far as beyond the scope of the database system),
- project management,
- design procedure management.

All the tools (including e.g. editors, compilers, design tools etc.) are thought to be selected as desired by the individual installation and "plugged into" the general framework. Of course, it is not sufficient to provide the technical means for integration, and thus a number of tools really have to fit together in order to reach the environment aimed at.

UNIBASE is designed and implemented for the UNIX operating system and its look-alikes. However, there are only few dependencies on it and thus other operating systems are not beyond reachability.

2. Rationale for and requirements of database support

From an information management viewpoint, software engineering tools generate, transform or analyse *documents*. Some of these documents are part of the product to be developed (e.g. source and object code, manuals), others represent intermediate results of the design process and may be used as starting points for eventual design iterations. A document is often called a *representation* of the software product being designed (the *design object*).

The management of a potentially large number of design objects in a software engineering enviroment has to fulfill numerous requirements. First, abstracting from the document contents itself, the following major problems have to be dealt with:

- Various interrelationships, like e.g. "depends on" or "has been constructed from", exist between the different representations of a design object and have to be managed consistently.
- In the design of software systems, engineers should try to reuse (parts of) existing systems and modules where appropriate. Thus, libraries of reusable software components and simple ways for accessing them and selecting from them have to be maintained.
- Software systems are complex systems that are always decomposed into manageable parts; their structure usually shows some sort of hierarchy, which should be reflected and exploited in the management of design objects.
- As a result of the development of alternative solutions for a given task, or of revisions due to changing requirements, error corrections and so on, the "same" document exists more than once, though in slight variations. Thus, we have to deal with *versions* of representation objects. Eventually, the designer has to choose a *configuration* of the (sub-) system by selecting a consistent set of versions, one for each representation needed.
- The production of large software systems usually is a team effort. Management of the design documents, therefore, has to provide for the controlled cooperation of a number of people, which entails proper synchronization, access control and the supervised exchange of both, completed and "in work" documents. Also, recovery capabilities in cases of user errors or system failures are needed.

A closer look at the documents themselves reveals a second set of requirements:

- Documents often have themselves an internal structure that may again be interpreted as a large number of objects and relationships among them. Consider, e.g., an attribute syntax tree. Moreover, relationships may exist between objects located in two or more different documents.
- Other documents do not show any sort of meaningful structure at all (e.g. the decomposition of a textual description usually is not relevant for tools of a software engineering environment).
- Complex consistency constraints have to be enforced for the objects and relationships of one document and across documents (e.g. entities imported by one module have to be exported by some other module).

Not surprisingly, this list of requirements does not look all that different from those for other design areas like CAD for VLSI-design or mechanical engineering [Lock85]. In all these cases, powerful yet efficient information management mechanisms are needed as basic components in order to relieve tools from trying to meet all mentioned requirements themselves. Business and administration applications (e.g. banking, payroll processing, inventory control, airline reservation) over the years have come to appreciate the features of database management systems (DBMSs). The same features should be attractive for use in SEEs, too:

- data integration (single, standardized data mangement and retrieval interface for all tools)
- application-oriented (in contrast to machine-oriented) concepts for structuring and accessing data: the database thus captures more of the application semantics instead of hiding it within the application code

- consistency control
- multi-user operation (synchronization)
- recovery
- authorization and access control
- data independence (each tool has only the view of the database it needs, and thus remains immune against structure changes caused by others, changes in storage management techniques, changes in hardware devices, etc.)

Previous approaches to software engineering environments tended to use conventional file systems to store their documents. As file systems do not offer most of the features just mentioned, additional management components had to be developed. They usually concentrated on version and configuration control [Tich85] and are not built to deal with all the other requirements.

Summarizing and interpreting the requirements gathered in the previous section from a database system point of view, we obtain the following list:

- The data model should allow the declaration and manipulation of complex objects (accounting for their elaborate internal structure) and of arbitrary inter-object relationships; object versions should be supported and a special domain type for the representation of unstructured information should complement the usual standard types.
- Complex consistency constraints have to be supported that may be checked at arbitrary times; in addition the reactions to constraint violations should be explicitly definable.
- Long transactions to model meaningful units of work in the software development process should use non-suspending synchronization techniques; special recovery features should prevent the loss of results even though the transaction may not yet have been committed.
- Authorization and access control techniques should be tailored to the objects supported by the data model.
- Libraries of predefined design objects, products currently under development and private data of the individual engineers should be accessible in a uniform way.
- A hardware architecture comprising a network of workstations and possibly a database server should be supported.
- Performance of the whole system should be high enough not to hamper the work of the software engineers.

Several projects tried to solve the data management problems of software engineering environments and other design systems by using traditional database technology [Habe82, Lint84, Nara85]. Not surprisingly, they experienced major problems, mainly emanating from the data model [Sidl80]. In a nutshell,

- simple record-oriented data models (e.g. relational, network) have too little expressive power to conveniently deal with complex object-oriented application semantics; consequently, database descriptions and access programs tend to become very obscure and tedious to handle,
- system internals are tailored towards flat records or homogeneous sets of records; thus, performance of object-oriented operations becomes extremly poor (recent results in the area of geometric modelling by [Härd86] show that one object-oriented operation may cause around 70 record-oriented operations in a network database system).

Augmenting a conventional database system by putting a more appropriate interface on top ("front-end") will cure the first problem, but leaves the second one unchanged. Another popular solution, keeping documents in files and using a conventional database system as a manager of the administrative information associated with them [Nara85], addresses only part one of the requirements (namely management of interdocument relations) and even introduces new problems with regard to the controlled cooperation between the file system and the database system.

3. The DAMOKLES approach

Our design of the *DAMOKLES* system has been guided by two objectives, namely

- to incorporate enough functionality to provide efficient support for the requirements of software engineering environments, but also
- to be general enough in order to support potentially arbitrary environments and to avoid a system that would be overloaded with concepts and thus too complicated to apply.

We thus decided to provide a number of rather general basic concepts that may be further refined by additional levels of tools within the SEE. For example, the UNIBASE document management provides a more specific concept of design documents, including specific version semantics that are based on the *DAMOKLES* data model concepts.

In this chapter, we present the *DAMOKLES* design object data model (DODM) and briefly touch some other system aspects. [Ditt86] provides more details and examples.

3.1 The design object data model

From a bird's eye view, the DODM tries to achieve this balance by providing for

- ***structured*** (or ***complex***) ***objects*** that may have ***versions***,
- ***relationships*** between objects and/or their versions,
- ***attributes*** that associate further information to objects and relationships in a more or less structured way.

DODM may be characterized as to belong to the entity-relationship class of data models, but its expressive power goes far beyond the classical approaches of this class [Chen76, ISO82]. In the sequel, we discuss each of the above constructs in turn.

Structured objects

A ***simple*** DODM object, like in classical data models (then called a record or a tuple) is composed of a number of ***attributes*** much like a record in programming languages. One or more of these attributes may be designated to be the object ***key*** and is thus required to be unique within the set of current database objects of the respective type, at any point in time. In addition, *DAMOKLES* automatically assigns a unique object identifier (OID) to any object upon its creation. Attributes form the ***descriptive part*** of an object. ***Structured objects*** also consist of a ***structural part***: it includes a set of ***subobjects*** (perhaps with relationships among them) that, in turn, are objects in their own right and thus may themselves be simple or structured.

In the database schema, the descriptive part of an object type is specified as usual by enumerating the desired attribute names and their associated value sets. For the structural part (if any), the type names of desired subobjects are listed. As there are no further restrictions,

- recursive objects may occur by (directly or indirectly) using their own object type within their structure,
- structured objects need not always be simple hierarchies of lesser objects but may overlap in arbitrary ways.

Instances of structured object types originate in two ways. First, an object of a subobject type may be created together with a given instance of (one of) its superobject types. Second, an existing subobject may dynamically be inserted into its superobject. Subobject removal, automatic subobject deletion upon superobject deletion are also supported. Operators for objects further include

- locating in sequential order the objects of a given type,
- retrieval based on object identifiers or on attribute values (using a *cursor*, to be discussed in more generality later),
- individual or joint attribute retrieval and modification,
- navigation within structural objects to the next subobject of a given type,
- locating the next structured object in which a given object participates (remember that structured objects may overlap),
- copying object attributes or entire objects.

To illustrate the object concept introduced so far, consider the following rather simplified example of describing the source code of programs (figure 2). A program has a name, an author, and various other attributes. It may consist of a number of subprograms that may again contain subprograms. Moreover, subprograms of common interest are collected into a library. Figure 2 shows an excerpt of a database schema and a graphical representation using DODM concepts, together with a sample database adhering to this schema.

Object versions

Object versions allow to represent multiple instances of the (semantically) same object under the auspices of the DBMS; they offer a basic mechanism to deal with revisions and alternatives [Ditt87]. The main characteristics of the DODM version concept are as follows:

- Versions are always associated with objects; more precisely, each version belongs to exactly one object, its *generic object*.
- Both, the generic object as a whole and its individual versions may have attributes and an internal structure. While the object attributes and internal structure are supposed to be common to all its versions, the version attributes and the composition from subobjects may differ.
- Generally, versions can be treated as objects in their own right. Linguistically, their type is denoted as <object type name>.VERSION with <object type name> being the type name of its generic object.
- Consequently, all versions of an object have the same kind of structure and the same attributes. They may even have versions themselves. Both, the entire object (with or without all its versions) or individual versions may be referenced.
- Among the versions of one object, an implicit predecessor-successor relationship is maintained which may optionally be linear, treelike, or acyclic; versions are numbered in creation sequence.
- Operators on versions allow to sequentially locate versions in the version graph for a given object where the order is determined by the graph structure or by version number, to locate the generic object of a given version, to insert and remove a version into/from the version graph of a generic object.

Figure 2 extends the example schema of figure 1 to include subprogram versions. Note that both programs and libraries contain subprogram versions only and not generic subprogram objects. However, the data model alone (at least as far as discussed up to now) does not guarantee that exactly one version of a subprogram is part of a program or library object (in fact, this property is essential for programs but need not apply to libraries).

Relationships

Relationships are n-place ($n \geq 1$) bidirectional associations of objects. Each place is characterized by a *role attribute*, and relationships in their entirety may possess further attributes. Similar to objects, the database schema describes relationship types with corresponding instances in the database. As an inherent consistency constraint, the user may specify for each role a minimum and maximum *cardinality* for each role. It defines how often an object at least must or at most may participate in a role of a given relationship type.

Relationships play a major part in defining structured objects: like subobjects, they may be included in them. Obviously, objects of any level, generic objects and individual versions may all be used to define relationships. Thus all kinds of inter-object or intra-object associations may be defined, and a powerful set of operators (similar to those for objects) allows to exploit them.

Attributes

DODM provides several predefined value sets for object and relationship attributes (e.g. for integer, boolean, character and string values). Also, a number of value set constructors exist (subrange, enumeration, array and record type). A predefined value set deserving special attention is LONG_FIELD. Long fields [Hask82] are byte strings of arbitrary length that are used to represent document contents without making its internal structure known to the DBMS. Long field operators provide for their manipulation in a way similar to direct access methods for files.

Further concepts

DODM includes a number of additional features we cannot discuss here in detail for lack of space. For example, it provides for ***referential integrity*** [Date81]: a relationship is automatically deleted if one of the participating objects is deleted. Another example is ***cursors*** that collect a number of otherwise distinct objects and/or relationships from the database and temporarily hold them as a unit for computation by an application program. The contents of a cursor is determined by a complex search expression that incorporates associative and structural criteria. Cursors can be introduced at will; they may be filled with contents determined by a search expression; the contents may subsequently be ordered on the basis of a sort expression; the cursor may be emptied of its contents. Usually, cursors are lost after session termination. However, they may be saved across sessions and subsequently be restored; such permanent cursors must explicitly be removed.

The contents of cursors can be manipulated on an element-by-element or a set basis. Operators in the first class include those for inserting or removing an existing object or relationship into/from a cursor and for sequentially navigating through the cursor. Operators in the second class are the classical set operators of union, intersection and difference.

Discussion

Principally, the object-subobject structure of complex objects as well as object versions may be regarded as nothing but special cases of general relationships. There are at least two reasons that justify the dedicated concepts we chose for DODM:

- Since complex objects and object versions are standard requirements for the application area of interest, it is more convenient for the database designer to have appropriate data model counterparts. Also, as experience shows, the schema becomes much more comprehensible than a pure entity-relationship schema.
- As the DBMS knows about the special semantics, it can provide efficient implementations for complex objects and versions (e.g. delta storage [Dada84, Tich85] in the latter case).

3.2 Other system features

Of course, there are more concepts in a database system than just the data model. Most of them have to be reevaluated for DBMSs that are to be used in design environments because the classical solutions are not well-suited. We sketch just a few of the techniques we pursue in *DAMOKLES.*

Multiple databases

The classical concept of a single integrated database is inappropriate for design applications. For one, design processes involve a high degree of trial and error, return to earlier stages, test of hypotheses, and they may extend over days, weeks or even months. Consequently, design data often tend to be transient, volatile, tentative, and tied to individual designers. Such data – usually expressed by the notion of revision – should be kept in the private databases of designers. Only design data that have been released should be transferred to public databases which may themselves be organized on two or three levels such as team databases or a project database [Klah85]. Once designs have been completed and are not subject to further modification or even maintenance, they may be transferred to archives. Software objects that are suited for reuse in other projects or for separate marketing, or that have been acquired from outside sources, may be kept in separate libraries. Consequently, an SEE will usually deal with several databases.

DAMOKLES handles multiple databases, subject to the following rules:

- An object is a member of exactly one database; however, copies are allowed in other databases.
- Relationships may be established between objects in different databases. However, the relationship attributes are confined to one of the involved databases (figure 4).
- All databases satisfy the same (global) DODM schema; however, the schema of an individual database may be just a part of the global schema. We note in passing that subschemas (views) may be defined as in conventional systems.

Long transactions

Transactions in traditional DBMSs suppose that work units are of short duration and involve small quantities of data. Thus, the traditional mechanisms implement synchronization strategies on the basis of suspending transactions, complete rollback on transaction failure, and the like. By contrast, work units in design applications take considerable time and may involve complex data structures. Appropriate mechanisms for these ***long transactions*** [Hask82] are based on the check out/check in-paradigm: to start a long transaction, the user must check out the desired objects to his private database; after completion of his work, he checks them back in. In the meantime, others may still read the (previous state of) the objects in the source database, but cannot check out these objects themselves. However, they may always create new versions (which incidentally is what they do when working without database support; note, therefore, that versions should only be entered into the database in a well controlled way).

As for recovery, we provide a save point facility that allows to define application-specific database states a designer can reset his transaction to (instead of losing the results of a longer working period as would be the case with present-day transaction mechanisms).

Other features

To provide for the cooperation of software engineers in teams, we prefer to use an extended ***access control facility*** based on the concept of structured objects (including individuals, groups, and "public" as subjects) instead of exploiting complexly nested transactions [Kim84]. A basic mechanism to be used, among others, for the enforcement of ***complex consistency constraints*** (outside the data model) is also included [Ditt85]. It relies on arbitrarily definable events that trigger user-defined actions. This allows to check for consistency whenever appropriate, and to react to violations of constraints in a dedicated manner.

4. Conclusions

A prototype system supporting the full DODM has been completed just recently (January 1987). First experience has been gained by some companies who developed DODM schemas for various software tools. Not surprisingly, it took some effort to teach the concepts to people that were not used to think in terms of data models (incidentally, something that has also been observed for the pure entity-relationship concept!). However, the examples completed to date did not show any lack or superfluity of DODM concepts, and after some training, engineers seem to do a good job in using the data model.

Up to now, there is little experience in using database technology in software engineering environments. With our experiments, we hope to be able to demonstrate that these systems and in turn their users can really benefit from incorporating an adequate DBMS.

While there are already advantages in integrating today's tools (even if they can hardly exploit the facilities for internally structuring the objects), we foresee major simplifications in the construction of future tools:

- Revision control tools and the like may become integral parts of the DBMS instead of being placed on top and thus render a really uniform object management interface for the builder of the "real" life cycle-supporting tools.
- Where appropriate, tools may conveniently use *common* database structures instead of redundant file contents that need additional transformation. In addition, the definition of complex data structures can be shifted to the DBMS (where it is done once) instead of repeating it in numerous tools.
- When defining object structures in detail to the database, DBMS mechanisms may be used to collect useful statistics (software metrics, [Perl81]), enforce consistency constraints (e.g. "import quantities of a module have to be exported by some other module") and so on.
- Moreover, since a system like *DAMOKLES* also allows to define schemas resembling those of classical commercial DBMSs (namely by just not using complex objects, versions, etc.), the same DBMS may even be used as part of a software system built by using the software engineering environment.

Efficient operation is the main requirement for making DBMSs viable parts of complex systems. We are sure that the techniques currently developed by the database research community are promising steps towards this goal.

5. References

[Chen76] Chen, P. P.-S.: *The Entity-Relationship Model — Toward a Unified View of Data.* ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976, pp. 9-36.

[Dada84] Dadam, P.; Lum, V.; Werner, H.-D.: *Integration of Time Versions into a Relational Database System.* Proc. VLDB 10, 1984, pp. 509-522.

[Date81] Date, C. J.: *Introduction to Database Systems.* 3rd edition. Addison-Wesley, 1981.

[Ditt85] Dittrich, K. R.; Kotz, A. M.; Mülle, J. A.: *Complex Consistency Constraints in Design Databases.* Technical Report No. 2, FZI Karlsruhe, 1985.

[Ditt86] Dittrich, K. R.; Gotthard, W.; Lockemann, P. C.: *DAMOKLES — A Database System for Software Engineering Environments.* Proc. Int. Workshop on Advanced Programming Environments, Trondheim, Norway, June 1986.

[Ditt87] Dittrich, K. R.; Lorie, R. A.: *Version Support for Engineering Database Systems.* To appear in IEEE Trans. on Software Engineering, 1987.

[Habe82] Habermann, N. et al.: *The Second Compendium of Gandalf Documentation.* Department of Computer Science, Carnegie-Mellon University, Pittsburgh, May 1982.

[Härd86] Härder, T. et al.: *KUNICAD – Ein datenbankgestütztes geometrisches Modellierungssystem für Werkstücke.* Universität Kaiserslautern, Report 22/86, January 1986.

[Hask82] Haskin, R. L.; Lorie, R. A.: *On Extending the Functions of a Relational Database System.* Proc. SIGMOD (ACM), June 1982, pp. 207-212.

[Hend84] Henderson, P. (ed.): *Proc. ACM SIGSOFT/SIGPLAN Symposium on Practical Software Engineering Environments.* SIGPLAN Notices, Vol. 19, No. 5, May 1984.

[ISO82] J. J. van Griethuysen (ed.): *Concepts and Terminology for the Conceptual Schema and the Information Base.* International Organization for Standardization, ISO/TC97/SC5/WG3, publication number ISO/TC97/SC5 - N 695, 1982.

[Klah85] Klahold, P. et al.: *A Transaction Model Supporting Complex Applications in Integrated Information Systems.* Proc. SIGMOD 1985, pp. 388-401.

[Kim84] Kim, W.; Lorie, R. A.; McNabb, D.; Plouffe, W.: *A Transaction Mechanism for Engineering Databases.* Proc VLDB 10, 1984, pp. 355-362.

[Lint84] Linton, M. A.: *Implementing Relational Views of Programs.* In: [Hend84], pp. 132-140.

[Lock85] Lockemann, P. C. et al.: Database Requirements of Engineering Applications — An Analysis. Proc. GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft", Karlsruhe, März 1985 (in German). Also available in English: Universität Karlsruhe, Fakultät für Informatik, Technical Report 12/85.

[Nara85] Narayanaswamy, K.; Scacchi, W.; McLeod, D.: *Information Management Support for Evolving Software Systems.* Technical Report USC TR 85-324, University of Southern California, Los Angeles, CA 90089-0782, March 1985.

[Perl81] Perlis, A. et al: *Software Metrics: An Analysis and Evaluation.* MIT Press, 1981.

[Sidl80] Sidle, T. W.: *Weaknesses of Commercial Database Management Systems in Engineering Applications.* Proc. 17th Design Automation Conf., Minneapolis, 1980, pp. 57-61.

[Tich85] Tichy, W. F.: *RCS – A System for Version Control.* Software Practice and Experience, Vol. 15, No. 7, 1985, pp. 637-654.

Acknowledgement

K. Abramowicz, C. Eick, R. Längle, T. Raupp and T. Wenner together with the authors have been involved in determining the requirements and designing the DODM. We are also grateful to our project partners in UNIBASE for their support and cooperation.

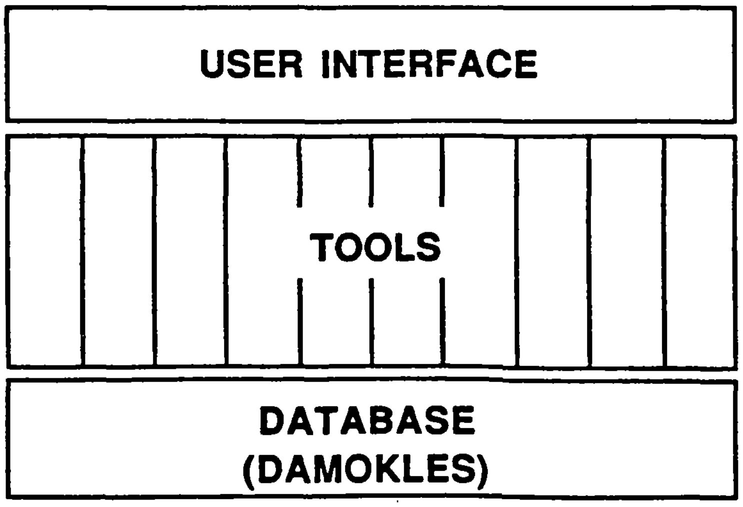
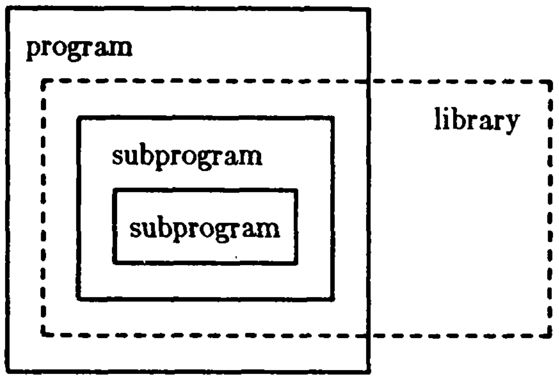


Figure 1 Overall UNIBASE architecture

```
OBJECT TYPE program
   ATTRIBUTES
      name   : STRING[30]
      author : STRING[20]
      ...
   STRUCTURE IS subprogram
END program
```

```
OBJECT TYPE subprogram
   ATTRIBUTES
      name   : STRING[30]
      author : ...
      ...
   STRUCTURE IS subprogram
END subprogram
```

```
OBJECT TYPE library
   ATTRIBUTES
      name : STRING[30]
      ...
   STRUCTURE IS subprogram
END library
```



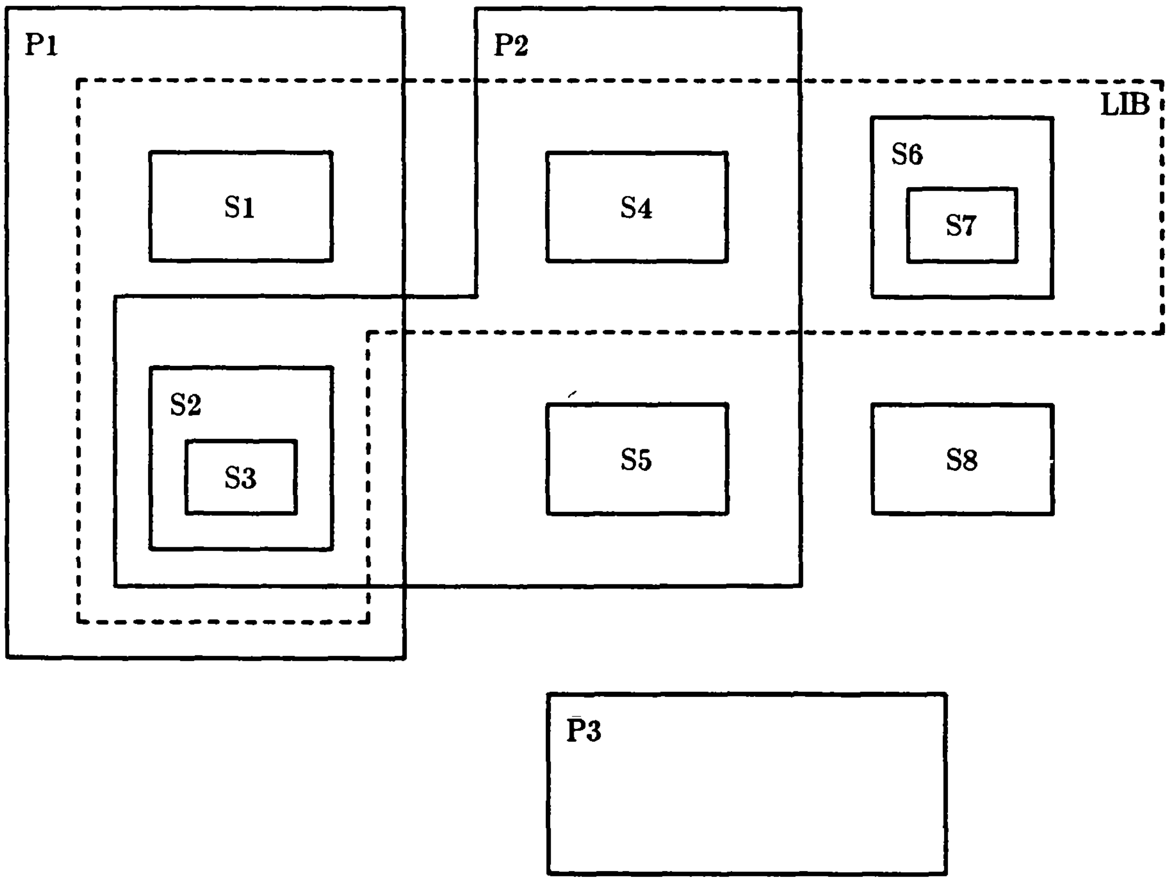


Figure 2 An example DODM schema together with a database adhering to it

```
OBJECT TYPE program
   ATTRIBUTES
      name   : STRING[30]
      author : STRING[20]
      ...
   STRUCTURE IS subprogram.VERSION
END program


OBJECT TYPE library
   ATTRIBUTES
      name : STRING[30]
      ...
   STRUCTURE IS subprogram.VERSION
END library
```

```
OBJECT TYPE subprogram
   ATTRIBUTES
      name   : STRING[30]
      author : ...
      ...
   VERSIONS LINEAR
      ( ATTRIBUTES
           vers_name   : STRING[30]
           vers_author : ...
           ...
      )
   STRUCTURE IS subprogram.VERSION
END subprogram
```

Figure 3 An example DODM schema (cf. fig. 2) involving versions

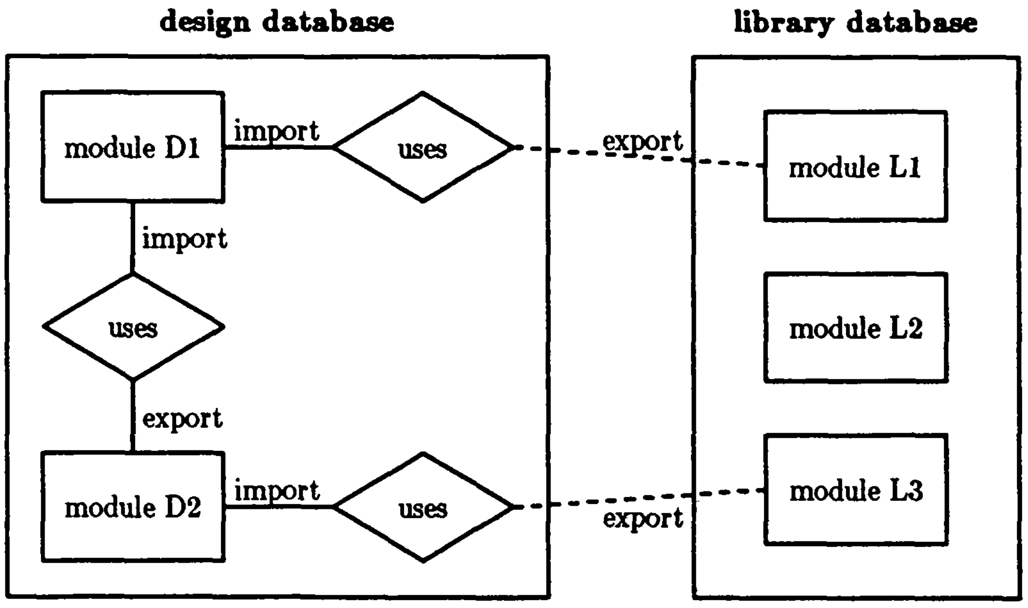


Figure 4 Relationships extending across database boundaries

Database Issues in Software Requirements Development

by Terry A. Welch and Michael D. Konrad
International Software Systems, Inc.
12710 Research Blvd.
Austin, Texas, 78759

Abstract

The requirements problem for large systems is the need to ensure that the mission needs of end-users are reflected in the technical requirements descriptions which guide development. The process of definition, analysis and revision of requirements descriptions involves a support environment in which two database engineering problems arise: (1) management of the multi-dimensional relationships that exist within the requirements descriptions, and (2) support for multiple views into the requirements as an aid to their creation and evaluation. We conclude that a requirements data base is an instance of a design data base, and presents similar implementation problems.

1. Introduction

To understand the database issues involved in supporting requirements activities, we characterize the requirements process, identify the entities and relationships that need to be managed, and the database usage implied. Section 2 defines the requirements engineering process. Section 3 defines the key entities that would need to be managed by the data base that would support the requirements engineering process. Section 4 discusses data access demands during requirements development.

This discussion presents requirements development as a rather formal activity, as would be appropriate when working on a very large target system. In smaller design efforts many of the activities described here become informal, intermixed with implementation actions. Likewise, requirements need to be more completely described for large systems, while present practice for smaller systems leaves much of this data undocumented. All of the activities and data covered here, however, appear to some extent in all cases, and perhaps will be more formally approached when tools become available to reduce the human effort needed to achieve that.

2. The Requirements Engineering Process

The three major types of information we consider in the process are: Goals, Requirements, and Solution Architectures. Goals are expressions of objectives and needs, generally mission-related, and not necessarily feasible or consistent with each other. Mission users are the primary source. Requirements are a consistent set of Goals, revised so as to be feasibly realizable within the available resources (especially time, money, and expertise). A Solution architecture is a model of the target system as a composition of parts that satisfy the requirements.

We describe the process by means of a simple scenario: a "requirements engineer" is required to produce a set of requirements for a system called the "target system". The target system must support the different roles of its users and administrators. Thus there will be different expectations, or "viewpoints" of what the system should do, of how well it should be done, and within what cost.

Through interviews with target system users/administrators and through references to documentation of similar, existing systems and their environments, the requirements engineer collects and organizes information on the operational context of the target system. The resulting information forms a domain model of the environment of the target system, providing the terminology and context through which user needs can be expressed, forming Goals. For each viewpoint, there will be one set of goals, and they should be consistent and complete within that viewpoint.

Goals are often inconsistent across viewpoints or infeasible. The requirements engineer can attempt to resolve such difficulties through further user interviews. He has other methods available to him as will be discussed below. The merged revised goals define the requirements.

Through his interviews with users, the requirements engineer identifies and documents scenarios that illustrate typical target system behavior and/or desired responses to stressful input. Scenario construction and analysis may aid stating the nonfunctional requirements, in particular, performance and reliability. Scenarios also become part of the requirements.

During the creation of goals and requirements, their consistency and completeness can be checked, to detect internal contradictions and lapses in coverage. While tools to aid in this will be under development in the future, these checks are presently manual operations. The requirements engineer might perform a walk-through, analyzing the dataflows and/or stimuli/responses through the various viewpoints.

At this point, the requirements engineer might construct a solution architecture and perform analysis on it to gain better insight into: target system interfaces, functions, performance and reliability, and implied development cost and risk. The requirements engineer creates a solution architecture by specifying how the target system is composed of parts (e.g. objects, functions) and how those parts use resources (e.g. people, software, hardware). Reuse of prior designs and resource models, with modifications, will often simplify this process.

From the solution architecture, the requirements engineer can specify a prototype or other simulation. He executes a prototype against canned or user-controlled scenarios, eliciting user comments on what should be changed. Simulations also provide performance and consistency information.

As a result of the insights gained through analysis and prototyping, the requirements engineer determines the revisions to be made to the requirements. There may be several iterations of prototype, analyze, evaluate and reformulate before the requirements have stabilized.

3. Requirements Data Characterizations

Requirements data can be structured in many ways, so the following characterization should be viewed as typical, not definitive. The size of the various data sets is very application dependent, with the amount of information formally stored growing non-linearly with the size of the system being specified. That is, larger systems not only have more components but they also have more levels of interconnection description so that each module can fit within the scope of human grasp. Further, each interface must become more formally defined as the development team gets larger and verbal documentation becomes less efficient.

System descriptions can occur at three levels of definition. First, context information or domain information states the common knowledge about the environment in which the system must operate. This would include usage of prior systems, physical constraints, war stories, available technology, etc. Second, the requirements statements themselves specify a region of capability which would make an acceptable system. These come at several levels of definition, reflecting goals statements, viewpoints of individual users, and system requirements as a consolidation of those various inputs. Third, a solution architecture which may satisfy the requirements is stated in terms that reflect an actual prospective implementation. Models built for simulations illustrate a form of solution description.

A system description can be viewed as having three components. First, a functionality description indicates what kind of logical results are produced by the system regardless of how it is implemented. Second, constraints on implementation are given, including timing, cost, reliability, etc. Third, scenarios of expected usage show how the system should react to typical or stressful input conditions. These three components are not orthogonal; frequently they provide alternative formulations of the same requirements. As a trivial example, consider a sort program; the functional description might say that outputs are ordered in increasing order, the performance constraint would specify an acceptable distribution of execution times, and a typical scenario shows that inputs ordered in decreasing order would be converted into outputs in increasing order.

The functionality description is, at all levels of definition (context, requirements, solution), typically a hierarchical construction reflecting multiple levels of detail. For example, in a large command and control system we expect at least six levels of abstraction from the top level of communications between major system components down to the human factors of an operator viewing a monitoring display. The functional description has three intertwined dimensions of data types, sequencing of events, and calculations; each of these is best viewed hierarchically to let the viewer determine the amount of detail visible at one time. Functional specifications may often have a graphical display, as in dataflow diagrams.

One example of functional description, the data dictionary, can illustrate some characteristics of requirements information. Type information for program variables, which helps form the data dictionary of a large application, is typically described as a hierarchical composition using three operators: record, array, and case (or union or variant or enumeration, depending on your favorite language). At the functional specification level for a large system, a type description can become complex with multiple levels of description. Reports and databases typify the more complex items. Data types are defined in terms of other data types, all of which may be used at different points in calculations and in a sequence of events, so the data type hierarchy is arbitrarily interconnected with the other hierarchical descriptions of target system functionality.

The constraint descriptions also entail multiple levels of detail. For example, a performance constraint at one level is stated as an average response time, but that needs to be broken down into an ensemble of response times for different events, and allocated over a set of component delays in a specification. Delay constraints tie into the functional description in a variety of ways. These and other constraints are often vague at one level of detail and extremely specific in other places.

Scenarios follow the same pattern, being compositions of several levels of detail, which tie rather arbitrarily into the functional and constraint descriptions. For example, the story of processing a transaction has component stories such as file updates, which include recovery provisions and concurrency synchronization, which in turn have a variety of possible component scenarios. Sometimes the internal structure of a scenario description matches the functionality decomposition, and sometimes it does not.

The common ingredient seen in the above descriptions should be clear. There is a great network of relationships between description objects which is unpredictable in pattern. For example, a single data type may be related to several things: other data types, data instances, functions, scenarios, integrity constraints, etc. All of this can happen at several levels of usage, so that a data type for some particular report will be vague in a goals statement, be constrained in a requirements statement, and defined in full detail in a solution.

4. Data Access Patterns

Requirements data must support three types of activity: end-user interpretation of functionality, developer interpretation of what has to be built, and analysis activity to bridge the difference between the first two as the requirements are being constructed. We presume that these activities will be carried out by teams of people working on a network of workstations.

The user of the target system will need to see the impact of requirements statements from a variety of viewpoints which often correspond to levels of functional abstraction. For example, one type of person examines the processing of a certain type of transaction while another person is interested in the flow of multiple transactions through a specific processing step. Different viewpoints are needed to review human interfaces, system security, communications capacity, etc. These users often see the system in stimulus-response terms, asking how the system will respond if hypothesized inputs are applied.

The developer of the target system is concerned about functional complexity of system components and about implementation performance. This requires seeing all system functionality which impacts a particular facility such as a database or a processor. Critical paths for speed or reliability must be determined because those components are more difficult to build. Often these internal system structures are not prescribed by the requirements, so developers investigate a variety of partitionings of functions to find a reasonable approach.

The systems analyst must translate user views into developer views and vice versa, while maintaining the consistency and completeness of the requirements descriptions. This entails attention to system state and to data structures. Constraints on parameters such as response time and reliability must meet a variety of user goals while not making the system unreasonable to build. The analyst will use several types of tools: static analysis, such as syntax checks; report generation to extract data for different viewpoints; prototypes; simulations; etc. These serve both to debug the requirements descriptions and to translate requirements into an operational form which is meaningful to potential system users.

The clear implication of all this is the need to retrieve requirements data from a variety of orthogonal views. Each of these views will be presented as a hierarchical composition of data elements, as a way of managing complexity. For example, a discrete event simulator will need to access the interconnectivity of components in a specification at several levels of abstraction in order to relate low-level delay estimates to high-level performance goals.

Since multiple people will be working on a particular set of requirements, and changes will be constantly occurring, good facilities are needed to manage versions of components and configurations of interconsistent versions.

5. Conclusions

From a database management point of view, requirements development looks very much like other engineering and design activities. The system is characterized by multiple relationships between description objects and multiple views for data access. The descriptions are inherently hierarchical in several dimensions.

An effective query language must combine the capabilities of select, project, and relate. Select retrieves a set of objects having common attributes. Project determines the amount of related information, usually hierarchical, which is provided for each object. Relate (or join or navigate) causes a shift in viewpoint following relationships from an original object set.

The traditional data models used for commercial transaction processing have proven inefficient for design data of the type encountered in this application. A hierarchical model poorly supports access to data across multiple views. The network model does not provide easy selection of sets of similar objects. The relational model is poor at retrieving hierarchical projections. When the best features of each are combined (hierarchy, pointers, sets), the result is clearly different than any of the prior models. Interesting work is being carried out on object-oriented or semantic-net type models, and maybe these will prove more appropriate.

6. Acknowledgement

This work was supported by a Rome Air Development Center contract on rapid prototyping (No. R30602-85-C-0129), which has included intensive investigation of the requirements development process. The authors have also benefited from discussions with members of the Lockheed Software Technology Center and other members of a panel brought together to participate in this RADC effort.

7. Bibliography

1. "Requirements Engineering Environments: Software Tools for Modeling User Needs", special issue of IEEE Computer, April 1985, Rzepka and Yutaka, editors.

2. Information Management for Engineering Design, R. Katz, Springer Verlag, 1984.

3. "Engineering Data Management", special issue of IEEE Database Engineering, June 1984, R. Katz, editor.

NEPTUNE: A HYPERTEXT SYSTEM FOR SOFTWARE DEVELOPMENT ENVIRONMENTS

Norman M. Delisle
Mayer D. Schwartz

Computer Research Laboratory
Tektronix Laboratories
Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

ABSTRACT

We describe the functionality of the Neptune hypertext system, and show how it can provide a complete repository for all technical information associated with a software project. Complete version histories are maintained for all forms of documentation. Related portions of separate documents can be interconnected allowing traceability among the phases of the development cycle. Special mechanisms are provided for configuration management and supporting for collaboration among teams of software developers. Examples of several software development environments built on top of Neptune are presented.

INTRODUCTION

Our primary motivation for building the Neptune hypertext system was to provide information management support for software engineering environments. Recent proposals describing project data base support for software engineering environments [Hun81, PeS85] repeatedly state the need to logically link together documentation, and source code; the need for making annotations for recording explanations and assumptions; and the need for good version management. Neptune meets all these requirements.

Using Neptune, all documentation, code, project management information and any other data associated with a software project is stored in a single data base. The documentation typically includes requirements, specifications, designs, source code, and tests. The project data may also include informal information such as explanations about why a particular design was accepted or rejected, or assumptions made about the operating environment. Neptune provides a uniform framework for recording a complete software development history.

Neptune is based on the notion of *hypertext*. Hypertext as defined by Nelson is *non-sequential writing* [Nel81]. In a hypertext system, documents consist of a collection of *nodes* connected by directed *links*. A node by itself is similar to a piece of normal text — the links between nodes give hypertext its non-linear aspects. The nodes of a hyperdocument are not, however, restricted to be text. They can represent graphical images, combined text and graphics, digitally encoded voice, or even an animation. Links can be used to connect the set of nodes that form a document or they can be used to denote relations between portions of a document or two separate documents. For more information about hypertext systems, there are several good surveys available [Con86, YMD85].

Neptune provides several features not found in most hypertext systems. These features include version histories, attributes, query mechanisms, and a partitioning mechanism called *contexts*. This paper will focus on the functions of Neptune that specifically support software development environments. Additional information about Neptune includes a complete description of its functionality [DeS86a] and a rationale for the design of contexts [DeS86b].

CASE DATA BASE REQUIREMENTS

Traditional database management systems have weaknesses when applied to Computer Aided Software Engineering (CASE) systems. The most glaring weakness is the relative lack of support they give to version control and configuration management, though Katz and Lehman [KaL84] describe an experimental system that attacks one aspect of the version control problem. Another weakness is that the traditional models (hierarchical, CODASYL, and relational) do not map well to the kinds of data that need to be stored in a CASE system. However, the entity-relationship model, and other semantic models, seem to provide a better fit [BaK85]. At the very lowest levels a relational model can be useful, possibly at the expense of performance [Lin84].

To support a large CASE application, many different kinds of data, both textual and graphical, need to be kept. This data includes source and object code, many forms of supporting documentation and highly structured information such as symbol tables and abstract syntax trees. Perhaps no single data model can meet the diverse demands of all the different types of information associated with a software project. However, we believe that hypertext can provide an excellent coarse-grain data model for CASE systems. In particular, the Neptune hypertext system can provide for making arbitrary connections between pieces of data, for interactively viewing and traversing the hypertext database, for accessing version histories, and for supporting collaboration among development teams. Additionally, Neptune serves as a common layer on top of which a wide variety of software engineering tools can be built.

AN OVERVIEW OF NEPTUNE

As illustrated in Figure 1, Neptune is designed as a layered architecture. The bottom level is a transaction-based server named the Hypertext Abstract Machine (HAM). The HAM presents a generic hypertext model which provides storage and access mechanisms for nodes and links. The HAM provides distributed access over a computer network, synchronization for multi-user access and transaction-based crash recovery.

Additional layers of functionality are built on top of the HAM. Typically, one or more application layers are built on top of the HAM and a user interface layer is built on top of the application layers. The application layers consist of programs that automatically manipulate or transform hypertext data. In a CASE application this layer could include high level language compilers or document processors. The user interface layer can provide a windowed interface for browsing and editing hypertext data and for controlling application layer programs.

When we speak of Neptune, we are generally referring to the functionality provided by the HAM. The HAM defines operations for creating, modifying and accessing nodes, links and contexts. It maintains a complete version history of the hypergraph and provides rapid access to any version of a hypergraph. The HAM makes no restrictions about the contents of nodes; at the HAM level a node just contains binary data. Applications provide the interpretation for the data.

Associated with each end of a link is a numeric value called an *offset* . If a node contains text, the offset can be interpreted as either a character position within the contents of a node. If the node contains graphics, the offset could be interpreted as a pair of numbers representing either Cartesian or polar coordinates.

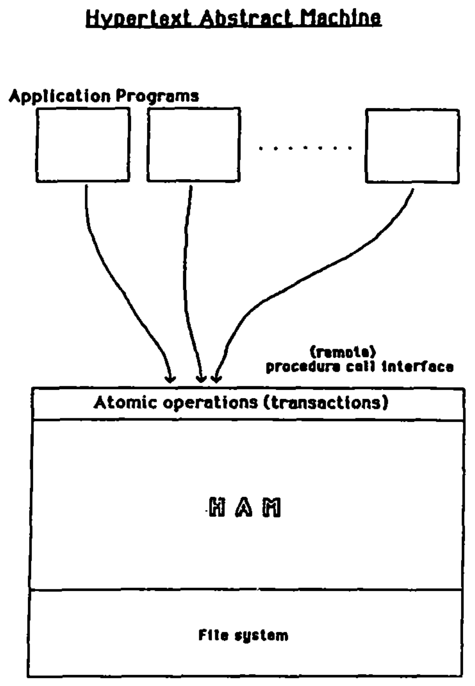

Hypertext Abstract Machine
Application Programs
(remote) procedure call interface
Atomic operations (transactions)
HAM
File system

Figure 1. Neptune System Architecture

An unlimited number of attribute/value pairs can be attached to a node, link or context. The attribute is a name; its value is either a string of bytes or a numeric value. A complete version history is maintained for attribute values.

Two basic query mechanisms are supported by the HAM: *traversal* and *filter*. The traversal mechanism, *linearizeGraph*, starts at a designated node and follows a depth-first traversal of out-links ordered by the links' offsets within the node. The filter mechanism, *getGraphQuery*, directly accesses a set of nodes and their interconnecting links. Both of these mechanisms use predicates based on attribute/value pairs to determine which nodes and links satisfy the query.

The attribute mechanism is particularly useful for building application layers. As an example, suppose a user (or an application program) adopts the convention of attaching an attribute called *document* to each node. In a CASE system its values could include *requirements*, *design*, *sourceCode* and *objectCode*. The node visibility predicate '*document = requirements*' could then be used in a *getGraphQuery* operation to access only those nodes that are part of the specification document.

Contexts are a partitioning scheme designed to support multi-person cooperative efforts. Each user can derive a private view of the hypertext graph and modifications made in this view are not visible outside the view. When a set of changes are completed, they can be released to other project team members by merging the private view with shared 'master' views. Conflicts may arise if other authors have modified the master since the time the private view was created. Neptune provides support for merging contexts including detecting conflicts and highlighting differences between contexts [DeS86b].

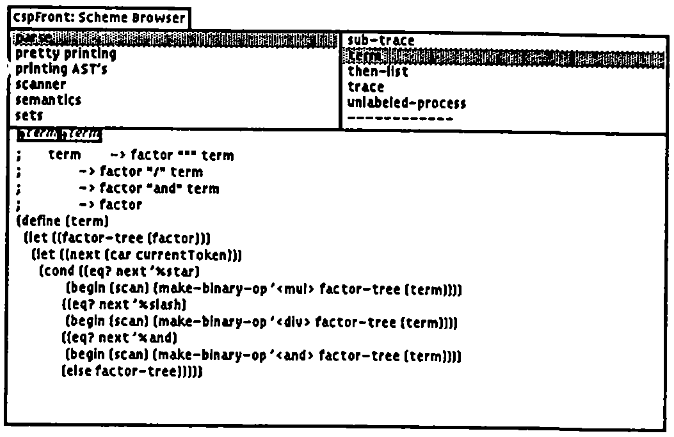
NEPTUNE-BASED CASE SYSTEMS

Two software development environments have been built using the Neptune system. The first is an environment for developing Scheme programs; the second is an enhancement to the Smalltalk-80 programming environment. In this section we will briefly describe both of these environments and show how the capabilities of Neptune were used.

A Programming Environment for Scheme

The Scheme programming environment uses Neptune to manage both documentation and Scheme source code. The environment consists of three parts: a user interface implemented in Smalltalk-80, a Scheme interpreter and the Neptune server. Typically the user interface and interpreter runs on the local workstation and the Neptune server runs on a remote host, accessed over a local area network.

Scheme is a dialect of Lisp [ReC86]. Scheme source code consists of a collection of procedure declarations. Scheme itself provides no provisions for modularization of encapsulation. To partially compensate for this language deficiency, we allow procedures to be categorized. A Scheme Source Code Browser, shown in Figure 2, supports editing of and navigating through these categories.



```
;    term     -> factor "*" term
;          -> factor "/" term
;          -> factor "and" term
;          -> factor
(define (term)
 (let ((factor-tree (factor)))
  (let ((next (car currentToken)))
   (cond ((eq? next '%star)
       (begin (scan) (make-binary-op '<mul> factor-tree (term))))
      ((eq? next '%slash)
       (begin (scan) (make-binary-op '<div> factor-tree (term))))
      ((eq? next '%and)
       (begin (scan) (make-binary-op '<and> factor-tree (term))))
      (else factor-tree)))))
```

Figure 2. A Scheme Source Code Browser

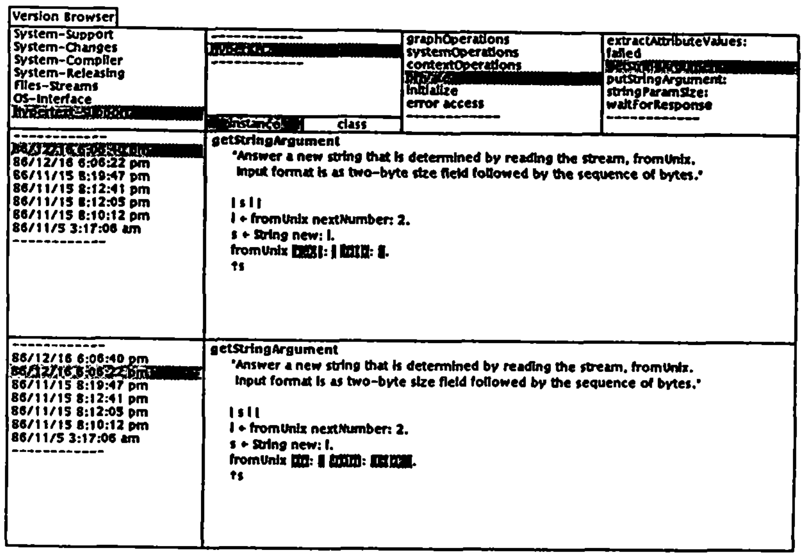
This browser has two panes at the top; the left pane contains a list of category names, the right pane has a list of Scheme procedure names. When one of the categories is selected in the left pane, a list of procedures defined in that category appears in the right pane. Categories and procedures can be created, modified or deleted using this browser.

A hypertext node is used to store each category and each Scheme procedure. Attributes are used to store the names of the objects and to identify how each object is used (i.e. whether the node is a category or a procedure). Each category node has links to all the procedures in the category. When the browser is first opened, a *getGraphViaAttributes* operation is performed that retrieves all the nodes that act as Scheme Categories (i.e. have the 'SchemeCategory' attribute defined). This operation returns a list of names of scheme categories which is displayed in the upper left pane. When one of these categories is selected a *traverseGraph* operation is performed rooted at the selected category node. This operation returns a list of names of scheme procedures which is displayed in the upper right pane. When one of the procedures is selected, its text is retrieved from Neptune and displayed in the lower pane. This pane acts as an editor, new versions of the procedure are created each time the user performs the editor's 'accept' operation.

The Scheme environment takes advantage of the hypertext model by allowing documentation to be intertwined with source code. This technique was inspired by Knuth's Literate Programming [Knu84]. In Knuth's system the documentation and code are physically intertwined in the same file and utility programs are available to extract either. Using hypertext, the documentation and code are stored independently, with links used to store interconnections; utility programs are available to combine the documentation and code. The hypertext approach has the advantage that the author can directly edit the separated views rather than being forced to deal with the physically interwined views.

A Source Code Manager For Smalltalk-80

A second CASE application for Neptune is a Smalltalk-80 source code manager. The standard Smalltalk-80 environment [Gol84] has two major deficiencies with source code management: inadequate version management [1], and no integrated mechanism for sharing source code among teams of programmers. To improve the support in these areas we are integrating Neptune with the Smalltalk-80 environment. Neptune provides storage and retrieval of versions of Smalltalk source code.



```
------------
[illegible]
86/12/16 6:06:22 pm
86/11/15 8:19:47 pm
86/11/15 8:12:41 pm
86/11/15 8:12:05 pm
86/11/15 8:10:12 pm
86/11/5 3:17:06 am
------------
```

```
getStringArgument
    "Answer a new string that is determined by reading the stream, fromUnix.
     Input format is as two-byte size field followed by the sequence of bytes."

    | s l |
    l ← fromUnix nextNumber: 2.
    s ← String new: l.
    fromUnix [illegible]: l [illegible]: s.
    ↑s
```

```
------------
86/12/16 6:06:40 pm
86/12/16 6:06:22 pm
86/11/15 8:19:47 pm
86/11/15 8:12:41 pm
86/11/15 8:12:05 pm
86/11/15 8:10:12 pm
86/11/5 3:17:06 am
------------
```

```
getStringArgument
    "Answer a new string that is determined by reading the stream, fromUnix.
     Input format is as two-byte size field followed by the sequence of bytes."

    | s l |
    l ← fromUnix nextNumber: 2.
    s ← String new: l.
    fromUnix [illegible]: l [illegible]: [illegible].
    ↑s
```

Figure 3. A Smalltalk Version Browser

Figure 3 shows a Smalltalk Version Browser. This window allows the programmer to view two versions of source code with differences highlighted. This browser is based on the standard Smalltalk-80 *system browser*. The top four panes allow navigation through the class structure; the bottom panes are used for editing and viewing source code fragments. In the version browser the bottom panes have version time lists on the left side. The code displayed in the right bottom pane is the version corresponding to the version time selected in the left bottom pane.

The Smalltalk source code is mapped into the hypertext model by separating each code fragment that can be edited independently into its own node. Thus a node contains a text fragment representing an entity such as a class definition, a method definition, a category list or a class comment. Links are used to represent the static structure of the class denoting concepts such as the protocols in a class, or the methods in a category. With this system we've built some of our largest hypertext databases filling almost ten megabytes of disk space with about 10,000 nodes and links.

Current research on the Smalltalk-80 source code manager focuses on support mechanisms for collaboration among Smalltalk programmers. We plan to use Neptune's contexts to provide both private and shared source code workspaces.

[1] Limited access to old versions is sometimes possible via the crash recovery mechanism.

SUMMARY

We have shown how hypertext can provide an appropriate coarse-grain data model for software engineering environments. We have briefly described Neptune, a hypertext system currently being used in software development environments research. We also described two examples of this research: a Scheme programming environment and a Smalltalk-80 source code management system.

Although neither of these languages is what you might call mainstream, the primary focus of both of these projects was to make the language and its associated development tools more suitable for programming in the large. The concepts used in these efforts are also applicable to CASE environments for languages like Ada or C.

In summary, hypertext provides a repository for all the information associated with a software (or hardware) project. It allows arbitrary structuring of the information, and it keeps a complete version history of the information and the structure. In Neptune, we have provided a hypertext machine that is particularly suited for building CASE systems.

REFERENCES

[BaK85] Batory, D.S. and Kim, W. Modeling concepts for VLSI CAD objects. *ACM Transactions on Database Systems* **10**, 3 (Sep. 85), 322-346.

[Con86] Conklin, J. A Survey of Hypertext. MCC Tech. Report STP-356-86, MCC, Austin, Texas, October, 1986.

[DeS86a] Delisle, N. and Schwartz, M. Neptune: A hypertext system for CAD applications. *Proc. ACM SIGMOD '86*, (May 1986) 132-143.

[DeS86b] Delisle, N. and Schwartz, M. Contexts — A Partitioning Concept for Hypertext. *Proceedings of the Conference on Computer-Supported Cooperative Work, Austin, Texas, December, 1986.*

[Gol84] Goldberg, A. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley Publishing Company, Reading, Mass. 1984.

[Hun81] Hunke, H. editor, *Software Engineering Environments*, North Holland, Amsterdam, 1981.

[KaL84] Katz, R.H. and Lehman, T.J. Database support for versions and alternatives of large design files. *IEEE Transactions on Software Engineering* **SE-10**, 2 (Mar. 1984), 191-200.

[Knu84] Knuth, D.E. Literate Programming. *Computer Journal* **27**, 2 (Mar. 1984), 97-111.

[Lin84] Linton, M.A. Implementing relational views of programs. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburg, PA, April 1984, published as *SIGPLAN Notices* **19**, 5 (May 1984), 132-140.

[Nel81] *Literary Machines*. T.H. Nelson, Swarthmore, PA., 1981.

[PeS85] Penedo, M.H., and Stuckle, E.D. PMDB — a project master database for software engineering environments. *Proceedings of the 8th International Conference on Software Engineering*, Aug. 1985, 150-157.

[ReC86] Rees, J. and Clinger, W. Revised report on the algorithmic language Scheme. *SIGPLAN Notices 1*, 12 (Dec. 1986).

[YMD85] Yankelovich, N., Meyrowitz, N., and van Dam, A. Reading and writing the electronic book. *Computer* **18**, 10 (Oct. 1985), 15-30.

INFORMATION SCIENCES - AN INTERNATIONAL JOURNAL

CALL FOR PAPERS

SPECIAL ISSUE ON DATABASE SYSTEMS

Research papers are solicited on all aspects of database management for a special issue of the "Information Sciences - An International Journal," to be published early in 1988. Topics of interest include, but are not limited to, the following:

- **Distributed Database Operating Systems**
- **Concurrency Control and Deadlock Handling**
- **Performance Analysis and Modeling of Database Systems**
- **Fault-Tolerant Databases**
- **Engineering Databases**
- **Heterogeneous/Federated Databases**
- **Object Oriented Databases**
- **Multi-media Databases**
- **Query Optimization**
- **Knowledge Bases**
- **Database Machines and Main Memory Databases**
- **Database Security**
- **Data Models**

Submit four copies of a complete manuscript by August 1, 1987 to the guest editor:

Ahmed K. Elmagarmid
Computer Engineering Program
121 Electrical Engineering East Building
The Pennsylvania State University
University Park, PA 16802
(814) 863-1047

Authors will be notified of the acceptance of their papers by October 30, 1987.

CALL FOR PAPERS

DATA ENGINEERING

Fourth International Conference on Data Engineering

February 2-4, 1988
Los Angeles, California, USA

sponsored by The Computer Society of the IEEE

COMMITTEE

Steering Committee:
C. V. Ramamoorthy, University of California, Berkeley
P. Bruce Berra, Syracuse University
Gio Wiederhold, Stanford University

General Chairperson:
Benjamin W. Wah, University of Illinois

Program Chairperson:
John Carlis, University of Minnesota

Program Co-Chairpersons:
Sushil Jajodia, Naval Research Laboratory
Iris Kameny, Rand Corporation
Roger King, University of Colorado
Z. Meral Ozsoyoglu, Case Western University
Joseph Urban, University of S.W. Louisiana

Tutorials:
Amit P. Sheth, Honeywell Corporation

Industrial and Inter-Society Coordinator:
Dick Shuey, Consultant

Awards:
K. H. Kim, University of California, Irvine

Publicity:
Jie-Yong Juang, Northwestern University

International Coordination:
Tadao Ichikawa, Hiroshima University
G. Schlageter, Fern Universitat

Treasurer:
Aldo Castillo

Local Arrangements:
Walter Bond, SDC

Committee Members (Tentative):

A. K. Arora	Robert Korfhage	J. F. Paris
J. L. Baer	Tosiyasu L. Kunii	Gruia-Catalin Roman
Farokh B. Bastani	Winfried Lamersdorf	Domenico Sacca
Don Batory	James A. Larson	Giovanni Maria Sacco
Kate Baumgartner	Matt LaSaine	Vikram Saletore
G. Belford	W.-H. Francis Leung	Sharon Salveter
Bharat Bhargava	Guo-Jie Li	Phillip Sheu
Richard Braegger	Victor O.K. Li	Edgar Sibley
C. Robert Carlson	Yao-Nan Lien	John F. Sowa
Nick Cercone	Leszek Lilien	David Spooner
David Du	Witold Litwin	David Stemple
Ramez El-Masri	Jane W.S. Liu	M. Stonebraker
Domenico Ferrari	Ming T. (Mike) Liu	Stanley Su
Hector Garcia-Molina	Raymond A. Liuzzi	Denji Tajima
Georges Gardarin	Vincent Lum	Marjorie Templeton
Robert Gerber	Yuen-Wah Eva Ma	A. M. Tjoa
Sakti P. Ghosh	Mamoru Maekawa	Mas Tsuchiya
Georg Gottlob	Sal March	Yosihisa Udagawa
Lee Hollaar	Gordon McCalla	Susan D. Urban
Yang-Chang Hong	Tadeo Murata	Patrick Valduriez
David K. Hsiao	Philip M. Neches	Yann Viemont
H. Ishikawa	Erich J. Neuhold	Kyu-Young Whang
Hemant K. Jain	G. M. Nijssen	Chao-Chih Yang
Won Kim	Ole Oren	S. Bing Yao
Dan Kogan	Gultekin Ozsoyoglu	Clement Yu
Walter Kohler	C. Parent	

SCOPE

Data Engineering is concerned with the semantics and structuring of data in information system design, development, management, and use. It encompasses both traditional applications and issues, and emerging ones. The purpose of this conference is to provide a forum for the sharing of practical experiences and research advances from an engineering point of view among those interested in automated data and knowledge management. Our expectation is that this sharing will enable future information systems to be more efficient and effective, and future research to be more relevant and timely.

We are particularly soliciting industrial contributions and participation. We know it is vital that there be a dialogue between practitioners and researchers. We look forward to reports of experiments, evaluations, and problems in information system design and implementation. Such reports will be processed, scheduled, and published in a distinct track.

TOPICS OF INTEREST

We invite you to submit papers on topics including but not limited to these:

- Applications
- Autonomous, distributed systems
- Data engineering tools
- Data management methodologies
- Data security and integrity
- Design of knowledge-based systems
- Distribution of data and knowledge
- Expert systems
- Architectures for database and knowledgebase systems
- Logical and physical database design
- Performance evaluation
- Statistical databases

PAPER SUBMISSIONS

Each paper's length should be limited to 8 proceedings pages, which is about 5000 words, or 25 double-spaced typed pages. Four copies of completed papers should be mailed before June 15, 1987 to:

John V. Carlis, Computer Science Department, University of Minnesota, 207 Church Street SE, Minneapolis, MN 55455, (612) 625-6092; carlis @umn-cs.

TUTORIALS

The day preceding the conference will be devoted to introductory tutorials. The day following the conference will be devoted to advanced tutorials. Proposals for tutorials on Data Engineering topics are welcome. Send proposals by June 15, 1987 to:

Amit P. Sheth, Honeywell Computer Sciences Center, 1000 Boone Avenue North, Golden Valley, MN 55427, (612) 541-6899.

CONFERENCE TIMETABLE AND INFORMATION:

Papers due:	June 15, 1987
Tutorial proposals due.	June 15, 1987
Acceptance letters sent:	September 15, 1987
Camera-ready copy due:	November 15, 1987
Tutorials:	February 1 and 5, 1988
Conference:	February 2-4, 1988

For further information contact the General Chairperson: **Benjamin W. Wah,** Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL 61801, (217) 333-3516; wah%uicsld @uiuc.arpa.

AWARDS, STUDENT PAPERS AND SUBSEQUENT PUBLICATION

Awards will be given to the best paper and to the best student paper (denoted as such when submitted and authored solely by students). The latter will receive the K. S. Fu award, honoring one of the early supporters of the conference. Up to three grants of $500 each to help defray travel costs of student authors. Outstanding papers will be considered for publication in the IEEE Computer Society publications: Computer, Expert, Software, and Transactions on Software Engineering. For more information contact the general chairman.

EPILOG

Several hundred people have been involved in the data engineering conferences as committee members, reviewers, authors, and attendees. We have benefited by being involved, and extend an invitation to you to participate.

THE COMPUTER SOCIETY OF THE IEEE

IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

CALL FOR PAPERS VLSI AND GaAs PACKAGING WORKSHOP

September 14-16, 1987
Research Triangle Park, NC, USA

The IEEE CHMT Society and the National Bureau of Standards are jointly sponsoring the Sixth Annual VLSI and GaAs Packaging Workshop, to be held at Research Triangle Park, North Carolina, USA. All attendees are expected to be specialists in the field and to participate in discussions.

Papers presenting new developments or critical overviews in the following areas are solicited :

- **VLSI and Wafer Scale Package Design : characterization and implementation ; cost and performance driven solutions.**
- **Package Thermal Design : characteristics, results, and issues.**
- **Package Interconnection Options : wire bonding, TAB, flip chip, or optical.**
- **GaAs IC Packaging : high speed packaging considerations.**
- **Package Electrical Issues : reduction of parasitics and improvements in electrical performances.**
- **Integrating Package Design (from die to system), including assembly and test issues.**
- **VLSI Package Materials Advancements.**
- **Die-Attach Solutions for Large Chips.**
- **New Failure Mechanisms in VLSI Packaging.**

Paper Submission

Abstracts should be sent to Arnold Pfahnl, Technical Program Chairman, AT&T Bell Laboratories, 555 Union Blvd., Allentown, PA 18103, (215) 439-6326, by May 7, 1987. Two abstracts are solicited as follows :

- Six copies of a 300-word abstract for review and paper selection by the Technical Program Committee. The abstracts, supplemented with up to four of your most important figures, will be published as the proceedings and given to all attendees.
- One copy of a 25-35 word abbreviated abstract describing the proposed paper. The abbreviated abstract of accepted papers will be included in the advance program.

Short "technology update" talks do not require a full abstract. Send a 25-50 word summary for your proposed 10-minute update to the Technical Program Chairman.

General Chairman
Archie H. Mones, Dupont

Technical Program Chairman
Arnold Pfahnl, AT&T Bell Labs

Technical Program Committee
USA Committee
George Harman, National Bureau of Standards
Phil Lutz, Semiconductor Research Corp.
John Nelson, Burroughs Corp.
Hal Seaman, IBM

European Committee
Karel Kurzweil, BULL, France
Walter Bräckelmann, SIEMENS, Germany
Jean Joly, BULL, France
Nick Chandler, MRC-GEC Research Laboratories, United Kingdom

Note:
Research Triangle Park is adjacent to the Raleigh-Durham, North Carolina airport.

PLEASE CIRCULATE TO INTERESTED ASSOCIATES