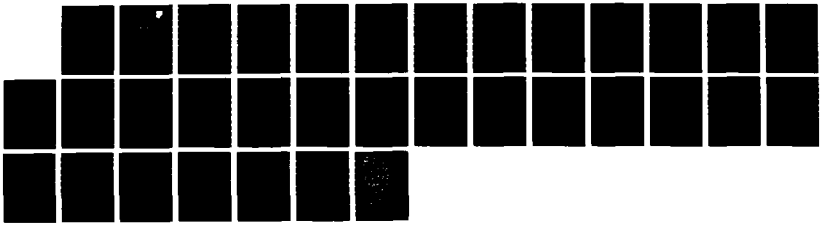
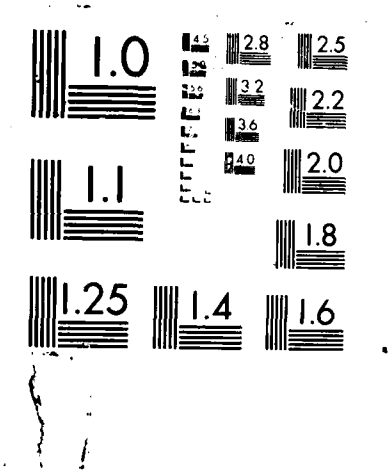
AD-A189 286 KBSA PROJECT MANAGEMENT ASSISTANT VOLUME 1(U) KESTREL 1/1
INST PALO ALTO CA R JUELLIG ET AL. JUL 87
RADC-TR-87-70-VOL-1 F30602-84-C-0109

UNCLASSIFIED F/G 5/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

DTIC FILE COPY

AD-A189 286

RADC-TR-87-78, Vol I (of two)
Final Technical Report
July 1987

KBSA PROJECT MANAGEMENT ASSISTANT

Kestrel Institute

Richard Jüllig, Wolfgang Polak, Peter Ladkin and Li-Mei Gilham

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC
ELECTE
DEC 2 9 1987
H

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

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

Although this report references RADC-TR-87-78, Vol II which is limited distribution, no limited information has been extracted.

RADC-TR-87-78, Vol I (of two) has been reviewed and is approved for publication.

APPROVED: Douglas A. White

DOUGLAS A. WHITE
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Richard W. Poulio

RICHARD W. POULIOT
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS N/A
2a SECURITY CLASSIFICATION AUTHORITY N/A	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5 MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-87-78, Vol I (of two)
6a NAME OF PERFORMING ORGANIZATION Kestrel Institute	6b OFFICE SYMBOL (If applicable)
7a NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)	
6c ADDRESS (City, State, and ZIP Code) 1801 Page Mill Road Palo Alto CA 94304	7b ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center	8b OFFICE SYMBOL (If applicable) COES
9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-84-C-0109	
8c ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
62702F	5581	19	34

11 TITLE (Include Security Classification)
KBSA PROJECT MANAGEMENT ASSISTANT

12 PERSONAL AUTHOR(S)
Richard Jüllig, Wolfgang Polak, Peter Ladkin, Li-Mei Gilham

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Final	FROM May 84 TO Nov 86	July 1987	36

16 SUPPLEMENTARY NOTATION
N/A

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Artificial intelligence, knowledge based systems, project mangement, software lifecycle,
12	05		
12	08		

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes the work performed by Kestrel Institute as part of a contract with the Rome Air Development Center to define a knowledge-based Project Management Assistant (PMA) that provides for the formalization of, and reasoning about, lifecycle activities in support of software project management. Most project management systems of the current generation simply automate traditional methods of charting project tasks. These systems provide limited assistance in project monitoring and tracking, falling short in intelligent inference capability, extensibility, and adaptability. The goal of the PMA, in contrast, is to provide an intelligent assistant that cooperates with project managers in planning, controlling, and coordinating all the software lifecycle activities. Kestrel's approach for achieving this goal has been to apply methods of knowledge based program synthesis to the software management problem. The premise of this approach is that the analogy that is observed between the many types of plans, agendas, and procedures that are generated, evaluated, and often discarded or executed in the course of managing a software project, (over

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Douglas A. White	22b TELEPHONE (Include Area Code) 22c OFFICE SYMBOL (315) 330-7... RADC (COES)

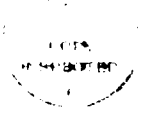
DD Form 1473, JUN 86

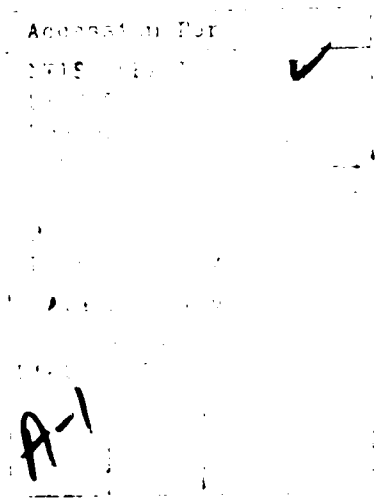
UNCLASSIFIED

Block 19. Abstract (Cont'd)

and the designs and programs of software synthesis. Both involve the development of an executable plan that optimizes certain objective functions within given constraints. Software programs are plans executed on hardware architectures; plans are programs executed on organizational architectures.

This report discusses the technology use, the formalisms and models developed, and describes the PMA project model that was constructed to demonstrate the concepts.





Contents

1 Introduction 1

2 Knowledge-Based Programming 3

2.1 Very-High-Level, Wide-Spectrum Languages 3

2.2 Reuse of Domain Knowledge 5

2.3 Encoding of Domain Knowledge in PMA 6

3 The PMA Model for Project Management 8

3.1 A Formal Model for Software Engineering Environments 10

3.2 The PMA Domain Model 10

4 The PMA Model for Time 14

4.1 Background to the Work on Time 14

4.2 Reasons for Extending The Model 16

4.3 The PMA Time Manager 16

4.3.1 A Taxonomy of Relations 16

4.3.2 Interval Time Units 17

4.3.3 Operators 18

4.3.4 Implementation 19

4.3.5 Additional Work 19

5 The PMA Prototype 19

5.1 PMA Functions 20

5.2 PMA User Interface. 22

6 Conclusions 23

References 24

1 Introduction

Software project management has the responsibility for planning, controlling and coordinating all the software lifecycle activities. Its objective is more cost-effective and more rapid development of quality software. Despite advances in software technology such as the use of higher level languages and improved management techniques (software engineering), currently project managers are severely hampered in achieving this objective by the informal and undocumented nature of lifecycle activities and the fragmentary, obsolete, and inconsistent data available to them. As the demand for new software is increasing faster than people's ability to develop it [8,24,7], we believe that the solution to software project management problems rests not only in improved management techniques, but also in a comprehensive *software environment* that captures all lifecycle activities and the rationale behind them so as to assist all the members of the project team in their respective project tasks.

This report describes the work at Kestrel Institute under a contract (No. F30602-84-C-0109) to Rome Air Development Center on developing a *knowledge-based Project Management Assistant* (PMA) that provides for the formalization of, and reasoning about, lifecycle activities to support software project management.

The Knowledge-Based Software Paradigm. Software development, whether in-the-large or in-the-small, is a knowledge-intensive activity. The conventional informal, person-based software paradigm leaves much of the extensive knowledge required for development implicit and thus fails to capture the entire programming process adequately. In order to solve the problems caused by this approach and many other existing programming methodologies and environments, the knowledge-based software paradigm has been proposed [5]. This new paradigm scores high on all the four software productivity improvement strategies (*i.e. write less code, get the best from people, avoid rework, develop and use integrated project support environments*) suggested by Boehm [8].

The main differences between the knowledge-based paradigm and the traditional paradigm are as follows. In the traditional framework, the emphasis is on the products, e.g. software specifications, program descriptions, and source code. Only the products are recorded, archived, analyzed, and possibly reused. Because the degree of formality of the languages used in these products is weak, it is difficult to support their production with automated tools. Because the transformation steps are carried out manually, inevitably errors are introduced, and additional phases are necessary to discover and rectify those errors.

The knowledge-based approach, on the other hand, attempts to capture the know-how of software production, and to support and record the processes together with the resulting products rather than just the results. To the degree that concepts used by software designers and the knowledge of programming can be formalized, software design and implementation becomes a process that is itself recordable, analyzable, reusable, and, to increasing degrees, automatable. Furthermore, once the formally represented derivation steps have been shown to be correct, all implementations generated using those steps are

guaranteed to be correct. Thus, the effort necessary for validation and verification is drastically reduced.

Transformational Program Synthesis. Transformational program synthesis has been a very active research area for over ten years. Partsch and Steinbrüggen give a survey of transformational systems [27]. Goldberg reviews program design and construction techniques that have been or could be formalized for use in automated program synthesis systems [13].

The literature cited strongly indicates that the research focus in the past has been on providing knowledge-based aids for the design and implementation of programs. Although many problems remain, the transformational program synthesis technology has matured to the point of productive use outside the research laboratory [10].

Knowledge-Based Software Project Management Support. To date, most project management systems simply automate traditional ways of charting project tasks with techniques such as CPM and PERT. These systems provide limited assistance in project monitoring and tracking; they fall short in intelligent inference capability, extensibility, and adapatability. In contrast, the PMA aims at behaving as an intelligent assistant that actively supports project managers in planning, controlling and coordinating all the software lifecycle activities. The complexity of this ambitious undertaking calls for a better modelling of the project management domain and for applying advanced technology in attacking the problem.

We have based our PMA effort on applying methods of knowledge-based program synthesis to software project management problems. The promise of this approach derives from the observation that in the course of a software project (including the "maintenance" phase) many types of plans, agendas, and procedures are generated, evaluated, and then discarded or executed. These plans and agendas can be viewed as programs over a suitable domain of primitive operations.

Viewed from this vantage point, the task and concomitant problems of a manager planning a project and those of a software engineer designing a software system become very similar in nature. Both tasks essentially involve the development of an executable plan that optimizes certain objective functions within given constraints. Software programs are plans executed on hardware architectures; plans are programs executed on organizational architectures.

Thus, the approach taken in knowledge-based program synthesis should apply: formalize pertinent knowledge in suitable formalisms, support the refinement of initial plans and report derivations, aid in the selection from alternative plans based on the objective functions to be optimized, perform automated synthesis of programs for specified goals, and so forth. The rationale for and benefits deriving from these elements are, *mutatis mutandis*, the same as for software design and implementation.

Overview Of The Report. In the remainder of the report, Section 2 briefly discusses and illustrates some of the elements of knowledge based synthesis technology that we use in attacking software project management support problems. Section 3 discusses a formalism for describing software engineering environments, i.e. the formalization of knowledge about software environments, and the PMA project model built on this formalism. Section 4 discusses the work on time modelling and the implementation of the PMA time manager. Section 5 describes the experimental PMA prototype built to demonstrate the feasibility of our approach. We briefly describe the functions and capabilities of a knowledge-based project management assistant as implemented in the prototype. A more detailed description of the PMA prototype can be found in the document "KBSA-PMA Functional Description". In Section 6, a summary is followed by a concluding discussion of some of the implications of our approach.

2 Knowledge-Based Programming

The articulation and formalization of program design and implementation knowledge depends on the development of very-high-level languages: the expression and manipulation of abstract, complex concepts requires suitable notations. The history of mathematics and chemistry demonstrates very clearly how much progress in these disciplines depended on the emergence of convenient formalisms. Computer science is no exception. On the other hand, very-high-level languages remain academic exercises without sufficiently powerful compilation technology that can translate very-high-level specifications into efficiently executable code.

The knowledge required to create a software solution to a problem can be divided into general programming knowledge and idioms, and special application domain knowledge. The former is captured by a set of transformation rules that translate a general wide-spectrum language into efficient code. The latter knowledge is represented by defining suitable concepts within the wide-spectrum language (see 2.2).

2.1 Very-High-Level, Wide-Spectrum Languages

Software development converts an abstract problem specification into a concrete implementation. From the large set of possible implementations, a sequence of design and implementation decisions selects one specific one; each decision fixes some of the details of the eventual solution.

The knowledge-based software paradigm attempts to capture the knowledge involved in the entire software development process, to reason about all the software lifecycle activities, and to interface the users at various levels. A formal language used for supporting this paradigm should provide the human user with natural methods of expressing different aspects of a computational system and focusing only on relevant details at any given life-cycle stage. Many languages, each somewhat suitable for different stages of software

development, currently exist. These languages, although useful, are far from adequate either for formalizing all the stages of the software lifecycle or for interfacing with the tools for any other language or system. Instead, we need a *very-high-level wide-spectrum* language that can be used both to state the software specification formally and succinctly and to cover the whole development range without having to cross artificial language boundaries.

In the knowledge-based software paradigm, software specifications are not only formalized but also executable. The process of deriving efficient implementations from very-high-level specifications can be automated using the transformational program synthesis techniques. In this approach, general programming knowledge is encoded as a set of transformation rules. By applying a sequence of appropriate transformation rules, a very-high-level specification can be stepwise refined into lower-level (wide-spectrum) expressions and finally into the target code which represents an efficient implementation of the original specification. This approach supports the proposed lifecycle change in the knowledge-based software paradigm – maintenance and evolution occur by modifying the specifications and then rederiving the implementation, rather than directly modifying the optimized implementation. When the process of rederiving the implementation from a changed specification is automated, the software reliability and productivity can be greatly improved.

The PMA is intended to be a knowledge-based project management assistant that supports this new software paradigm. It plays different roles, the most fundamental one being "the corporate project memory". It is its "omniscience" about "who is doing what when why at what cost" that enables it to be a flexible source of information to managers and a resourceful assistant to project team members. Thus, the PMA's most vital organ will be a knowledge base, a central receptacle of facts, rules and constraints expressing relations between facts, and meta-rules expressing knowledge about the use of rules and constraints. Clearly, the utility of the PMA is a function both of its ability to reason about facts and to do so efficiently, as well as its interface to human users. It is therefore of prime importance to employ a language that allows the formalization of knowledge that is convenient, i.e. at the conceptual level of humans and efficiently manipulable by the PMA.

In order to achieve these somewhat opposing objectives as well as to increase developer productivity, we have used a very-high-level and wide-spectrum language, *REFINE*™, to develop the PMA prototype. The REFINE language was designed explicitly for the purpose of capturing the software development process and supporting the knowledge-based software paradigm; it allows a variety of programming/specification styles that are suitable for encoding the different types of knowledge required for the PMA.

- *Object oriented specification:* the language supports definition of hierarchical classes of objects and provides an inheritance mechanism. This feature allows the PMA to model different kinds of objects in the project management domain, to describe the various relationship among the objects, and to express class taxonomies with inheritance conveniently.

- *Function specification:* recursive functions can be defined by explicit specification of an implementation, or implicitly by specification of a set of constraints from which

the system can synthesize an implementation. The latter allows functions to be defined in a more natural and declarative way; it frees the description of the problem from the responsibility for efficiency.

- *Logic specification:* the ability to define functions implicitly provides capabilities more general than traditional logic programming languages. In addition, logical assertions can be used to constrain values of program variables. This feature can be used to automatically update dependent variables or to warn of violations of system integrity (see Westfold [33]).

- *Procedural specification:* procedural concepts are required in order for the language to be "wide-spectrum". They are "lower level" in that they are closer to the architecture of conventional machines. But procedural specifications are also a valuable specification concept in their own right: many algorithms are best described procedurally.

The REFINE compiler is a knowledge-based transformation system that compiles these various types of specifications into efficient code; it enables the software developers to program at a very high (specification) level.

2.2 Reuse of Domain Knowledge

The formation of domain specific knowledge is only possible after suitable theories and models have been developed which generalize the concepts and knowledge implicit in ad hoc solutions. An approach to formalizing concepts underlying software engineering environments is described in Section 3. Here we focus on the general mechanisms to represent domain knowledge in very-high level specification languages.

- *Subprograms:* this is the traditional form of reuse of domain knowledge. But clearly the kind of parametrization available in traditional languages limits the usefulness of subprograms to capture general knowledge.

- *Abstract data types:* these are slightly more general. Techniques such as algebraic specifications are satisfactory to specify the formal semantics of abstract data types. But they do not provide for specification of pragmatics and alternate implementation. Lack of this information in the domain knowledge may lead to unacceptably inefficient implementations.

- *Object classes:* Objects and methods in the sense of Smalltalk also capture domain knowledge. But as in the above two cases this representation is still too specific.

- *Axioms and constraints:* combined with sophisticated translation techniques axiomatic definitions provide a highly flexible way to specify knowledge. It is necessary to define the appropriate objects and concepts in terms of which axioms can be stated.

- *Transformation rules:* we believe that transformation rules are a very general form of knowledge representation. Instead of capturing the final product (e.g. abstract data type definition, subprogram, etc) they capture individual steps of the process by which concrete implementations can be rederived.

In the following section, we describe how the techniques described above are used in PMA for encoding the domain knowledge specific to software project management.

2.3 Encoding of Domain Knowledge in PMA

Software project management requires knowledge from broad areas, including

- knowledge of project management
- knowledge of software development, and
- knowledge of the specific projects to be managed.

In order to provide automated intelligent support for software project management, the PMA combines various techniques to represent the domain knowledge formally within the system.

Object Classes and Associated Attributes. Different types of objects in the software project management domain, such as *component*, *task*, *version*, *person*, and *date*, are defined in PMA as "object classes" in appropriate hierarchical structures. Attributes associated with an object class are defined as mappings whose domain is the object class. These mappings provide a general, canonical mechanism to interrelate and annotate the objects in the knowledge base. For example, to capture the domain knowledge about milestones, we might define an object class *milestone* with the following attributes:

object-class	*milestone*	
attributes	*of-task* :	(mapping *milestone* → *task*)
	milestone-description :	(mapping *milestone* → string)
	earned-value-percentage :	(mapping *milestone* → real)
	due-date :	(mapping *milestone* → *date*)
	date-completed :	(mapping *milestone* → *date*)
	reviewed-by :	(mapping *milestone* → (set *person*))

Axioms and Constraint Maintenance. Attributes can have their values either *stored* explicitly or *computed* on demand. In the latter case, an axiomatic definition will be given which is invoked automatically to compute the value of the attribute when it is needed. For instance, the value of the attribute *remaining-duration* for a task can be computed on demand:

attribute *remaining-duration* : (mapping *task* → *duration*)
computed-using *remaining-duration-uptodate*

assertion *remaining-duration-uptodate*
tsk is a task
∧ (duration *tsk*) = *dur*
∧ (actual-start *tsk*) = *start-date*
→ (remaining-duration *tsk*)
= (subtract-duration *dur* (subtract-dates **today** *start-date*))

A data-consistency or policy constraint can also be specified in association with an attribute. When so defined, the constraint maintenance is implemented by attaching a *demon* to the knowledge base object representing the triggering attribute. The demon is invoked automatically to check or enforce the constraint whenever that attribute is set or modified on any knowledge base object.

For example, to maintain a constraint "for each component there must be a task to build that component" we can define a triggering attribute *subcomponents* and an assertion *each-component-has-task* as follows:

attribute *subcomponents* : (mapping *component* → (set *component*))
maintaining *each-component-has-task*

assertion *each-component-has-task*
comp is a component
∧ *s-comps* = (subcomponents *comp*)
∧ *tsk* is a task which builds *comp*
⇒ (sub-versions *tsk*)
= {*s-tsk*: *s-comp* ∈ *s-comps*
∧ *s-tsk* is a task that builds *s-comp*
∧ *s-tsk* produces tested component *s-comp*}

On the other hand, we might want to check a constraint "each task has personnel assignment two weeks before its scheduled start" whenever today's date is updated and give warning to the manager if this constraint is violated.

attribute *todays-date* : (mapping *date* → *date*)
checking *personnel-assigned-two-weeks-before-scheduled-start*
complaining *warning-of-need-for-personnel-assignment*

assertion *personnel-assigned-two-weeks-before-scheduled-start*
(scheduled-start *tsk*) = *s-st*
∧ today is within two weeks of *s-st*
⇒ (personnel-commitments *tsk*) ≠ {}

Transformation Rules. Transformation rules are used in PMA as high-level specifications of test/action programs that encapsulate state transformations in the knowledge base. When using rules to declaratively express the state changes, much processing can be done automatically without being explicitly stated. In PMA, the stepwise refinement of the component hierarchy is achieved by applying the following transformation rule:

rule *refine-component-into-subcomponents (comp)*
comp = 'the-component @*comp-name*'
∧ *subcomp-names* = (get-subcomponent-names-from-user *comp*)
→ (subcomponents *comp*)
= {'the-component @*subcomp-name*' : *subcomp-name* ∈ *subcomp-names* }

Besides the techniques mentioned above, PMA also uses conventional procedures for defining certain operations when appropriate. Other tools used in encoding PMA include pattern language and context mechanism which are briefly described below.

Pattern Matching and Pattern Instantiation. In encoding PMA, a concise pattern language is used to describe networks of knowledge base objects connected by the values of specified attributes. A pattern can be used to either test whether a given knowledge base object matches it or construct a new knowledge base object which matches it. In the above example, the first use of pattern ('the-component @*comp-name*') is for pattern matching, and the second use of pattern ('the-component @*subcomp-name*') is for pattern instantiation.

Context Mechanism. A *context mechanism* is used for maintaining inexpensive access to each of a collection of distinct *contexts*, or states, of the knowledge base, without storing each of these states as a separate copy of the knowledge base. The set of contexts is tree structured by the "successor state" partial-order; that is, each context node in the tree (except the root) is a successor knowledge-base state to its parent context, obtained by some incremental change such as the transformation of a single node. These changes are stored along with other information necessary to return the state of the knowledge base to any given context in the tree. The context mechanism provides a tremendous savings in space as compared to saving a complete state of the knowledge base for each context, and a similarly large savings in time as compared to regenerating ancestor contexts by starting over from the original and redoing part of the transformation sequence.

3 The PMA Model for Project Management

In this section, we describe the framework of PMA on developing a model for software project management. First we discuss some of the global premises of this work.

A basic observation is that (software project) planning and (software) design are closely related activities. The task of the software designer consists in exploring the space of possible implementations that satisfy a given specification. The design process consists in successively investigating the consequences of alternative decisions, and then selecting one of them further narrow the space of admissible implementations. The design knowledge is explicit in the decisions taken but only implicitly represented in the final implementation. The capture of explicit design knowledge and intermediate design stages is one of the elements of the knowledge-based software paradigm.

The task of the software project manager consists in determining a subspace of the space of possible project states through which the project should proceed (project planning) and to ensure that the project stays within the planned subspace (project control). The space of the project is determined by many factors: the target system to be built, the development methods employed, and the organizational structure, to name a few.

Different approaches to software development and maintenance, as expressed in different life-cycle models, differ in their underlying project space and the restrictions they impose on project progression through that space. Shortcomings of the different software process models consist in failures to adequately model the space or in restricting project progression in a way that is incompatible with the complex interaction of the factors that determine the space. For instance, Daly points out the interdependence of target system structure and the development organization [11]. Boehm discusses the conditions under which one of the various process models should be selected [9]. The *spiral model*, described by Boehm in the paper cited attempts to integrate the different models in one unifying model. The spiral model supports the exploration and co-evolution of the target software system and the project plan.

Our work on project management assistance aims at providing a tool that helps the project manager in charting the project space, navigating through it, and recording a trace of the project's journey to allow for backtracking and post-mortem analysis. The PMA therefore provides a mechanism for recording not only different software versions and related documents but also the evolutionary steps of the project plan. This mechanism is based on Polak's software engineering environment model [30]. We have started to investigate the formalization of refinements used in the development of a project plan.

Versions and Derivations. Software project management is constantly confronted with the need for changing plans which could be derived from changes to requirements, personnel, budget, deadline, or problems of schedule slip, budget overrun, software revision, and so on. Due to the interdependencies of project activities, the tasks of *change control, change impact analysis, consistency checking, derivation tracking and restoration, system build with compile, load dependencies, parallel development with private development*

threads and *release management* have become intricate and difficult to manage. This is especially true in the environment of programming-in-the-large. In order to provide automated support for these tasks, a model for describing the different types of objects in the software project management domain, the interactions among them, and the derivation traces in the evolutionary history is developed for PMA. This model is built on the work of W. Polak [30], which we will briefly describe below.

3.1 A Formal Model for Software Engineering Environments

In Polak's formal model for software engineering environment, an environment is defined by specifying a set of domains, relations between domains, and the semantics of available tools. Environment objects are divided into disjoint domains, e.g. program modules, documentation objects etc. which share the common properties:

- *hierarchical structure:* how is an object composed of subobjects,
- *dependencies:* how are these subobjects interacting.

An object is modelled as a node in a flow-graph [25]. Each node has a number of labelled input ports (*references*) and output ports (*definitions*). Edges of the graph connect input to output ports with the same label. To represent a hierarchy, each node in turn represents a complete flow graph describing its composition.

The objects modelled represent *versions* (or instances) of software components, documents and so on. A component is given as an equivalence class of objects. The edges of the flowgraph represent the "*requires*", "*references*", or "*calls*" relationship; the nesting of modules represents the "*is built with*" relationship. Different kinds of dependencies can be represented within this model by using many-sorted sets of labels. (Sub)objects can be shared among different composite objects and logical assertions about properties of objects can be stated without having to account for in-place changes.

The semantics of an available tool in the environment is given by the effect of the tool's application on the state of the environment. For a particular organization, rules and management procedures can be stated formally. Finally, one can express goals to be achieved. Such goals might be to "release a system", to "update documentation" and so forth. Given the right kind of high-level concepts, such goals become simple specifications which can then be compiled into programs that achieve the goals.

3.2 The PMA Domain Model

Based on Polak's model, we have developed a taxonomy of object classes where objects that evolve in the software lifecycle have each of their saved states classified as a *version* of a certain domain, such as *requirement*, *source code*, *document*, *test case*, etc, and the history

of changes from versions to other versions is encapsulated in *derivations*. The general model of *versions* and *derivations* gives a unified view of the system objects' underlying structures and their changes. For supporting project management in the environment of programming-in-the-large, we consider the following goals as important in developing the PMA model.

- *synthesis of the change impact:* Based on the principle of *information hiding*, *interfaces* among source versions are recognized so that the propagation of changes can be minimized. Furthermore, using *small granularity*, change impact is localized down to the *object* (functions, variables, types, ...) level instead of the conventional file level.
- *automation of system build procedure with minimal reloading and recompilation:* including dependency analysis from source code, data flow analysis, file-loading-sequence determination according to dependencies, and incremental loading and compilation.
- *change control and consistency checking:* including giving warnings when inconsistency is detected, enforcing change authorization or approval, recording reason of change, and tracking derivation history.
- *support for parallel development and individual development threads*
- *friendly user interface:* including graphical display, graphical editor, and menu-driven interface.

The PMA models software project management knowledge using an object-class hierarchy, where project entities, activities, their attributes and inter-relations are defined. This object-class hierarchy is illustrated in Figure 1. An example of the object-classes listed is *Version* which characterizes saved states of a project element during its development process. Version objects are further divided into different domains (subclasses) including *Task-Plan*, *Requirement*, *Specification*, *Source-Code*, *Test-Case*, *and Document*. While the details will differ, all domains share the common properties:

- *hierarchical structure:* how a version object is composed of sub-version objects.

  In the domain of *Document*, primitive document objects can be grouped into sections, chapters, and manuals. The hierarchical structure of a document could reflect the hierarchical structure of the program being documented. If the program exists in different configurations then so does the associated documentation.

  In the domain of *Requirement*, the structure of requirement objects is important to relate requirement to individual code modules in order to check compliance and facilitate enhancements after changing requirements.

  Furthermore, the *component* version hierarchy and the *task* version hierarchy together compose the *Work Breakdown Structure* (WBS) which supports a general approach to organizing a software project. The WBS is developed during project plan formation and provides the basis for project monitoring and control.

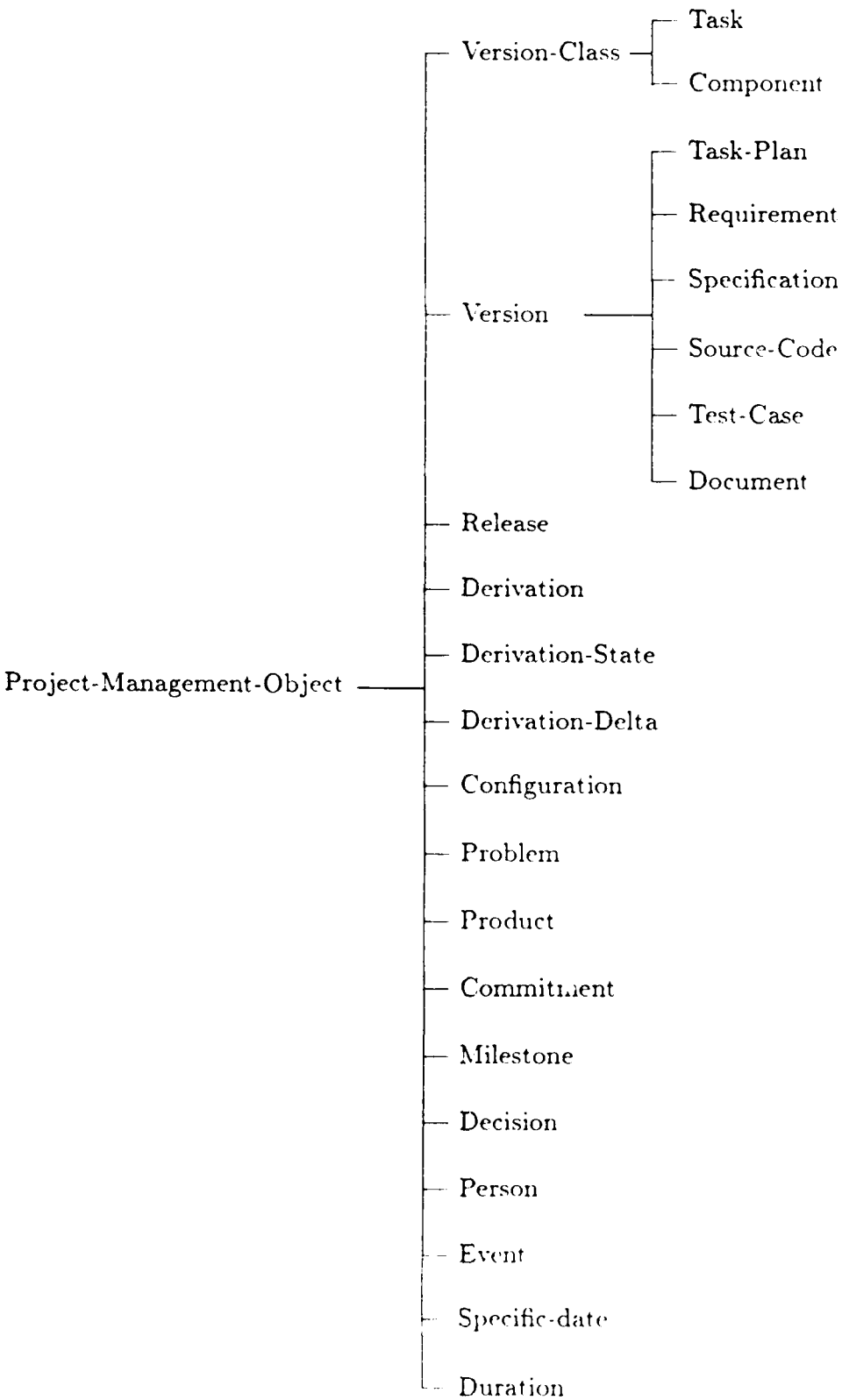


Figure 1. PMA Object-Class Hierarchy

- *derivation trace:* how a version object is derived from and to other version objects.

  The derivation trace of version objects provides an audit trail of system evolution: it allows for rolling back to a previous project state as well as performing post-mortem analysis.

- *dependencies:* how are version objects interacting.

  A representation of various kinds of dependencies between *documents* can be used in a number of ways. For example, after changes to a document a new, consistent set of documents can be compiled, or a minimal set of update notes can be prepared.

  Relations and dependencies among *requirements* are useful to structure and organize partially developed requirements, and to analyze the impact of changes in requirements.

  Source-code dependencies in terms of *interfaces* can be used to automate minimal recompilation during system build.

These common properties are characterized through the following attributes of the object class *Version*.

object-class:	*Version*	
attributes:	*version-for* :	(mapping *version* → *version-class*)
		;;; maps to the equivalence class
		;;; associated with the version
	version-name :	(mapping *version* → symbol)
	initiator :	(mapping *version* → *person*)
	version-time :	(mapping *version* → string)
	super-versions :	(mapping *version* → (set *version*))
	sub-versions :	(mapping *version* → (set *version*))
	derivation-in :	(mapping *version* → *derivation*)
	imports :	(mapping *version* → (set symbol))
	exports :	(mapping *version* → (set symbol))
	definitions :	(mapping *version* → (set symbol))
	other-dependencies :	(mapping *version* → (set symbol))
	version-release :	(mapping *version* → *release*)
	patched-file-name :	(mapping *version* → string)
	version-status :	(mapping *version* → symbol)
	predecessor-versions :	(mapping *version* → (set *version*))
	successor-versions :	(mapping *version* → (set *version*))
	approved-by :	(mapping *version* → (set *person*))

Using the project model supported by the PMA, knowledge about the projects to be managed and the available project resources can be represented formally in the system. The capture of this project knowledge provides a basis for the understanding of and further reasoning about the project space.

Other knowledge required in software project management includes software development methodologies, management policies, data access privileges, data consistency requirements,

and so on. For instance, there may be organization imposed policies describing the quality assurance and documentation requirements that need to be met before a software system can be released. These methodologies, policies, and constraints can be formalized in the PMA system using logical assertions and transformation rules. They impose restrictions on project progression through the project space. By monitoring these constraints, the PMA can support the prevention or detection of constraint violation, or use them to direct the system's actions.

For example, take the obvious constraint that for any task personnel assignments must be made by the actual start date of the task. By alerting the manager responsible ahead of time, the PMA helps to prevent a violation of that constraint. If a violation does occur, the system may react by notifying upper-level management. This also illustrates how the PMA can use its knowledge about the functions and responsibilities of people to route messages intelligently.

When the various software development methodologies are formalized in the system, we can implement Boehm's *spiral model* by specifying the conditions under which one of the methodologies should be selected. As a result, the PMA will support a robust, risk-driven software lifecycle model that provides guidance as to which combination of previous models best fits a given software situation. Although not included in the current PMA prototype, this implementation should be part of the PMA follow-on effort because of the significant productivity improvement it can bring to software development.

4 The PMA Model for Time

4.1 Background to the Work on Time

We have devised a new system for representing time by means of time intervals. This work builds on the work of James Allen, who, in [1,2], suggested using interval representations of time for planning and other theories of action. Allen's work has been concerned with *convex* intervals of time, that is intervals which have no gaps in them. His work has been used and extended by Allen, Pat Hayes, Henry Kautz, Richard Pelavin and others [3,4,28].

The use of intervals rather than points for time representation has also been suggested by logicians and others investigating temporal logic [32,16,15,31,12,26]. Allen observed that there were precisely thirteen relations that convex time intervals could have to each other, including equality, assuming that time is linear. Allen showed how to maintain constraints and perform calculations in this interval system in [2].

A convex interval looks like this:

i ——————————

Some examples of Allen's relations are:

- **i meets j**, which means **i** is before **j**, but there is no gap (interval) separating them

  i ——————

  j ——————

- **i precedes j**, which means **i** is before **j**, and there is an interval separating them

  i ————

  j ——————

- **i starts j**, intuitively **i** has the same starting point as **j**, but is otherwise strictly contained in **j**

  i ——————

  j ————————————

and there are ten others; briefly:

- **equals**
- **overlaps**
- **ends, during**, which are different forms of containment (along with **starts**)
- the converse relations to all these (except **equals**, which is symmetric)

This model of time is important for its unification of the many aspects of temporal reasoning needed for effective project management. For example, in work in progress, we have shown that a quite general class of scheduling algorithms may be defined functionally using the implemented interval model augmented by one new operator on a data type of *weighted-intervals*, which are easily definable from the basic model. This work is a significant step towards synthesizing scheduling algorithms from the basic interval model, and such a synthesizer is part of the vision of an automated PMA. It is our belief that this could not be satisfactorily accomplished without a sophisticated model of time such as has been implemented for the PMA.

Additionally, there is the issue of portability. The PMA should rely as little as possible on the specific development hardware. The model of time we have developed implements calculations that otherwise would have to be accomplished by system calls to the Symbolics time package, with the exception of the call that returns the current clock value (and this call we regard as an essential function on any machine which would support real time functions of the sort needed for the PMA). This allows the PMA time manager to be ported to any machine supporting the REFINE system, with the redefinition of only one function, clearly identified and trivial to rewrite.

4.2 Reasons for Extending The Model

We observed that convex intervals are not sufficient for representing all periods. Stop-start processes, and propositions which become true, then false, then true,... cannot be assigned a convex interval as a time parameter. Rather, the time parameter is an interval which is convex-with-gaps, or, as we prefer to call it, union-of-convex.

The general union-of-convex interval will look something like this:

For an illustration, suppose we are representing scheduling of a processor amongst multiple processes. A process **P** will occupy the CPU in time-slices, and there will in general be many of these time slices. Other processes, or the system, will be using the CPU interleaved with **P**. Thus the time period over which **P** will have control of the CPU looks like the union-of-convex interval in the picture above.

In project management we can find ample examples of how this extended time representation is useful. One of them is to express periodic events such as a project review on every odd Thursday. Another is to calculate a person's availability for further task assignment knowing the commitments he has already made. The time range and effort level of a personnel commitment can be conveniently represented as a *weighted union-of-convex interval*. An "aggregation calculus" for weighted union-of-convex intervals facilitates the summing up of a person's present commitments and the calculation of his availability.

If we had a way of calculating with union-of-convex intervals, we could specify many time-dependent problems and their solution in an elegant manner. We would be able to include the time dependency as a single parameter, and then use the interval calculus to derive the results about the time relations that we need. Our work on the specification of time dependencies using union-of-convex intervals is a promising start to this investigation. A prototype time manager using the results of our investigations is used as the basis for time management in the PMA.

4.3 The PMA Time Manager

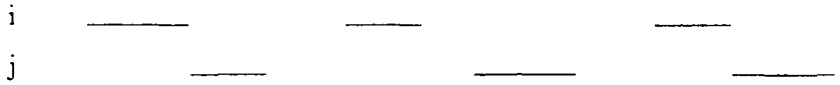
4.3.1 A Taxonomy of Relations

We have developed a taxonomy of relations between union-of-convex intervals, which provides us with a rich specification language for time dependencies amongst tasks, actions, processes, and propositions. This work has been reported in [20,22].

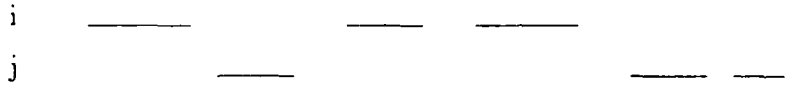
We give below some examples of the many relations in the taxonomy. In the examples, a *maxconsubint* is a single line segment. The name is a contraction of *maximal convex subinterval*, which is the appropriate mathematical definition of a complete line segment in the diagrams below.

- **i always-meets j**. Intuitively, any maxconsubint of **i** has to **meet** a maxconsubint of **j**, and every maxconsubint of **j** is met by some maxconsubint of **i**.

- **i always-(precedes-or-meets) j**. Intuitively, every maxconsubint of **i** either precedes or meets some maxconsubint of **j**, and every maxconsubint of **j** is preceded by or met by some maxconsubint of **i**.

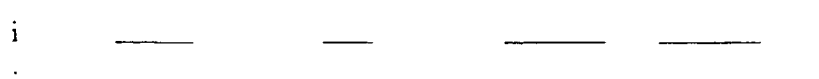


- **i disjoint-from j** Intuitively, **i** and **j** have no subintervals in common. (All of the relations illustrated above are subrelations of **disjoint-from**).

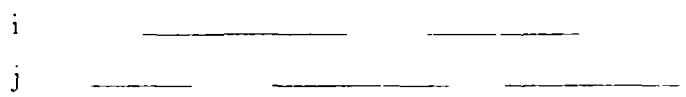


- **i always-starts j**. Intuitively, every maxconsubint of **i starts** some maxconsubint of **j**, and every maxconsubint of **j is-started-by** some maxconsubint of **i**

- **i contained-in j**. Intuitively, every subinterval of **i** is contained in some subinterval of **j**.



- **i bars j**. Intuitively, the 'union' of **i** and **j** is a convex interval.



4.3.2 Interval Time Units

We have also designed a system of interval time units, which is very flexible, and can incorporate all the normal units of time with which we work, such as years, days, minutes,

picoseconds, weeks, Mondays, Januarys, First Days of the Month, etc. The unit system has been implemented, up to the level of granularity of days, and this prototype has been used as a basis for the time manager of the PMA. This work has been reported in [21,22].

We use sequences of integers to represent our standard time units. These sequences are interpreted as containing entries corresponding to standard time lengths, i.e. **year, month, day, hour, minute, second, ...** arranged as a sequence. Rather than giving a definition, we provide a sample of units below, along with the interpretation of each unit. We illustrate the units down to the level of granularity of *seconds*, so our sequences in this example will have lengths of up to six elements. Clearly, the system is easily extendable to smaller units such as microseconds, simply by using longer sequences.

- **[1986]** represents the year 1986
- **[1986,3]** represents the month of March, 1986
- **[1986,3,21]** represents the day of 21st March, 1986
- **[1986,3,21,7]** represents the hour starting at 7am on 21st March, 1986
- **[1986,3,21,7,30]** represents the minute starting at 7:30am on 21st March, 1986
- **[1986,3,21,7,30,32]** represents the 33rd second of 7:30am on 21st March, 1986 (the first second starts at 0)

Other important periods of time, such as *weeks*, *Mondays*, *Januarys*, *First Days of the Month*, *Mondays-in-January-1987*, may be defined from these basic time units in a simple manner using the definitional apparatus available to us on the system we use at Kestrel Institute.

4.3.3 Operators

There are certain primitive polymorphic functions on intervals, sets of intervals, and numbers, that are required both for a full specification language, and for the implementation of a time interval system. We have reported on these functions in [21,22], and we have made substantial progress on implementing these functions in the time manager.

We illustrate with a pictorial example of the operator *combine*

Suppose the set of intervals is **A** = { **i** , **j** , **k** } , where **i**, **j** and **k** are

Then **combine({i , j , k })** will be the interval

(These two diagrams are intentionally aligned)

4.3.4 Implementation

We have mentioned that the system of time units has been implemented down to the level of granularity of days, and is used as the basis for the PMA time manager. Allen's relations are implemented amongst the time units. Primitive operators such as *combine*, and some of the relations in the taxonomy of union-of-convex relations are also implemented on intervals formed out of the units by using the *combine* operator.

Ultimately, we wish to incorporate the full calculus of union-of-convex intervals, as an abstract relation algebra, with a special purpose theorem prover.

4.3.5 Additional Work

During the course of investigating interval systems for incorporation into the PMA, we discovered that Allen's relations formed a *relation algebra* in the sense of *Tarski* [17,18,23]. We have investigated the mathematical structure of Allen's algebra with the intention of discovering new algorithms for consistency checking that may be used with Allen's relations. In joint work with Roger Maddux of Iowa State University, we have discovered a semi-decision procedure for inconsistency of interval specifications (see [2] for definitions and an example). To our knowledge, this is the only such algorithm that has been proved correct. Further joint work with Maddux has clarified the relation-algebraic structure of our calculus of relations on union-of-convex intervals. This work is currently being written up for publication, and the major results were announced at invited talks at SRI, Stanford, and the AAAI-86 conference in Philadelphia.

5 The PMA Prototype

PMA Prototype Architecture. With regard to the software project entities modeled, the PMA knowledge base is similar to the Project Master Data Base (PMBD) described by Penedo and Stuckle [29]. In distinction to PMDB, the PMA has many *active* components.

It is our belief that a project management tool like the PMA is most effective if tightly integrated with the technical software development environment. Integration with software engineering tools makes it possible to automate many data collection functions, reducing

the amount of drudgery for both managers and software engineers. Both are more likely to accept the PMA if it does not impose extra work on them. It also allows for flexible use of the available information without having to contend with artificial boundaries.

The implementation of the PMA used the program synthesis technology described in Section 2. A summary of the functions and interface of the experimental PMA prototype follows.

5.1 PMA Functions

Below we give a cursory description of the principal PMA functions.

Project Structuring. The PMA prototype supports a fairly general approach to structuring a software project, i.e. to organize project activity elements into the *Work Breakdown Structure* (WBS) consisting of a *component-version hierarchy* and a *task-version hierarchy*. The two hierarchies in the WBS may be interrelated in different ways dependent on the project; they are developed in the project planning phase and provide the basis for project monitoring and control.

The component-version hierarchy reflects the breakdown of the overall software system into planned components, e.g. subsystems, modules, and routines. The task-version hierarchy is a hierarchy of activities: the task of building the overall system is successively broken down into smaller tasks. For instance, the task of building a component can be divided into designing, coding and testing that component.

The PMA allows the user to perform stepwise refinement to the component-version hierarchy; it maintains a constraint to ascertain that each component has a task to build it. As a task is being refined, the PMA also monitors the allocation of required personnel effort and duration to its subtasks. It allows the refinement of a task into a sequence of activities including design, prototyping and test, and assumes default allocation percentages unless the user overrides them. The PMA also maintains certain data consistency constraints as the project knowledge base is being updated.

Task Scheduling. The PMA calculates earliest and latest possible start and finish dates for the tasks in the project based on the task dependencies and the project start and finish dates. It calculates the critical paths in the schedule based on the earliest and latest start and finish dates for the tasks in the project. It generates task schedules, allowing the manager to pursue a philosophy of evenly "loading" the team members, or aiming for early push or late push in the task. It produces Pert and GANTT charts.

Task Assignment. In the PMA prototype, it is assumed that each task in the task hierarchy of a project is given an estimate of personnel effort required to complete the task. The function *task assignment* serves to associate a given task with personnel and a percentage of their time. Since task assignment is one of the areas in project management that requires human judgment, the PMA provides assistance but leaves the actual assignment choices to the managers.

Specifically, the PMA prompts task managers for personnel assignments to tasks whose start date is within a certain, manager-specifiable time span. It displays a multiple-choice menu listing the people available together with their availability percentage. It checks whether a task assignment entered by a manager satisfies all related constraints and gives warnings if not. It supports multi-level task assignment.

Policy Enforcement. In each organization, there is a set of policies and procedures that govern the software process. The PMA helps to enforce policies and procedures by preventing violations or triggering remedial events or actions.

In the PMA prototype, policies and procedures are expressed in a very-high-level language as logic assertions or rules. This makes it very easy to adapt the PMA to changing guidelines and standards.

Monitoring Functions. The PMA monitors requirements, schedules, and cost. Authorized changes, and their underlying reasons, to requirements, schedules and budgets can be tracked and recorded in the knowledge base. This trail is part of the "project development history" that can be used in post-mortem analysis and as a reference in the planning of future projects.

Requirements Monitoring. The PMA maintains a mapping from requirements to system components that fulfill the requirements and monitors their development status.

Schedule Monitoring. The PMA computes and keeps track of all the schedule data in its project knowledge base. It maintains consistency among schedule data. It monitors the dependency relationship among software components and derives task scheduling constraints based on that. It displays the schedules graphically using GANTT chart or Pert chart upon user's request. It also monitors the actual progress of each scheduled task so that potential crises can be detected.

Cost Monitoring. The PMA calculates and tracks expenditures for each task based on personnel assignments and expense of an agent's time. Task progress is measured relative to user-defined milestones. On demand, the PMA displays planned cost, actual cost and earned value graphically on budget charts. It detects cost overrun and underrun and gives early warnings to the task managers.

The PMA calculates the planned budget for each task using personnel allocations and expense of agent's time. It tracks actual expenditures for each task in terms of the amount of work done on the task and expense of agent's time. Task progress is measured relative to user-defined milestones. Earned values of tasks are calculated and tracked based on progress reported on their associated milestones. On demand, the PMA displays planned cost, actual cost and earned value graphically on the budgetary progress chart.

Version Control and Derivation Tracking. Version control and derivation tracking are based on the version concept of the environment model described in Section 3. A component represents an equivalence class of versions. The PMA relates the versions of a component by their derivation history, a record of the changes that led from one version to another. Changes are traceable to the tasks under which they were performed. The derivation history of a component can be browsed, summarized, and edited in various ways.

More specifically, the PMA keeps tracks of the derivations states of the module that a developer is currently working on, and displays the derivation history of the current module upon his/her request. It allows the user to save the current state of the module worked on into a new version. The versions of a module are saved incrementally as "deltas" from previous versions. It allows the restoration of a previous derivation state. It supports parallel development (more than one derivation thread). It also allows the user to undo changes, to save a previous derivation state into a new version, and to unsave a previous version.

Problem Tracking. The PMA allows users to report problems, problem-fix assignments, and problem-fixes to the system. A trace of user activities is automatically included in the report to provide context information for problem diagnosis, including files loaded and software units compiled.

The PMA distributes problem-reports intelligently. Using its knowledge about task assignments and relationships between system components, it distributes problem-reports and reports about problem-fixes to interested parties. Requests for problem-fixes are sent to the person responsible. Problem and problem-fix notices are distributed to personnel working on components related to the one the problem was found in. Users can query the PMA for known problems in a system component.

5.2 PMA User Interface.

The user interface of the PMA is menu-driven with graphic display. The PMA prototype is being implemented on a Symbolics 36xx Lisp machine which provides high-resolution graphic output plus a keyboard and a pointing device (mouse) for input. At any time, there are a number of windows into the display concerned with various aspects of one or more activities in which each user is engaged. The PMA provides a variety of displays, both graphical and textual.

Input. When appropriate, the system requests data inputs from the user by printing a menu which lists all possible input values. Otherwise, the system prompts the user for input from the keyboard.

Output. When appropriate and possible, the system presents it's output in graphical form; the formats for WBS, Pert charts, GANTT charts, budget charts, and calendars are similar to those used in current management practice. Alternatively, the WBS and the derivation history may be displayed as mouse-sensitive, indented text. This allows for selective, compact viewing (browsing) of the WBS and derivation history. For certain types of output, such as messages and notifications, text is the only convenient display form.

6 Conclusions

Summary of Report. The work described in this report is part of efforts that aim at the creation of software engineering environments based on the knowledge-based software paradigm. In the past, most work oriented toward the knowledge-based paradigm has concentrated on the design and implementation of software per se. Building on the results of knowledge-based programming, we investigated the application of program synthesis technology to programming in the large, i.e. the plans and procedures that are generated and executed at different software project activity levels. We employed a formalism suitable for the codification of software project knowledge, and developed a model for software project management environment in which it is possible to generate automated tools that synthesize plans or agendas of project activities. We described our work on developing a general time model that is capable of representing time concepts useful to project management. In Section 5, we briefly described an experimental prototype of knowledge-based project management assistant which both gave rise to some of the ideas introduced previously and provided a testbed for them.

The PMA prototyping effort has concentrated on project monitoring and communication support. This is a well-chosen starting point since monitoring is in many ways more tractable than project planning; nevertheless the design and prototyping of a project monitoring assistant required a thorough understanding of the many agents, functions, objects, and procedures that participate in project management. The monitoring functions required the modeling of tasks, policies, requirements, schedules, cost, resources, performance, coding and testing *results*, *management decisions*, *people*, and *time*, among others. This work has provided a knowledge-based framework on which further enhancement to the PMA prototype in the areas of intelligent planning, risk assessment, and cost management support can be incrementally implemented.

For the purpose of getting feedback and indicating areas needing improvement, we had planned to use the PMA prototype internally during its development. The practicality of such an internal usage implies the need for the PMA prototype to provide a friendly user interface and a reasonable performance even in the early stages of development. We

have some experience indicating that a system like [illegible] used by technical personnel only if it is well-integrated with [illegible] environment and its presence is unobtrusive. These user requirements could not be met by a system under development. Furthermore, our current computer infrastructure did not allow for convenient access to the PMA by administrative personnel. As a result, this plan was not actually carried out. In retrospect we feel that in spite of the unavoidable trade-off issue, an internal usage of the PMA prototype is still desirable and we plan on doing this in the near future.

Concluding Remarks

The close similarity between software design and software project planning (i.e. project design) explains why the productivity impeding problem sources are the same: only end results are recorded, and the lack of development histories and rationales behind the development steps precludes the analysis and reuse of knowledge gained.

Research on automated program synthesis has produced methods with which these problems can be tackled in the domain of software design and implementation. Our work on knowledge-based project management assistance has strengthened our belief that the same methods can be gainfully employed in the automation of software life cycle support functions.

Work on formal environment models opens up the possibility of *specifying* software environments. This in turn would provide project designers the added flexibility of co-designing the target software system, the project plan, and the development environment.

On a final note, we offer the following observation. Better software environments are needed to increase software productivity. But the converse also seems to hold: We need higher software productivity to provide better software environments. A knowledge-based software environment at once models and is part of a software producing organization. Organizations change continually; therefore, its model, the software environment, must change with it to remain useful. The difficulties experienced by many MIS departments corroborate this observation. Knowledge-based program synthesis could turn out to be the enabling technology for better software environments.

References

[1] **J. Allen.** *Towards a General Theory of Action and Time.* Artificial Intelligence 23 (2), July 1984, pp. 123-154.

[2] **J. Allen.** *Maintaining Knowledge about Temporal Intervals.* Comm. ACM 26, 11, November 1983, pp. 832-843.

[3] **J. Allen, P. Hayes.** *A Commonsense Theory of Time.* in Proceedings IJCAI 1985, pp. 528-531.

[4] **J. Allen, H. Kautz**, *A Model of Naive Temporal Reasoning*, in J. Hobbs, R. Moore, editors, *Formal Theories of the Commonsense World*, Ablex 1985.

[5] **R. Balzer, T. Cheatham, C. C. Green**, *Software Technology in the 1990's: Using a New Paradigm*, IEEE Computer, November 1983, pp. 39-45

[6] **David R. Barstow**, *Knowledge-Based Program Construction*, Elsevier, New York, 1979

[7] **B. Boehm**, *Software Engineering Economics*, Prentice Hall, 1981.

[8] **B. Boehm**, *Understanding and Controlling Software Costs*, Personal Communication, February 1986

[9] **B. Boehm**, *A Spiral Model of Software Development and Enhancement*, ACM Software Engineering Notes, Vol. 11, No. 4, August 1986, pp. 22-42

[10] **T. Brown, L. Markosian**, *Knowledge-Based Software Development: From Requirements to Code*, submitted for publication

[11] **E. Daly**, *Organizing for Successful Software Development*, Datamation, December 1979, pp. 107-116. Reprinted in D. Reifer, *Software Management*, IEEE Tutorial, second edition, pp. 143-150

[12] **D. Dowty** *Word Meaning and Montague Grammar*, Reidel, 1979.

[13] **A. Goldberg**, *Knowledge-based Programming: A Survey of Program Design and Construction Techniques*, IEEE Transactions on Software Engineering, July 1986, pp. 752-768

[14] **A. Goldberg and G. Kotik**, *Knowledge-Based Programming: An Overview of Data and Control Structure Refinement*, In H. Hausen, ed., *Inspection, Testing, Verification, Alternatives*, pp. 287-309, Elsevier North-Holland, 1984

[15] **C. Hamblin**, *Instants and Intervals*, Studium Generale 27, 1971, pp. 127-134.

[16] **I. Humberstone**, *Interval Semantics for Tense Logic: Some Remarks*, J. Philosophical Logic 8, 1979, pp. 171-196.

[17] **B. Jonsson, A. Tarski**, *Boolean Algebras with Operators I*, American J. Mathematics (73), 1951.

[18] **B. Jonsson, A. Tarski**, *Boolean Algebras with Operators II*, American J. Mathematics (74), 1952, pp. 127-162.

[19] **G. Kotik**, *Knowledge-Based Compilation of High-Level Data Types*, Kestrel Institute, Technical Report, 1983

[20] **P. Ladkin**, *Time Representation: A Taxonomy of Interval Relations*, Proceedings of AAAI 86, pp. 360-366, Morgan Kaufmann, 1986

[21] **P. Ladkin**, *Primitives and Units for Time Specification*, Proceedings of AAAI 86, pp. 354-359, Morgan Kaufmann, 1986

[22] **P. Ladkin**, *Two Papers on Time Representation*, Kestrel Institute Research Report KES.U.86.5, Kestrel Institute, Palo Alto, CA, 1986.

[23] **R. Maddux**, *Topics in Relation Algebras*, Ph. D. Thesis, University of California at Berkeley, 1978.

[24] **E. Martin**, *Strategy for a DOD Software Initiative*, Computer, March 1983

[25] **Robin Milner**, *Flowgraphs and Flow Algebras*, J.ACM 26/4, pp. 794-818, October 1979

[26] **W. Newton-Smith**, *The Structure of Time*, Routledge Kegan Paul, 1980

[27] **H. Partsch, R. Steinbrüggen**, *Program Transformation Systems*, Comput. Surveys, Vol 15, No. 3, Sept. 1983, pp. 199-236

[28] **R. Pelavin,, J. Allen**, *A Formal logic of Plans in a Temporally Rich Domain*, Proceedings of the IEEE 74 (10), October 1986, pp. 1364-1382.

[29] **M. Penedo and E. Stuckle**, *PMDB - A Project Master Data Base for Software Engineering Environments*, Proceedings on the 8th ICSE, August 1985, pp. 150-157

[30] **W. Polak**, *Framework for a Knowledge Based Programming Environment*, Workshop on advanced programming environments, Trondheim, 1986

[31] **P. Roper**, *Intervals and Tenses*, Journal of Philosophical Logic 9, 1980.

[32] **J.F.A.K. van Benthem**, *The Logic of Time*, Reidel 1983.

[33] **S. Westfold**, *Logic Specifications For Compiling*, Kestrel Institute, KES.U.83.6, June 1984

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.

END

DATE

FILMED

MARCH

1988

DTIC