KES.U.83.2

Kestrel Institute

REPORT ON A KNOWLEDGE-BASED SOFTWARE ASSISTANT

Prepared by

CORDELL GREEN (Co-Chairman)

DAVID LUCKHAM (Co-Chairman)

ROBERT BALZER

THOMAS CHEATHAM

CHARLES RICH

Prepared for
ROME AIR DEVELOPMENT CENTER
Griffis AFB, New York 13441
June 15, 1983

This research is supported by Rome Air Development Center through the University of Dayton, Ohio Order No. RI-23321 under United States Government contract #F30602-81-C-0206. This report has appeared as RADC # TR 83-195. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Kestrel Institute or Rome Air Development Center. Additional copies are available from Kestrel Institute.

KESTREL INSTITUTE • 3260 HILLVIEW AVENUE • PALO ALTO, CA 94304 • (415) 493-6871

Contents

Plates

ABSTRACT

This report presents a knowledge-based, life-cycle paradigm for the development, evolution, and maintenance of large software projects. To resolve current software development and maintenance problems, this paradigm introduces a fundamental change in the software life cycle – maintenance and evolution occur by modifying the specifications and then rederiving the implementation, rather than attempting to directly modify the optimized implementation. Since the implementation will be rederived for each change, this process must be automated to increase its reliability and reduce its costs. Basing the new paradigm on the formalization and machine capture of all software decisions allows knowledge-based reasoning to assist with these decisions. This report describes a knowledge-based software assistant (KBSA) that provides for the capture of, and reasoning about, software activities to support this new paradigm. This KBSA will provide a corporate memory of the development history and act throughout the life cycle as a knowledgeable software assistant to the human involved (e.g., the developers, maintainers, project managers, and end-users. In this paradigm, software activities, including definition, management, and validation will be carried out primarily at the specification and requirements level, not the implementation level. The transformation from requirements to specifications to implementations will be carried out with automated, knowledge-based assistance. The report presents descriptions for several of the facets (areas of expertise) of the software assistant including requirements, specification validation, performance analysis, development, testing, documentation, and project management. The report also presents a plan for the development of the KBSA, along with a description of the necessary supporting technology. This new paradigm will dramatically improve productivity, reliability, adaptability, and functionality in software systems.

§1 EXECUTIVE SUMMARY

1.1 Objectives

The purposes of this report are:

1. To propose a formalized computer-assisted paradigm for the development, evolution, and long-term maintenance of very large software application programs.
2. To describe the knowledge-based software assistant (KBSA) needed to support that paradigm.
3. To outline a long-term development plan designed to realize such a knowledge-based assistant.

1.2 The Problem

The existence of a software problem for large systems and its relevance to the military, which is becoming ever more reliant on software in its weapon systems, its planning, its logistics, its training, and its command and control has long been recognized and is well chronicled [1]. To date, attempts to resolve this problem have yielded only modest gains (a factor of 2-4 compared with the thousand fold increase in hardware performance) arising primarily from use of higher level languages and improved management techniques (software engineering).

Although further modest gains can still be achieved by continuing and accelerating this current technology, a fundamental flaw in the current software life cycle precludes larger qualitative improvements. The **process** of programming (conversion of a specification into an implementation, requirement into specification, etc.) is informal and largely undocumented. It is just this information, and the rationale behind each step, that is crucial, but unavailable, for maintenance. The current paradigm fails to recognize the general need to capture all life-cycle activities and the rationale behind them.

In order to capture the programming process and use knowledge-based tools appropriately, we must formalize **all** levels of activities as well as the transformations between them. Consider the current situation in which only the source code (implementation level) is available, but the specification and the mapping from it to the source code is not. In this situation, maintenance can be performed only on the source code (i.e., the implementation) which has already been optimized by the programmers. These optimizations spread information (take advantage of what is known elsewhere) and substitute complex but efficient realizations for simpler abstractions. Both of these effects exacerbate the maintenance problem by making the system harder to understand, by increasing the dependencies among the parts, and by delocaliz-

ing information. Similar situations hold for the mappings between requirement and specification, or requirement and testing levels.

1.3 Solution: A New Computer-Assisted Paradigm

We propose to shift the current informal person-based software paradigm into a formalized computer-assisted paradigm and to provide the knowledge-based software assistant (KBSA) required to support the paradigm.

The goals are more reliable and rapid development of systems with greater functionality and the build-up of a computerized corporate memory which, through the KBSA, will aid the continued evolution and adoption of the system, especially in the face of personnel turnover. The processes targeted for such assistance constitute the entire life cycle of major software development: project management, requirements definition, validation, implementation, testing, documentation, and maintenance. Thus, KBSA clearly parallels the DoD Software Initiative [1] and is a natural, long-term complement to it.

The basic KBSA paradigm can be summarized as "machine-in-the-loop", where **all software life-cycle activities are machine mediated and supported.** Initially, the KBSA will automatically document the occurrence of every activity and ensure the proper sequencing and coordination of all the activities performed by the individuals involved in a large project. Then, as the various activities are increasingly formalized, more sophisticated knowledge-based support will be provided.

In addition to mediating and supporting all life-cycle activities, all decisions (whether they concern requirements, validation, implementation, testing, or maintenance) must also be recorded together with their rationale. All these data must be machine readable and machine manipulable, so that the system can utilize programming and application knowledge bases as well as inference-based methods to explain complex aspects of the program and support its maintenance. Eventually – on the basis of understanding the relationships between the goals and the code of the application program – KBSA should be able to suggest plausible strategies for the design of program modifications and bear an appreciable portion of the burden of implementing and testing those strategies.

Because KBSA is mediating all development activities, it can support not only those individual activities, but also the development as a whole. It can coordinate one activity with another to maintain consistency; it can alert management if resources are, or are about to be, exceeded; it can help prevent maintainers from reexploring unfruitful implementations. In short, by mediating all of the development activity and by being knowledgeable about that development, KBSA can help people bring to bear whatever knowledge is relevant for their particular task. This is especially important on large projects where it is currently difficult, if not impossible, for people to comprehend and assimilate all the relevant information, which may be fragmented, incomplete, or inconsistent.

Rather than being merely a collection of capabilities, the KBSA would be an intelligent assistant that interfaces people to the computerized "corporate memory," aids them in performing their tasks, and coordinates their activities with those of other members of the team.

1.4 Areas of Assistance

We plan to incrementally formalize, and provide knowledge-based support for, all aspects of the software life cycle. In this section we highlight three areas that readily distinguish the KBSA paradigm from incremental improvements to the current paradigm. The first such area is "development," which encompasses both implementation and maintenance. We propose to formalize it so that implementations are the result of a series of transformations of the specifications. This formalization of the development process will enable maintenance to be performed by **altering the specification and replaying the previous development process** (the series of transformations), slightly modified, rather than by attempting to patch the implementation.

Such a capability will have profound effects. Systems will be larger, more integrated, longer lived, and will be the result of a large number of relatively small evolutionary steps. Software systems will finally reach their potential of remaining "soft" (modifiable) rather than becoming ossified, hardware-like, with age.

Another important life-cycle area is specification validation. Rather than validating already implemented systems which are difficult and expensive to change when problems are detected, validation will be performed by **using the specification itself as an executable prototype**. Specification errors detected will be much simpler and cheaper to correct, and systems will normally undergo several such specification/validation cycles (to get the specification "correct" and to get the end-users to completely state their requirements) **before** an implementation is produced.

A third life-cycle area is perhaps the most important: project management. The formalization, mediation, and support of life-cycle activities includes project management itself. Protocols will define the interaction between successive activies of a single agent and the concurrent activities of multiple agents. These activities will be mediated by an "activities coordinator." New management techniques will have to be developed for such a formalized and partially automated environment.

The other areas of assistance discussed in this report include requirements, performance, testing, and documentation.

1.5 Usage

As the KBSA evolves, it will be able to serve the needs of all participants in the program development, from the program manager to the journeyman coder. As it

serves those needs, it will also serve as the repository of corporate knowledge, making possible both effective coordination of a large number of programmers and smooth transitions without serious setbacks as programming personnel change. While the KBSA will support all programming activities, it will present very different faces to different participants, depending on their roles in the program development process. To the project manager, it will appear as a planning assistant to help allocate tasks, and as a crisis monitor, warning of significant changes in system requirements or schedules and serving as a recording communications channel to the echelon of managers below. To the programmer in charge, for example, of testing a particular module, it will also serve as a news wire informing him/her of relevant program changes. But, in this case, it will further bring to bear its knowledge of program dependencies and of the rationale of prior test designs in order to assist the programmer in both the design and execution of the consequent retesting.

The application programs targeted include the very large (more than one million instruction) programs, such as those associated with command and control or weapons systems, that today require teams of more than a hundred programmers working several years on the original development and at least a decade on system maintenance.

1.6 The Development Plan

The KBSA development plan calls for the study and construction, over approximately a 10- to 15-year period, of individual mechanized facets of the assistant knowledgeable in program management, requirements analysis, implementation, validation, performance optimization, testing, and portability. At first, most of these facets will serve primarily as advanced documentation systems, recording the rationale for all design and implementation decisions. The first major technical efforts must be to formalize the representation of the subject matter and strategies in the domain of each facet. Next, inference mechanisms must be introduced to support the mechanical exploitation of the formal system development databases. Finally, knowledge bases specific to each facet, e.g., heuristic knowledge about the circumstances under which various choices of program transformations induce performance efficiencies, must be compiled. The true strength of KBSA will emerge as these knowledge-based methods provide greater levels of automation for the individual domain facets. KBSA must, just as importantly, provide a full life-cycle program development environment – a matrix in which the several facets may be integrated and which can serve as the all-important communication and coordination channel between them. The development of such advanced program coordination and a new form of project management appropriate for such an environment is an integral part of the KBSA proposal.

To achieve these goals one must:

1. Incrementally formalize each software life-cycle activity (with particular emphasis on project management, development, and validation) and create knowledge-based

tools and automated aids to support their use.

2. Formalize the coordinations and dependencies that arise in large software projects, create a language for stating project management policy in terms of these coordinations and dependencies, and an "interpreter" which coordinates all project activity in accordance with these rules.

3. Construct a framework in which all the tools and capabilities can be integrated (i.e., a life-cycle support environment) as they are incrementally created.

We believe that these requirements necessitate the use, and further development, of knowledge-based artificial intelligence techniques. Toward this end, our plan includes a major thrust of fundamental work in the supporting technologies of integrated knowledge representation, knowledge base management, and inference.

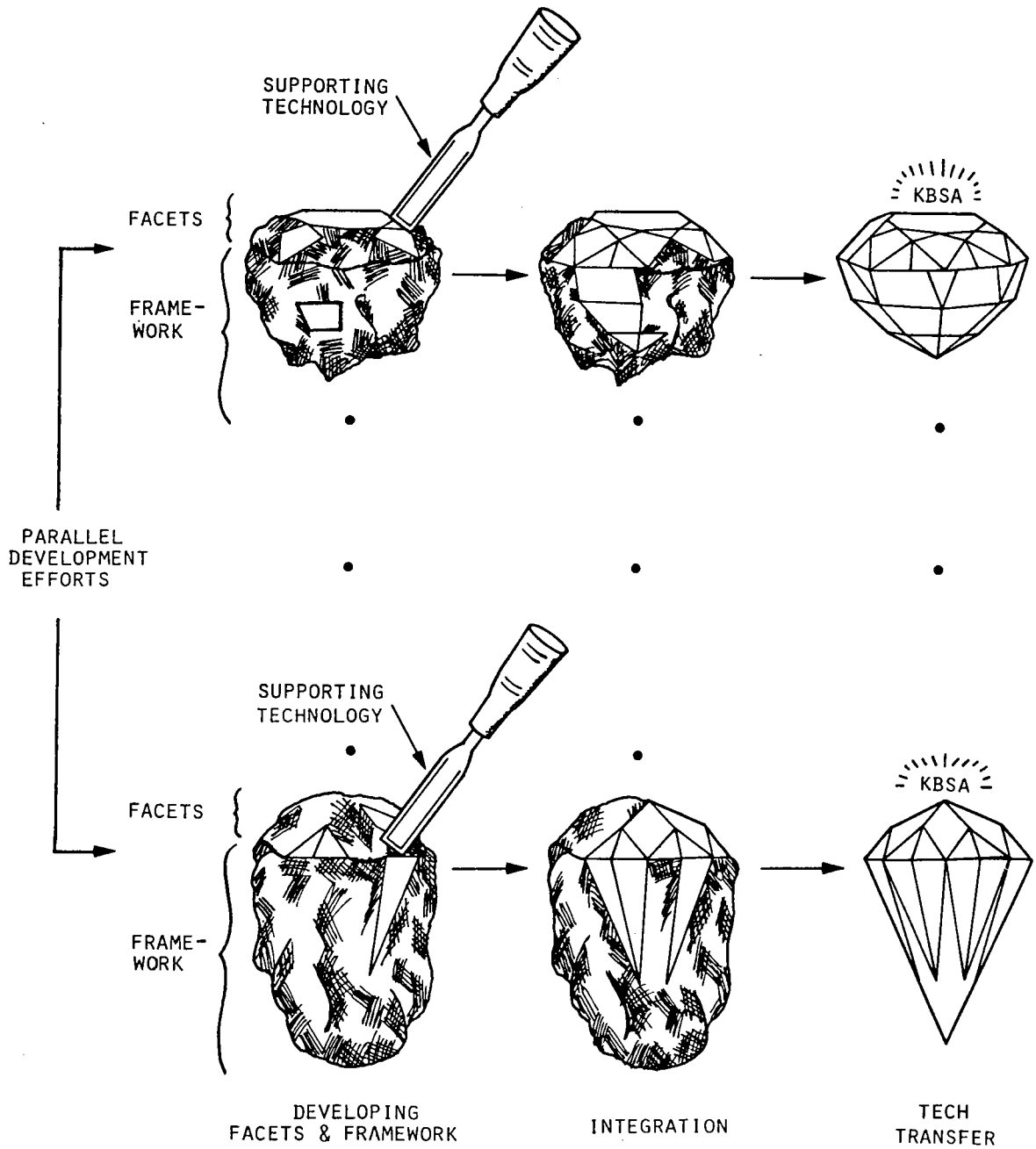

Figure 1. DEVELOPMENT OF PARALLEL KBSAs OVER TIME

In the short term, the plan calls for several parallel efforts to construct the system framework, including activities coordination. The more successful will lead to the standards into which the formalized activities and automated aids will be integrated as they mature. Several such separate, unintegrated formalizations and automation efforts will be started. These development efforts are illustrated in Figure 1.

In the mid term, the separate formalization and automation efforts will be integrated into the standard frameworks to produce demonstrable prototypes. Meanwhile, the separate formalization and automation efforts will continue.

In the long term, one or more integrated prototypes will be production engineered for real use and transferred.

The plan distinguishes between varying degrees of automation and promises a certain amount of near- and mid-term technological "fallout." It also provides for a crucial mid-term attempt at system integration, the watershed test of whether the evolving KBSA meets the goals of rapid re-prototyping and retained software flexibility. Only if it does will knowledge-base managed system development and maintenance go beyond a brave new paradigm to become a reality.

The plan calls for a steering committee to help in further planning and to oversee the development of the KBSA.

§2 PROBLEMS AND SOLUTIONS

2.1 Statement of the Problem

The existence of a software problem and its relevance to the military, which is becoming ever more reliant on software in its weapon systems, its planning, its logistics, its training, and its command and control has long been recognized. The multitude of problems with the existing software development and maintenance life cycle, and their particular acuteness for the military, have been well chronicled elsewhere [1]. As pointed out in the DoD Software Initiative [1], merely doubling current productivity would result in yearly DoD savings of $2.5 to $3 billion and a payoff factor of over 200 on the investment.

Yet, attempts to date to resolve this problem have yielded only modest gains arising primarily from use of higher-level languages and improved management techniques. These improvements, which by the most optimistic estimates, have resulted in far less than an order-of-magnitude gain over the last 15 to 20 years, have in no way kept pace with the astounding thousand-fold increase that has occurred in hardware performance over the same period. Because the hardware revolution apparently will continue at this pace for at least the rest of this decade, it is clear that the utility of computers to the military, and to society as a whole, will be limited primarily by our ability to construct, maintain, and evolve software systems.

Continuation of existing efforts to improve the current software paradigm, broadly characterized as software engineering, will undoubtedly yield further incremental improvements more or less commensurate with those previously obtained, subject to the law of diminishing returns.

Rather than discuss problems with the current software paradigm here, we instead examine the underlying causes of these problems and suggest that qualitative improvements cannot be made until these underlying causes are removed. Unfortunately, the current software paradigm, which arose in an era when machines rather than people were expensive and in limited supply, is fundamentally flawed in a way that precludes larger qualitative improvements.

The flaw is that there is no technology for managing the knowledge-intensive activities that constitute the software development processes. The process of programming (the conversion of a specification into an implementation) is informal and largely undocumented.

It is just this information, and the rationale behind each step of this process, that is crucial, but unavailable, for maintenance. As a consequence, maintenance is performed on the implementation (i.e., the source code) because this is all that is available. All of the programmer's skill and knowledge have already been applied in optimizing this source code. These optimizations spread information. That is, they take advantage of

what is known elsewhere and substitute complex but efficient realizations for (simple) abstractions.

Both of these effects exacerbate the maintenance problem by making the system harder to understand, by increasing the dependencies among the parts, and by delocalizing information.

Requirements analysis, specification, implementation, documentation, and maintenance are all knowledge-intensive activities. **But the current paradigm precludes the use of automated tools to aid these processes because it deals only with the products of these processes rather than with the processes themselves.**

Thus, the current software paradigm must be changed to explicitly represent and support these knowledge-intensive processes. The rest of this report is a description of such a knowledge-based approach to software support, and an identification of the technology needed to achieve it.

2.2 Proposed Solution

This section describes the long range objective of this effort in terms of a shift from the current informal, person-based software paradigm to a formalized, computer-assisted software paradigm and the knowledge-based software assistant that it both facilitates and requires. A more detailed view of the KBSA and its various facets is given in Section 3. The technology needed to support this paradigm is discussed in Section 4, and our incremental approach toward obtaining the goal KBSA system described here is presented in Section 5.

2.2.1 The Basis for a New Knowledge-Based Software Paradigm

The knowledge-based software paradigm of the future will provide a set of tools and capabilities integrated into an "assistant" that directly supports the human developers in the requirements analysis, specification, implementation, and maintenance processes. It will be characterized by the fact that "the machine is in the loop."

- All software life-cycle activities are machine mediated and supported

by the knowledge-based assistant as directed by the developers. These activities will be recorded to

- provide the "corporate memory" of the system evolution

and will be used by the assistant to determine how the parts interact, what assumptions they make about each other, what the rationale behind each evolutionary step (including implementation steps) was, how the system satisfies its requirements, and how to explain all these to the developers of the system.

This knowledge base will be dynamically acquired as a by-product of the development of each system. It must include not only the individual manipulation steps which ultimately lead to an implementation, but also the rationale behind those steps. Both pieces may initially have to be explicitly stated by the developers. Alternatively, explicit statement of the rationale by the developer may enable the automated assistant to select and perform a set of manipulations which achieve that objective for the developer. To make the process possible, it will be necessary to

- formalize all life-cycle activities.

For the knowledge-based assistant to begin to participate in the activities described above, and not just merely record them, the activities must be at least partially formalized. Formalization is the most fundamental basis for automated support; it creates the opportunity for the assistant to undertake responsibility for the performance of the activity, analysis of its effects, and eventually deciding which activities are appropriate. Not only will the individual development activities become increasingly formalized, but so, too, will coordinated sets of them which accomplish larger development steps. In fact, the development process itself will be increasingly formalized as coordinated activities among multiple developers.

2.2.2 Major Changes in Life-Cycle Phases

We have described three major differences between the knowledge-based software paradigm and the current software paradigm – the role of the history of system evolution, the formalization of life-cycle activities, and the automation it will enable – but we have not yet described the changes that will occur in the various phases of the software life cycle itself.

We are shifting from an informal person-based paradigm to a formalized computer-assisted paradigm. This formalization and computer support will alter and improve each life-cycle activity. But our intent is not to incrementally improve the current life-cycle activities, nor even to attempt to make large quantum improvements in them via advanced knowledge-based support. As we have argued, the current paradigm is fundamentally flawed and even large quantitative improvements will not correct those flaws.

Instead, our goal is to alter the current life cycle to remove the flaws and take advantage of the formalized computer-assisted paradigm described above. We therefore focus here on four life-cycle activities that differ in kind, rather than just degree, from current practice. They serve to distinguish the KBSA from incremental improvement of the current life cycle.

2.2.2.1 The Development (Implementation) Phase

First and foremost among these changes will be the emergence of formal specifications (expressed as machine-understandable descriptions) as the linchpin around which the entire software life cycle revolves.

In contrast to current practice, in which a specification serves only as an informal description of functionality and performance, which implementers and testers use as a guideline for their work,

- the actual implementation will be derived from the formal specification.

This will occur

- via a series of formal manipulations, selected by the developer and applied by the automated assistant

which convert descriptions of **what** is to happen into descriptions of **how** it is to happen efficiently. To the extent that these formal manipulations can be proved correct, the validation paradigm will be radically altered. Rather than testing the resulting implementation against the (informal) specification,

- the validity of the implementation will arise from the process by which it was developed.

That is, the development and the proof of correctness will be co-derived.

2.2.2.2 The Maintenance Phase

In order to maintain a program, it will be necessary only to

- modify the specification and/or refinement decisions and reimplement

  by "replaying" the development.

- Systems are not static;

even ones that, via prototyping (see below) match the user's original intent, and are validly implemented via automated assistance require updating. They evolve because the user's needs evolve, at least in part in response to the existence and use of the implemented system. Today, such evolution is accomplished by modifying (maintaining) the implementation. In the knowledge-based software paradigm, such evolution will occur by modifying (maintaining) the formal specification (rather than the implementation) and then reimplementing the altered specification by modifying and "replaying" the previously recorded implementation **process** (the sequence of formal manipulations that converted the specification into the implementation).

This represents another major shift from current practice.

- Rather than consisting of attempts to "patch" the optimized implementation,

- the maintenance activity will much more closely parallel the original development.

That is, first the specification will be augmented or revised (just as it is modified as a result of feedback from the prototyping/specification-revision cycle). Such modifications should be much simpler because they more closely approximate the conceptual level at which managers understand systems and for which their perception is that such modifications are trivial (it is the highly intertwined, delocalized, and sophisticated optimizations that make modification of implementations so difficult). The second step in maintenance is reimplementing the specification. This is another situation in which recording the development process provides leverage. Rather than recreating the entire implementation process, the developer will identify and modify those aspects of the previous development which either must be altered because they no longer work, or should be altered because they are no longer appropriate (i.e., no longer lead to an efficient implementation). Then, this altered development will be "replayed" by the automated assistant to obtain a new implementation.

- Increased development automation facilitates the "replay."

To the extent that automation has been used to fill in the details of the implementation process, as described earlier, the need to modify the development will be lessened as these details can be automatically adjusted to the new situation. In any case, the effort required to reimplement a specification is expected to be a small percentage of that required for the initial implementation, which in turn is expected to be a small percentage of that required for current conventional implementation.

Thus, in the knowledge-based software paradigm, the effort (and corresponding time delay) required both for implementation of the initial specification and especially for incremental modification (maintenance) of that specification, will be greatly reduced. This will allow that saved energy to be refocused on improved specification (matching the user's intent), on increased functionality in the specification (because implementation costs and complexity restrictions have been mitigated), and on increased evolution of that specification as the user's intent changes over time (at least in part because of the existence of the implemented system).

This will produce three of the most profound effects of the knowledge-based software paradigm:

- Systems will be larger, more integrated, longer lived, and will be the result of a large number of relatively small evolution steps.
- Software systems will finally reach their potential of remaining "soft" (modifiable) rather than becoming ossified, hardware-like, with age.
- Evolution will become the central activity in the software process.

In fact, rather than being limited to maintenance after the initial release,

- evolution will also become the means by which the "initial" specification is derived.

The current "batch" approach to specification in which the specification emerges full-blown all at once (often as a several-hundred-page tome) will be replaced by

an "incremental" approach in which a very small formal specification is successively elaborated by the developer into the "initial" specification. These elaborations will occur via semantic manipulations (rather than "text editing") which capture the various types of elaboration (i.e., adding exceptions to a "normal" case, augmenting the functionality of a process, revising an earlier description, and so on). Thus, specifications will undergo a development just as the implementations they describe. Maintenance of the specification, whether after initial release of the implemented system or as part of the elaboration of the initial specification will occur by modifying this development structure rather than "patching" the specification.

2.2.2.3 Specification Validation Phase

Current testing supports more than just the comparison of the implementation with the (informal) specification, it also provides the means, through hands-on experience with the working implementation, to compare actual behavior to the user's intent. Often, if not usually, mismatches are detected and the implementation must be revised.

This second function of current testing will be replaced in the knowledge-based software paradigm by

- treating the specification as a testable prototype.

To make this possible, a subclass of formal specifications, called

- executable specifications must be employed.

Furthermore, some form of

- automatic or highly automated "compilation" must be used to provide reasonable (though not production quality) efficiency for running test cases.

Thus, the formal specification will be used as a prototype of the final system.

- This prototype will be tested against the user's intent.

Once it matches that intent, it will be developed into that final production implementation.

As opposed to current practice, in which prototyping is the exception,

- prototyping will become standard practice

in the new software paradigm because of its ready availability (via automatic or highly automated "compilation"). In fact,

- most systems will go through several prototyping or specification-revision cycles before implementation is undertaken.

2.2.2.4 Project Management

Project management has the responsibility for controlling, and therefore monitoring, all the software life-cycle activities. Currently project managers are severely hampered in this objective by the informal and undocumented nature of these activities and by the fragmentary, obsolete, and inconsistent data now available. In the KBSA paradigm, the situation will be very different. All the life-cycle activities will be formalized, their operation will be mediated and supported by the KBSA, and their progress will be recorded in the "corporate memory."

Thus, all the data needed for effective management will be available through the KBSA.

- Management must define what information it needs for on-line management

in terms of these data.

Furthermore, since the KBSA is mediating all life-cycle activities, the opportunity exists to

- formalize the coordination of activities.
- Management must define the project policies and procedures

to be implemented as protocols between the activities. These policies and procedures describe the operation of the project as a whole in terms of differentiated management styles. They define project organization, resource allocation, states and choices, transition between those states, and authorization of those transitions.

It should be noted that two desirable capabilities have been explicitly omitted from the knowledge-based software paradigm: fully automatic program synthesis (the automatic generation of production quality code from a formal specification) and natural language specification (the translation of an informal description into a formal specification). The rationale behind these omissions is described in Section 5.

To summarize, the knowledge-based software paradigm will differ markedly from the existing paradigm. The basis for this new paradigm will be capturing the entire development process (the identification of requirements, the design of the specification, the implementation of that specification, and its maintenance) and supporting it via an automated knowledge-based assistant. The development process will revolve around machine-understandable descriptions. Capabilities will exist to develop an "initial" specification incrementally from a kernel via a series of formal manipulations, to test the specification against the user's intent by treating it as a prototype (because the specification is executable), to develop an efficient implementation from that specification via further formal manipulations (which co-derive its proof of correctness), and to maintain the system by further developing the specification and its implementation and then replaying that implementation development. This will result in evolution as the central development activity and will produce systems that are longer lived, larger, more highly integrated and which remain pliable to further modification as user needs themselves evolve.

2.2.3 An Automated Assistant

In describing the knowledge-based software paradigm, frequent reference was made to an automated assistant. This paradigm both facilitates and requires the existence of such an assistant, as a consequence of having the whole development processes (requirements analysis, specification, implementation, and maintenance) machine mediated and supported. Thus, these development processes must be broken up into individual activities.

The KBSA will participate in all the coordinated development activities (including the coordination itself) to aid the developers. The existence of such an assistant will, in turn, fundamentally alter the software life-cycle activities, as described in Section 2.2.2, as its capabilities alter the feasibility and cost of these various development activities.

- The KBSA will support the new software paradigm by recording the development activities, performing some of them, analyzing their effects, and aiding their selection.

It is because of the sophistication of the capabilities involved and the fact that several different sources of knowledge will be involved (knowledge of requirements, specification, implementation, evolution, validation, analysis, etc.) that this assistant is called the knowledge-based software assistant or KBSA.

Because KBSA is mediating all development activities, it can support not only those individual activities, but also the development as a whole. It can coordinate one activity with another to maintain consistency; it can alert management if resources are, or are about to be, exceeded; it can help prevent maintainers from reexploring unfruitful implementations. In short, by mediating all of the development activity and by being knowledgeable about that development, KBSA can help people bring to bear whatever knowledge is relevant for their particular task. This is especially important on large projects where it is currently difficult, if not impossible, for people to comprehend and assimilate all the relevant information, which is often fragmented, incomplete, or inconsistent.

The KBSA is an intelligent assistant that interfaces people to the computerized "corporate memory," aids them in performing their tasks, and coordinates their activities with other members of the team.

- The evolutionary creation of the KBSA and the incremental formalization of the development activities upon which it is based is the central theme of our research plan.

As we learn to formalize the various life-cycle activities, we will build KBSA capabilities to perform, analyze, select, and/or coordinate them. Over time, this will allow developers to concentrate more and more on the higher level aspects of the development process and turn more and more of the low-level details over to the KBSA. That is,

- as the development process is incrementally formalized, it can be increasingly automated.

Because, for the foreseeable future, we intend to keep the developer, as well as the machine, in the loop, provision of suitable interfaces are necessary so that the developers and the KBSA can work effectively together.

To summarize, we begin with the commitment to having "the machine in the loop." This will cause the development processes of requirements, specifications, design, implementation, and maintenance to be divided into a larger number of smaller, more formalized steps. This finer granularity and increased formality will enable the emergence of a KBSA that aids developers in coordinating and performing of all of the activities and records those activities as the documentation of the system's development. This incremental approach to formalizing the individual development activities and their coordination, and to providing automated assistance to the developers through the KBSA, is the foundation of the shift from the current informal person-based software paradigm to the new formalized computer-assisted KBSA paradigm.

Related Work

Since surveys are available,and to limit the scope of this planning effort, we have intentionally not prepared a survey of related work. We refer the reader to three references. The first book [2] covers knowledge-based systems in general, and the referenced chapter specifically covers applications of knowledge-based systems to software assistance. The second two references, [3] and [4], review software engineering environments.

§3 KBSA INTERNAL STRUCTURE

In Section 2, we described the KBSA as a single unified knowledge-based assistant that mediated and supported all the life-cycle activities, recorded them to provide a "corporate memory" of the development, and coordinated the activities of the individual project members. Here we consider the internal structure needed to realize such capabilities.

The KBSA is a complex, highly interconnected system. Nevertheless, it is necessary to divide it, both for explanation and creation purposes, into its major functional blocks. There are four of these, as illustrated in Figure 2. The central foundation of the KBSA is the framework, which includes an activities coordinator and a knowledge-base manager.

The job of the KBSA is to validate each activity as it is performed, record that activity, and coordinate it with other activities as defined by formal protocols.

Project management policies and procedures establish those protocols. Its documentation requirements are satisfied by the recorded activity, and its tasking (resource allocation) is handled as a coordinated activity.

The other activities which are coordinated in the KBSA could be grouped in many different ways. We have chosen to group them according to the familiar software life-cycle phases to make them more understandable and to present one feasible decomposition. While this grouping helps us describe the evolutionary staged development of automated support we envision in each area, it is important to remember that major change in the life cycle (as described in Section 2.2.2) will result from the KBSA. Therefore, other groupings may well be more appropriate for the construction of the KBSA. Our choice of groupings and their evolutionary development must be considered illustrative and is not meant to restrict the selection of other groupings or development scenarios.

Eventually, as the integration between these activities becomes tighter, we expect them to lose their individual identity and to become the single knowledge-based assistant described in Section 2. Only then will we have fulfilled the promise of the new software paradigm.

To prevent misinterpretation, we feel it is important to reiterate that although the rest of this section describes the facets separately, the users will see but a single entity, the KBSA, with many capabilities. Instead of a multitude of interfaces, languages, and conventions, users will experience a single KBSA, expert in all aspects of software development.

Finally, there is the support environment upon which such a system is built. It includes version and access control, an inference engine, and user interface capabilities.

This KBSA internal structure is further described in the following subsections. At the end of each section we have provided a set of short and mid-term goals for each facet.

The long-term goals are given in the description of each facet, and in some sections, certain long-term goals have also been included in the list of goals at the end of the section.

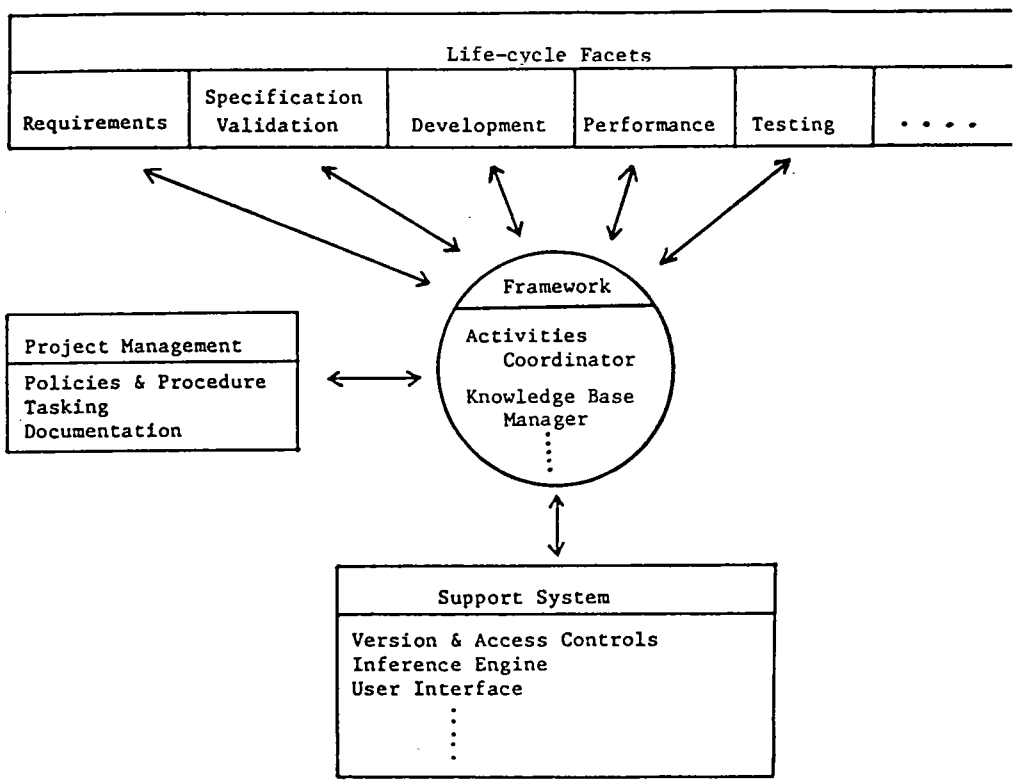

Figure 2. GENERALIZED KBSA STRUCTURE

3.1 Activity Coordination

The facets of the KBSA must be embedded in a large framework and support system, which includes an activities coordinator, knowledge base manager, inference mechanisms, program analyzer, version and access control, user interfaces, etc. In this subsection we have singled out for discussion the novel concept of the activities coordinator; the more familiar supporting components of knowledge-based systems are discussed in Section 3.4, "KBSA Support System," and Section 4, "Supporting Technology."

The development and subsequent maintenance of a large application program or family of related application programs often involve a considerable number of agents – analysts, programmers, test engineers, managers, documentation specialists, users, and so on. The activities being carried on by these agents require various kinds of coordination. For example, suppose that some agent has the task of modifying some program module. Before incorporating the result of the modification into a new release, project management policy may require that certain tests have been performed satisfactorily, that the changes have been logged appropriately, and that relevant documentation has been updated. Furthermore, the approval of some manager may be required before the result can be distributed.

How can such policies and procedures be formalized so that automated support can be provided? Modern programming environments, with their software data bases and integrated tool sets, often already provide some preliminary coordination capablities. For example, they usually provide mechanisms for version control so that one can determine those elements of the software data base that are up to date and those that are not. In addition to being part of a version, a program module derived by some tool (e.g., a compiler) often has a derivation history that relates it to the parent modules involved in its derivation. If one or more of these parents is subsequently modified, resulting in a new version, then the derived module also requires updating.

These environments also include lock and key mechanisms to ensure that only those agents having the requisite authority (the key) are permitted to take certain actions, like modifying certain modules or invoking certain tools.

While such version control and locking mechanisms are certainly necessary, they are not sufficient for the kinds of protocols needed to describe other software development and maintenance activities. Instead, we need a language to describe the types of coordination (i.e., protocols) that exist between the software development agents and an interpreter of that language. This would provide the basis for the formalization not only of the types of coordination, but also of the software development activities being coordinated. As described earlier, such formalization is the basis for the entire KBSA approach to computer-assisted support. Such a language would enable the wide range of idiosyncratic policies and procedures that have successfully been used by managers to be expressed. The interpreter could then monitor and facilitate project development in compliance with these policies and procedures. As with other aspects of the

KBSA, staged incremental introduction of knowledge-based capabilities would enable increasingly sophisticated support from the KBSA with less explicit user direction.

In addition to knowing about the elements of the software data base and the tools available in the tool set, this extended system would have knowledge of agents, both human and mechanized, that participate in the development and maintenance activities and the relationships among them. Rather then being limited to the current mode in which users explicitly invoke separate discrete tools, the extended system would support a collection of ongoing activities with each activity having an underlying protocol that specifies the coordination with other activities. Thus, the environment would be active rather then passive. It would ensure the validity of each agent's actions and instigate further activity from other agents as defined by the coordination protocols.

Communication among the agents and activities would be via messages. These messages would not just be text but would be formal objects in the system that included references to other formal objects – the modules, agents, organizations, activities, other messages, etc. Examples would include queries regarding some element (e.g., a "bug" report), replies to specific queries, requests for permission to take some action, grants and/or denials of such requests, and so on. The movement of a message (plus other messages generated on account of that message) would generate an audit trail that would, for example, enable the determination of the status of or prognosis for some activity that was generated in accordance with a query (for example, the repair of a problem in accordance with some "bug" report).

Each activity ongoing in the system would, at any point in time, be in some state. For each state there would be a set of choices that were possible, some of which could result in the transition to a new state. The inter- and intra-coordination of activities would be accomplished by controlling the choices that were possible at each state of an activity. There could be a number of ways of controlling these choices. One would most certainly be through the usual lock and key mechanisms; an agent could choose a certain action because he had the right (the key) to do so. Another means of control would be to require the agent to obtain formal permission from another agent or organization that had the right to authorize the action proposed. The request for a permission and the grant of the permission would be via formal message objects that were so interpreted by the system. A third means of control would entail a collection of rules that dynamically described the relationships among the various elements of the system.

That a particular choice was permitted or denied would result from demonstrating that the predicate enabling the choice could or could not be inferred from the current state of the activities within the system.

3.2 Project Management and Documentation

This section describes two facets – project management and documentation – that are called out for treatment here in a separate section from the other facets because

they have strong, across-the-board interaction with all the other facets. That is, the power of these two facets contributes to the power of each of the other facets, and is derived from the existence of each of the other facets. For example, the project management facet helps to manage tasks being carried out with the assistance of the development assistant and also derives information from the development assistant. The documentation facet helps to explain specifications, requirements, performance, etc., using information from these facets.

3.2.1 Project Management Facet

The long-term goal of the project management facet is to provide knowledge-based help to users and managers in project communication, coordination, and management tasks that range from simple inquiries about tasks to reorganization of project plans. The goals are to reduce project costs, speed project development and maintenance, manage more effectively, provide greater project continuity, improve project communication, increase software reliability, and improve responsiveness to change. The management facet will assist throughout the life cycle from inception through maintenance. It will provide assistance to all KBSA users, not just managers.

The project management facet (PMF) consists of a formalism, a knowledge base and message manager, and an accompanying set of knowledge-based tools and procedures. All important (designated) project information, communication, and decisions will be formally expressed, recorded in the knowledge base, and available through these tools.

The project management facet uses the coordination and message handling capabilities of the activities coordinator to carry out its work. It is distinguished from the activities coordinator by its domain of discourse and types of decision making (task assignment, etc.). The PMF will use other general KBSA inference and knowledge-base management tools where appropriate. A tutoring system will help human agents (designers, users, and managers) learn how to use the assistant system.

The knowledge base, including the set of scripts and procedures, forms a **semantic model** of the entire project, including its history, and its procedures and policies. The **power** of the PMF derives from being able to use this semantic model or knowledge source to **reason** about the project rather than just act as a data management system. Most activities will have at least an underlying protocol that provides the means for internal coordination with other activities. More complex management activities will have more complex protocols and inference procedures to provide means for reasoning about management decisions, implementing policies, weighing evidence, etc. A uniform interface will allow human or automated agents to make requests for management assistance without having to know details about all the tools.

Short-Term Goals

- Project Management Formalism

  The first step is to develop a machinable formalism for project management knowledge.

  This formalism is the framework that will be used to implement all knowledge base operations, message handling, inference procedures, and other facet capabilities. To the extent that messages, tasks, etc. are not described within this formalism, the PMF will not be able to do intelligent things with them. At first there will be more free-form text associated with these entities, largely incomprehensible to machines, but as the PMF grows, more of this text will be expressed as knowledge in the PMF formalism.

- Knowledge Base and Message Handling

  Using the above formalism and the activities coordinator, the PMF knowledge base manager and message handler will deal with all formalized aspects of PMF knowledge. They will store and retrieve PMF knowledge and send and receive PMF messages. All messages and knowledge base entries will either include descriptors within the formalism or be entirely within the formalism. All important communications and decisions can be recorded, but at first their formalization (which allows indexed entry into the knowledge base) will be manual.

  The knowledge base of project tasks will let managers and other agents keep track of the tasks to be done and keep records of what is completed. (At first, completion will be explicitly reported to the KBSA; later the PMF facet will recognize completion automatically.) Managers will be able to look at the set of tasks and organize and assign them. The task structure will reflect the development of the system and the tasks completed, and those remaining will be explicit and available for study when the project is reviewed.

  An initial set of message-handling capabilities will be developed to allow agents to be assigned tasks and report their progress and to allow human agents to communicate. In the early stages, the arbitration of messages will be entirely by human designer or manager, but human arbitration will gradually receive increasing knowledge-based assistance. An interface will allow people to understand and monitor the formal messages to and from automated agents.

- Task Tracking

  The above formalism, knowledge-base system, and message-handling system together lay the necessary groundwork so that the PMF can be extended by the addition of simple project management procedures and deductive inferences. For convenience, we will group these simple procedures and inference capabilities under the heading of task tracking. The inferences will require that dependency links, messages, and other items can be traced through the knowledge base.

Scripts for project management disciplines or paradigms will be developed and used to guide or enforce these management disciplines.

Mid-Term Goals

- Suggesting Simple Management Decisions

  Using all the above tools, the capability of the PMF will then be extended so that it will suggest simple management decisions. The inferences made in this decision making differ from those inferences made in the task tracker, in that the decisions here require weighing of evidence and more detailed models of tasks and agents. The decisions will still be limited to relatively local decisions about particular tasks or agents, however.

- Plan and Procedure Creation and Modification

  Using all the above tools, the PMF will be extended to generate or modify plans and procedures. At this stage the PMF will deal with entire plans and procedures and carry out significant refinements and transformations of them. Transformational methods developed in the development facet could be brought to bear on the problem.

- Knowledge Acquisition

  Extensions of the above tools will allow simple knowledge aquisition by having PMF knowledge available and all transactions capturable and manipulable within the formalism.

3.2.2 Documentation

The long range goal of the KBSA is to provide the project manager and each project member with the equivalent of an expert on personal call to answer specific questions on any aspect of the project or the software being developed. For example, a user may inquire about the possible arguments to a command. A system developer/maintainer may inquire about the purpose of a particular line of code. The project manager may want to know the testing status of a particular module. In all these cases, the KBSA could answer their questions because the relevant knowledge has been captured and formalized as part of the software development process. In a sense, all of the knowledge used in each of the KBSA's activities is available for explanation and documentation.

Given adequate underlying knowledge, the main issues in documentation have to do with how to communicate this information cogently. For example, what constitutes a good explanation? How and when is it appropriate to summarize information? One good way to explain something is to identify it as an instance of some familiar general class, such as "this is a temporary variable" or "this is a kind of directory listing command." Another effective type of explanation is to describe the role of a thing in some causal or goal structure, such as "the setting of this flag causes the following actions to occur" or "the purpose of this test is to guarantee that the input satisfies

the following condition." The feasibility of automatically generating explanations to unforeseen queries about the internal workings of a complicated program is being explored in current research.

An example of the use of the documentation facet is in project management. Maintaining up-to-date and accurate documentation is crucial to the management of any large software product, as well as in providing help and tutoring facilities for the users of the project management facet. The initial documentation facet will be for experienced managers, designers, and programmers. The help facilities initially will be for these experienced people and will be similar to current help facilities. Once the KBSA has evolved enough, it will have an environment that includes naive users as well as designers and managers with a range of experience. The KBSA will include many automated agents and will be used for production systems and maintenance. Therefore, tutoring capabilities will be added that go far beyond the original help system. The tutoring will allow new (and old) team members to learn about the assistant itself (and about other agents) and about the state of the project (task structure as well as design decisions and code state).

The most obvious benefit of this kind of explanation as compared to current documentation practices is that the information delivered is more focused and directed to the specific needs of the person inquiring at the time. However, the greatest benefit of this technique results from the fact that the underlying knowledge from which explanations are drawn is necessarily kept up to date because the KBSA mediates and supports all project activities. In the best current practice, most of the underlying decisions are lost from the beginning.

Short Term Goals

- On-Line Documentation

  There are many fairly standard kinds of documentation for various different audiences that are now in common use: " A, B, and C specs," hierarchical flowcharts, user reference manuals, "help" files, and so on. In the short term, it will not be possible to formalize much of the knowledge in these documents; most of it will have to remain in the form of text strings to be read and interpreted by the user. However, it will be a step in the right direction to provide a central data base with a defined (and possibly extensible) vocabulary of structuring primitives available to all agents in the software development process throughout the entire life cycle (this is an instance of KBSA's "corporate memory"). Furthermore, by cross-indexing this documentation to other parts of the software that are also kept on-line, such as the code or the requirements, it will be possible to automatically monitor whether the documentation is being kept up to date. Finally, it will be possible to automatically generate various kinds of standard-format documents using specially written procedures that read the appropriate subset of information out of the data base. Examples already exist in which a simple hierarchical data base (with text files at each node) is used to maintain the status of all modules for project management purposes. Final

deliverable documents could then be automatically generated from the same data base by combining the text files with standard boiler-plate.

- Partially Formalized Documentation

  To effectively increase the degree of formalization of the documentation, we propose to reduce the "chunk" size in the data base and extend the vocabulary of keywords describing the chunks and their relationships. At this stage, the chunks of unformalized text should not exceed the size of paragraphs and might often be smaller, such as a single line describing the purpose of a variable. One benefit of this fine-grain structure is to allow an incremental change in the software to require only incremental effort in revising the documentation. The pointers between the software and the documentation help to localize those parts of the documentation that are affected by a particular modification to the software. Also at this point, one could begin to design protocols for accessing and perusing the documentation which adjusted to the user's level of expertise, prior knowledge, and so on. With these facilities the emphasis begins to shift from "documentation," which suggests static pre-formatted text, to an "explanation" dynamically generated in answer to specific questions in an interactive relationship.

Mid-Term Goal

- Partially Automated Knowledge Acquisition

  In mid-development of the KBSA, the system developer's burden of being the sole source of documentation information will begin to be lessened by having the KBSA automatically gather and record the knowledge needed for explanation and documentation as a by-product of other system development activities. For example, a natural by-product of using the KBSA is the knowledge about design decisions which is needed for reference during future modifications. Another general source of information is various kinds of program analyses, such as those performed for performance optimization.

3.3 KBSA Facets

This subsection describes an example set of KBSA facets selected to aid comprehension by their correspondence to current life-cycle phases. By selecting and describing these specific facets, one particular view is provided in sufficient detail to define what would suffice as a KBSA.

3.3.1 Requirements

The long-term goal of the requirements facet is to provide the following: comprehensive requirements management, intelligent editing of requirements, testing of require-

ments for completeness and consistency (both self-consistency and consistency with application domain models), performing requirements reviews, maintaining and transforming requirements in response to changes, decomposing and refining requirements into executable specification languages, and acquiring requirements knowledge. The knowledge base available for these actions will include both general and application-specific knowledge.

Requirements will be acquired by KBSA via dialog with end-users (systems analysts will have to be used until the level of this dialog becomes sufficiently high-level and application-specific). These end-users will define and modify the requirements and behavior of their desired system by a combination of high-level, domain-specific requirements languages, examples, traces, state-transition diagrams, graphics, and so on, in whatever mix they find comfortable. The process will be a mixed-initiative dialog, where the sequence of statements need not correspond to the organization of the final program. KBSA's role is to have enough knowledge about requirement analysis and about specific application domains to be able to accept and process these descriptions. The requirements facet will organize the stated requirements and incorporate them into existing descriptions. It will notice inconsistencies and missing parts of the requirements, and suggest remedies, fill in pieces, and point out trade-offs whenever it can. The facet will also, on request, describe the current state of the requirements specifications in natural language, graphically, or by simulating the behavior of the system as much as possible. The facet will help integrate new requirements into an existing requirements specification and will use knowledge-based program refinement techniques to help transform these requirements into executable specification languages.

Knowledge-based tools for the requirements facet will have a high payoff. Because the lower-level program development and management tasks will be increasingly automated and will take place in the background with less and less human intervention, requirements definition and specification will be of increasing importance, with most of the human effort in software development eventually going into this process.

Software development efforts today do not approach such an ideal. In most projects, requirements are largely unformalized and stated in natural language. Current requirements languages do allow some formalization, principally in the characterization of dependencies, but requirements are rarely machine comprehensible to any significant extent. An additional consequence of informal requirements statements is that requirements usually cannot be executed in any conventional sense.

A formal requirements language will allow, and in fact will demand, knowledge-based requirements processing. The reason is that formal requirements are incomplete; they only partially describe the intended behavior of any system. For them to be understood and processed in some meaningful sense, these partial descriptions must be integrated and completed in some reasonable manner. Such inference capabilities are prototypical of the type of assistance required by the KBSA facets. This formal language might also be executable to allow rapid prototyping (see "Specification Validation" Section 3.3.2, for a more complete description of rapid prototyping). It is important to

note that it is unlikely in the near future either extreme of machine understanding of natural language requirements descriptions or formal languages for complete requirements specifications will be realizable. However, a knowledge-based facet could provide capabilities that allow requirements to be combinations of formalized specifications, machine-understandable but restricted natural language, keyword recognition, and unparsed text strings. The KBSA effort might use whatever natural language comprehension technology becomes available, but it is not committed to, or dependent upon, advances in this area.

A few knowledge-based software systems have been built that dealt with requirements specifications and have helped determine the consistency of these descriptions. They demonstrated the basic feasibility of formal requirements analysis, despite the added difficulty of working with restricted natural language.

Since generation of natural language is a more tractable problem than comprehension, paraphrasing or summarizing requirements definitions from multi-format presentations is possible. This is an achievable and valuable capability for helping people to handle the complexity of large systems and could be especially useful in validation activities.

Domain models for different application areas will facilitate requirements definition. These models give the requirements facet more knowledge to help understand user descriptions, to notice inconsistencies, and to suggest missing parts of descriptions. Since the potential range of applications areas is quite broad (and includes research topics such as reasoning about time and space), it is unlikely that a complete set of domain models can be supplied in advance. However, some simple and frequently used domain models are likely to be available.

Requirements definition can also be viewed as a knowledge-acquisition problem. The requirements activities will consist not only of acquiring new requirements descriptions, but also of acquiring models of new application areas. These activities will draw from the research areas of knowledge acquisition (including mixed initiative acquisition of domain models from experts), problem reformulation, rule-acquisition, inductive inference of requirements from examples, etc. Some practitioners of the requirements analysis art feel that a very important part of their task is generalizing and structuring the user's ill-defined needs. As the lower levels of software production are increasingly automated, requirements acquisition will become the main interface between the user and the programming environment. This is an exciting and high payoff area of research.

Short-Term Goals

- Analysis of Requirements Problem Definition

  There has been less research on knowledge-based tools for the requirements level than for the later phases of the software development life cycle. Accordingly, less is known and further problem definition should occur in the early phase of the KBSA project. The first year of work on the knowledge-based requirements facet should include a planning phase to review and refine the short-, mid-, and long-term goals.

- A Formal Requirements Language

  An initial requirements language will be designed that allows a combination of formal specifications and text strings. This very high level language (VHLL) will probably be an extension of the very high level specification languages being developed today. In the early stages of the KBSA project, new insights will arise about the KBSA life cycle and its effect on this facet. By the mid-term, these should be incorporated into a revised VHLL for requirements. The language will also describe histories of requirements modifications and refinements.

- Smart Editing and Managing of Requirements

  Knowledge-based editing and management capabilities for the requirements facets, including help facilities and a friendly user interface, are important aids to the requirements definition activity. A variety of specification fragments, including both a formal language for requirements and text strings, need to be managed. For convenience, even text strings will be handled in simple ways such as storage, retrieval, keyword analysis, etc. Dependencies among requirements will be specifiable.

  An intelligent editor will be used to create and modify requirements definitions. At first, the editor will ensure only that the syntactic structure of the formal requirements language is followed. Next, the editor will be used to trace through the connections of related requirements during editing to ensure consistency. Later, generic requirements descriptions (for example, an input-process-output sequence) will be stored in the knowledge base. Such descriptions can be used as models to fill in and can be matched against a user-created description. These models will be used to check completeness and may provide additional consistency tests.

- Reviewing Requirements Definitions for the User

  Getting the user to view the requirements in a new light and possibly see problems or opportunities is an important capability for producing requirements descriptions. Review methods could include paraphrase in natural language, graphic displays of domain models in the knowledge base, executing the specification, writing stubs or facades that demonstrate the format, if not the content, of the specified system, and rapid prototyping to help determine behavioral requirements.

- Requirements Testing

  Simple inferences can be used to help determine the adequacy of requirements. One source of help in requirements definition is the adaptation of capabilities and paradigms from lower levels of the KBSA. Some activities can be carried out without any domain knowledge–consistency checking, analysis, and explanation, for example. The first set of requirements tools will therefore check for a simple heuristic kind of completeness and consistency. These inference procedures can extend the requirements editor's ability to either fill in details or test against stored general knowledge. In addition, by employing traditional, non-knowledge-based analysis, the requirements tools will detect entities that are undefined, entities defined but never refer-

enced, data flow anomalies, etc. Later, for suitably limited domains, we can include the performance facet at the requirements level to help in assessing the cost of desired features.

Mid-Term Goals

- Incorporating Domain Knowledge into the Requirements Capabilities

  Simple models of frequently used domains will be developed. The first model will be for a fairly narrow domain or application (e.g., simple classification programs). The models will supply the knowledge base that will be used for domain-specific support of the requirements capabilities.

  As domain models are added, the requirements capabilities will be augmented to take advantage of the new knowledge. For example, domain knowledge will be used by the intelligent editor/manager to retrieve application-specific, previously described, or generic requirements descriptions from the knowledge base that match the user's current needs. These descriptions will serve as useful models to be compared to the specified requirements to check consistency and completeness. By employing more sophisticated techniques such as symbolic evaluation of the requirements language and some inductive inference, more application-specific inconsistencies can be inferred.

- An Automated Structured Walk-Through System for Requirements Engineering

  Many of the capabilities described above will be combined in a script (process description) and applied with a form of symbolic interpretation. For example, a structured walk-through tool based on a fault model representation and on a requirements language will aid an expert systems analyst to keep track of loose ends and problem areas. It accepts requirements as input, together with other management information (e.g., who should approve it, who heads up the prime user groups, who heads the implementor group). Such a tool will be able to perform some useful background analysis for missing or incompatible requirements.

- Requirements Transformation and Refinement

  At this stage, techniques from knowledge-based program synthesis could be extended to allow transformations of requirements. For example, if a program is set up for monthly reports, and weekly reports are required, the knowledge base could supply descriptions of the necesary changes to make. Depending on the level of difficulty, the facet might either suggest and remind the user of the kinds of changes, or actually carry out the transformations automatically. Requirements refinement is the other type of transformation. In this case the requirements are brought through successively more detailed stages until they reach the level of executability. This type of decomposition and filling in of detail is exactly what happens in program refinement discussed in the development facet (Section 3.3.3), but higher-level knowledge is needed here. The facet suggests alternative refinements and decompositions. The transformations may be manual, interactive, or automated as fits the situation.

- A Requirements Tutor

  Tutoring capabilities at the requirements level will help new members of the software design team to start contributing sooner. For example, tutoring will help them learn to use the software assistant's capabilities. It also will help them to understand the current configuration of requirements specifications and the previous decisions that provide the context for new requirements decisions.

3.3.2 Specification Validation

Eventually, formal specifications will be developed using KBSA and starting from informal requirements. Specifications will be the first formal representation of the system to be built. As in all other areas, this representation must be formal for KBSA participation and support. Furthermore, it is crucial for the KBSA paradigm that the specification language be executable so that the specification can be used as a testable prototype and so that source-to-source program transformations can be used to convert it into an efficient implementation.

As the first formal representation of the system to be built, the question arises as to whether this formal statement matches the user's original intent. Even though the specification is much more abstract than the implementation, it is still complex for real systems. Therefore, the first formal specification will usually be wrong and will have to be "debugged." In fact, several debugging cycles will normally be needed to get the specification correct.

Since we are dealing with a specification rather than an implementation, we use the term "validation" rather than "debugging" to describe this process. Because the formal specification is being compared to the user's informal intent, only the user can make this comparison.

Three techniques exist for validating the specification: prototyping, static validation, and dynamic validation. They are complementary and will be intermixed in practice.

Prototyping consists of running test cases on the specification. This is theoretically possible since the specification is executable. However, to achieve reasonable efficiency (so that test cases can be run), considerable optimization must be achieved. This would either be done via a partial interactive development or, preferably, by a smart compiler capable of producing testable, rather than production-quality, code. Such prototyping has all the strengths and weaknesses of current testing. Specific cases can be tried quickly and easily and can expose some bugs rapidly, but such probing is far from comprehensive.

The second validation technique is static validation, which consists of paraphrasing the formal specification in natural language so that an easily read form is available for the end-user to conduct a design review (as is being done with manually produced B5 specifications). Two advantages arise from such paraphrasing: first, formal

specifications in any language are hard to read and comprehend; second, by regrouping the elements, a different view or perspective is presented which also aids comprehension.

The last validation technique is dynamic validation, which is an extension of the prototyping technique. Rather than running specific test cases, symbolic execution will be used to characterize all the behaviors produced for an entire class of test cases. In order to understand the set of such behaviors, an explanation must be produced which characterizes the "main line" and then details the exceptions and/or augmentations that are test-case specific. A mixture of natural language and graphic animation will be the medium of such an explanation.

Short-Term Goals

- Executable Specification Language

  We propose to develop a high level specification language that is still capable of being executed (albeit extremely slowly). Note that interaction with the development facet will occur via the specification language. The KBSA specification language must be both executable and wide spectrum.

- Specification Wellformedness Checking

  This would include the capability to check for internal consistency within a specification (e.g., all types and actions used are defined, number and type of actual arguments agree with the formal argument).

- Specification Testing

  It would be important to develop the capability to run test cases, both concrete and symbolic, on the specification.

- Specification Paraphraser

  The capability to automatically paraphase a formal specification in natural language (to make it more comprehensible, especially to end users) should include the ability to identify which portion or portions of the specification to emphasize.

Mid-Term Goals

- Rapid Prototyping

  This would entail developing the capability to automatically (or at least nearly automatically) compile a formal specification to an efficiency level that permits realistic testing of the specification as a prototype.

- Self Consistency Checker

  This would include verification of satisfaction of formal requirements, establishment of pre and post conditions, and detection of deadlock and starvation.

- Behavior Explanation

  We propose to develop the capability to explain in natural language the behavior of a

specification (as opposed to just the result produced) on both concrete and symbolic test cases.

Long-Term Goals

- Summarize Behavior

  The idea is to develop the capability to automatically summarize specification behavior in natural language for different audiences and experience levels (e.g., highlight surprising results or normal case behavior).

3.3.3 Development

The job of this KBSA facet is to aid the creation of a production quality implementation. Since the full functionality of the intended system has been captured in the formal specification, that specification "merely" needs to be compiled to accomplish this task. Unfortunately, even smart, knowledge-based compilers, may not be capable of producing production-quality implementations. The reason is that to the extent that the specification language is fulfulling its purpose as a description of what rather than how, the gap between the formal specification and an efficient implementation is too wide to bridge totally automatically. Therefore, we will need to keep people, the developers, in the implementation loop. What should their role be and how can we aid them in that role?

These questions can best be answered by considering a related question: what implementation functions are difficult to automate (and hence will be performed by the developers)? The answer is simply, the decision-making portion. The implementation process consists of numerous implementation decisions such as how to represent some information, what algorithm to employ to obtain some result, what information to save, when and how to recompute that information not saved, etc..

There are three difficulties in automating these decisions. First, the decisions are not independent. The choice made for one decision often affects which choice should be made for another. Second, techniques for evaluating the relative values of different choices in the presence of other unmade decisions are quite limited or nonexistent (partly because of the interactions among these decisions). Finally, little is known about the order in which these decisions should be considered (good designers are observed to employ very different orderings).

These difficulties argue for a continuing role for the developer as decisionmaker in the implementation process. But what of the rest of the process? It consists of carrying out these decisions. Currently this is done all at once, after most (or all) of the decisions have been made, by incorporating them in the code of the implementation (the first and only formal representation of the system).

In the knowledge-based software paradigm, this process will proceed very differently. First, each decision will be captured as it is made to document the development process. Next, it will be realized in the "specification." That is, portions of the specification will be replaced with pieces of "implementation." As later decisions get made and realized, other pieces of the specification will be replaced, or the replacements themselves may undergo further refinement. Many such levels of implementation may occur before the final efficient implementation is obtained. Thus, through realizing the decisions as they get made, implementation will become a process of gradually replacing the constructs in the specification language by those in the implementation language. Since this replacement is gradual, the specification constructs must coexist with the implementation constructs. This requires a "wide-spectrum" language that contains both the specification and implementation languages as subsets.

The gradual refinement of the specification is accomplished via formal manipulations that realize the implementation decision chosen by the developer. Such formal manipulations are possible because the specification, and all its refinements, are formal (i.e., expressions in the wide spectrum language), and are necessary because such manipulations can be quite complex (as sophisticated algorithms replace simpler ones) and quite distributed (as information is spread through optimization). Automation is needed both to ensure that the manipulations are correctly performed (this presupposes that the transformations have been formally verified) and performed everywhere that is required and because the sheer magnitude of the task would be overwhelming otherwise. Fortunately, such automation of the formal manipulations required to realize decision making appears quite feasible and several prototype systems exist that accomplish implementation in this manner. Furthermore, the codification of programming knowledge in catalogs of such formal manipulations has already begun.

But experience with incremental implementation systems has shown that in addition to the implementation decisions, many other formal manipulations are required which either "prepare" the specification for the decision being realized or "simplify" the result of that realization. These low-level manipulations are much more numerous than the decisions made by the developer, and their employment must also be automated. In fact, the set of developer decisions forms a rich hierarchy (actually a heterarchy) of preparatory and simplification manipulations for each other. This raises the possibility of having the developer make only the "conceptual" or "strategic" implementation decisions with a knowledge-based problem-solving tool filling the remaining "tactical" implementation decisions automatically. Advanced versions of the knowledge-based software paradigm will employ such capabilities.

In this incremental implementation process, the automation of the formal manipulation will ensure that the resulting implementation is correct (i.e., is functionally equivalent to the specification). This means that the current phase of testing the implementation can be eliminated. The energy thus saved will be shifted to validating the specification (ensuring that it matches the user's intent, as described in Section 3.3.2) and evolving the system as the user's requirements change.

This brings us to the question of reimplementation. In the knowledge-based software paradigm, maintenance will be performed by modifying the specification (which is normally straightforward and simple) and then reimplementing that specification. But rather than repeating the incremental implementation process from scratch, the KBSA will help the implementer modify (normally only slightly) the previous incremental implementation, which will have been automatically recorded, and then replay it to obtain the new implementation. This reimplementation facility is another powerful automated tool for the developer. Furthermore, to the extent that the original (or previous) incremental implementation was achieved by the KBSA that filled in the "strategic" developer decisions with the remaining "tactical" ones, this development will tend to be automatically self adapting to changes in the specification and/or any changes the developer wishes to make in the decisions previously made.

Short-Term Goals

- Wide Spectrum Language

  We will develop a wide spectrum language capable of representing the design of a system in all stages from formal specification through optimized implementation.

- Transformation Language

  This language should be capable of describing transformations from the more abstract constructs within the wide spectrum language to the more concrete.

- Property Language

  This should be a language capable of describing the properties of program segments (such as the variables set and referenced, the module involved, the criteria under which it is reachable, the effects it creates, and the invariants it maintains.)

- Interactive Mechanical Development

  The idea is to develop a system capable of performing and documenting the development steps selected by the user. This requires the creation of a catalog of transformations.

- Automated Property Proving

  The aim is to develop an inference facility to automatically prove (or disprove) properties as they are needed during development.

Mid-Term Goals

- Automated Development

  We propose a system capable of taking a simple goal stated by the user and creating a short sequence of development steps to achieve that goal.

- Automated Replay

  We propose a system capable of adapting a previous development to an altered

specification with a degree of automation commensurate with that available in the original development.

Long-Term Goal

- Enhance Replay

  This would mean extending replay capability so that, in addition to changing those designer decisions that had to be changed for correctness, the system also detects those which ought to be changed for performance reasons and suggests appropriate changes. Notice that this entails interaction with the performance facet.

3.3.4 Performance

The long term goal of the performance facet is to help to create and maintain efficient programs that meet their performance requirements. The performance facet will guide performance decisions at many levels from requirements specifications to very-high level programs to low-level code. Performance assistance capabilities are critical for making practical tools of very-high-level, executable specification languages. Because the key disadvantage of such specifications is their lack of efficiency when executed straightforwardly, the important factor in their utility is being able to find efficient implementations. During development, efficiency estimation will be used to predict and compare the costs of proposed alternative data structure choices. With this capability, either a programmer or an automated program synthesizer can select a data structure. KBSA will also give performance advice about what control structures to use, what optimizing transformations to apply, and what algorithms to use. Thus, program analysis includes not only data flow and control flow analysis, but also higher-level analysis, such as algorithm analysis, to determine the time and space efficiency of programs, to suggest modularizations, and to find bottlenecks. It also involves augmenting application domain models to include some cost information. At the requirements level, advice will be given about the relative costs of different proposed features.

Currently, most efficiency estimation and optimization is performed by designers and programmers without much automated assistance. There are a few tools for estimating program timing information, and some data flow information is derived by compilers. However, the information is usually neither available in machine-understandable form nor available outside the compiler.

Performance advice can be given regardless of the degree of automation in the development phase. We assume that some combination of the following three development methods will be used: manually implementing programs from specifications; interactively synthesizing programs by applying transformations; and automatically synthesizing programs by a system that selects and uses transformations, simplifications, and

inferences. In the case of automated synthesis, it is the efficiency estimator that bridges the gap from interactive synthesis to automated synthesis.

In all these cases the user, programmer, or knowledge-based assistant searches a space of possible combinations of implementations and decides among them on the basis of knowledge and their relative efficiency. Other factors come into play, such as the amount of effort (human or machine) available to implement the program and the relative importance of the particular part of the program being implemented. For example, a human programmer or a synthesis program might well try to find the most important bottleneck in a program and allocate the largest optimization effort to that portion.

Efficiency analysis facts can be gathered in several ways: rule of thumb estimations, algorithm analysis, or simulation coupled with statistics-gathering. By simulation we mean either directly executing the specification or executing an automatically compiled prototype implementation. By having default implementations for all levels of refinement, we could ensure that, at any time during program development, the program can be quickly implemented (even if it has been only partially refined or optimized). These multiple-level executable specifications can be used for both validation and collecting performance statistics. When analysis and simulation both fail, the fallback position will be to implement various versions and measure their performance. The feasibility of this technique will be dependent upon the cost of creating multiple implementations, which will in turn depend strongly upon the degree of automation of the development phase.

Efficiency estimation is also valuable in the knowledge-based project management and requirements activities, for example, a bottleneck analyzer can locate causes of delays in implementation. As with processor allocation, projects could be reallocated to the most efficient implementors, taking into account their workload and cost.

Short-Term Goals

- Symbolic Evaluation

  Symbolic evaluation (see Section 4.3.1) is a basic analysis technique that is useful in many of the KBSA facets described. However, it is crucial for the performance facet. The performance facet needs to be able to propagate and integrate efficiency estimations and to perform symbolic analysis on partial specifications.

- Data Structure Analysis and Advice

  A short-term target for the performance facet will be a set of estimators for data structure selection that are reasonably robust when handling conventional data structures (probably excluding external memory devices). These estimations could be used for automatic data structure selection or for advice to manual implementors. Efficiency estimation activities will be limited to those necessary for data structure selection, including the use of both rules of thumb and heuristic algorithm analysis. As an initial target, efficiency estimation will provide approximate, average-case

performance analysis. The agents will compute and transform annotations about efficiency characteristics as programs are transformed, and will record cost analysis decisions for the benefit of future users. Some bottleneck finding also should be feasible in the short term; it is valuable for both automated and manual systems, and is a fairly straightforward extension of the basic performance analysis capability. By limiting the performance facet initially to data structure selection advice, we take a conservative position and increase the likelihood of success. It may be necessary, if certain applications are undertaken, to include other optimization decisions.

- Subroutine and Module Decomposition Advice

  One class of performance decision is when to create new subroutines or modules. Given a definition of a potential subroutine, the decision about whether it should be kept as a subroutine or compiled in line is relatively easy, and such a capability will be developed as a useful adjunct for the manual programmer. However, the ability to logically find or formulate subroutines or modules that share substantially the same function is a more complex task and may require inductive inference. Such advice may not be available until later in the project.

Mid-Term Goals

- Domain Models for Analysis

  Once domain models have been developed to help with other activities such as requirements definition, they will be augmented to cover performance or other analysis information. This domain information will be an inexpensive replacement for information that would otherwise have to be gathered by some form of simulation and monitoring.

- Algorithm Design Analysis and Advice

  The data structure analysis and advice capability will be extended to include the ability to analyze control structures and other classes of optimizations. Some optimizations are almost always performed when possible (such as combining two enumerations through the same set) and thus are not especially interesting for efficiency analysis, but their effects need to be understood so efficiency characteristics can be updated. Also, combinations of optimizations sometimes need to be compared (say to determine file aggregation). Determining how to apply some of these transformations and deciding which combinations are really most efficient is a difficult problem.

  We could also consider the effects of simplifying the cost function the user specifies. For example, additive cost functions are much easier to compute and may be sufficient for the user's needs.

- Real-Time Performance Advice

  Real time systems are an important application domain. To achieve analysis and advice in these domains, the degree of completeness and accuracy of performance

estimation will be improved to deal with worst-case performance. Better analyses will be available, and the ability to specify different accuracy goals for analysis will be provided. While in many cases a fast, approximate estimation is sufficient, for important cases (bottlenecks, real-time critical response programs) a more expensive analysis, taking closer account of interactions, should be available.

In the longer term, even more sophistication might be attempted, such as taking into account statistical distributions on input data.

3.3.5 Testing

In current software practice, program testing is a haphazard activity, generally not supported by sophisticated tools. In the best current practice, a set of test cases is defined at the beginning of the project, before detailed design has taken place, and put aside to be run after the final implementation is complete. More typically, test cases are generated after implementation has taken place with a view toward "exercising" all parts of the code. Test cases are almost never kept up to date during the long-term maintenance and evolution phase of the typical software life cycle.

In the long term, program testing will disappear as a separate activity in an automated, knowledge-based software development process. Most of what we now think of as program testing will be redistributed into the validation and development activities discussed in preceding sections. To understand this redistribution, we need to reexamine what a test case is and how it functions in the program development process. Fundamentally, a test case has two features: it is a small fragment of the total behavior of a system, and there is some sense in which that behavior can be judged correct or incorrect. The purpose of a test case differs, depending on whether it is primarily concerned with the specifications or with the implementation of a system.

From the point of view of specifications, the fragments of the total possible system behavior selected for test cases are determined by knowledge of the application task. The purpose of defining a set of test cases and their correctness conditions is to help clarify what the user desires. In the mid term, program testing should therefore begin to be coordinated and integrated with requirements and specification validation (see Sections 3.3.1 and 3.3.2). For example, the emergence of executable specifications will make it possible not only to define and record test cases early in the development process, but actually to run test cases before the bulk of the implementation is begun. The benefits of this methodology will be both in the area of helping users figure out what they actually want, and avoiding effort wasted in implementing what turn out to be incorrect specifications.

Automatic generation of test cases based on specific knowledge about the user and the application is also a possibility. This knowledge may be either in the form of domain-specific test generation procedures or precompiled, but highly parameterized, test cases for specific types of applications. For example, the KBSA will have knowledge about

how to generate test data for specific computer-controlled hardware devices, such as a radio scanner or a motor mount.

A number of automatic test generation tools already exist which, given a complete program in some high level language, produce test input data guaranteed to satisfy some form of completeness property over the program, such as traversing each branch point in each direction. The main weakness in this approach is that the tools are in a sense too general – they treat all parts of the program the same, and at the code level. There is no way to incorporate specific knowledge of either the application domain or the software design. Given that program testing is inherently a partial process (i.e., in real software one can never test all possible input data), the advantage of the knowledge-based approach over uniform test generation algorithms is the use of specific knowledge to increase the density of tests in the areas of most relevance.

The second major purpose of current program testing has to do with the implementation process. The purpose of test cases from this point of view is to compensate for the fact that implementing a large and complicated software design is an error-prone process. Here, the fragments of the total system behavior selected for test cases are determined from knowledge of the software implementation design. In the long term, most of this kind of program testing will become unnecessary because a more formal program development methodology (see Section 3.3.3) will allow the interacting properties of different implementation steps to be explicitly managed and checked by automatic or semi-automatic tools.

Knowledge-based methods will also be applicable to the generation of test cases which address program implementation needs. In this area, the specific knowledge has to do with how to properly test specific kinds of software design structures, such as a multi-level interrupt system or a hash-sorted data base. As with application-specific test generation, this will be achieved through a combination of design-specific test generation procedures and libraries of parameterized test cases.

Short-Term Goal

- Test Case Maintenance Assistant

  The first step toward more automated, knowledge-based program testing is to provide tools that better support the current best practices. What is called for immediately is a uniform mechanism for associating test data with every unit of a software project (e.g., a requirement, specification, module). The purpose of a test (which may initially be only a keyword meaningful to the user) should also be recorded with the test itself. The main functions of the KBSA at this level will be to accept changes in test data, to schedule the running of relevant tests automatically when units undergo changes, and to give notification of problems. Such a facility will make it easier and therefore more likely for the system developer to define a test case at any point in the software development process at which it naturally comes to mind. Also, with more detailed knowledge about the relationship between specific test cases and features of the requirements, design, and implementation, testing will become much less of an

all-or-none business, as it is today. A knowledge-based test-case maintenance system will allow incremental rerunning of test cases appropriate to the particulars of the modification.

Mid-Term Goal

- Knowledge-Based Test Generation

  In the mid term, it will be possible to begin to move from simply maintaining user-provided test cases to some automatic generation of test cases. The same underlying test case maintenance facilities can then be used to keep track of a mixture of user-defined test cases, test cases generated by uniform, automatic procedures, and those generated from specific domain and design knowledge. One of the first knowledge sources to exploit for automatic test case generation may be software "fault models," which are accumulations of heuristics, based on past experience, for the kinds of errors that correlate with specific kinds of tasks and programming structures.

Long-Term Goal

Testing will disappear as a separate activity; it will be redistributed into the validation and development activities.

3.3.6 Reusability, Functional Compatibility, and Portability

Many costly problems in present software production are essentially special cases of a general problem which we refer to as **compatibility** between modules. For example, a software module is **reusable** if it can be used as a component in differing systems; the facilities it **exports** must meet the requirements of a component and the facilities it **imports** must be provided by other components of the system. A module is **portable** to a new installation if the facilities it requires (imports) are provided by that installation. In the long term, complex systems will be hierarchically specified in wide spectrum specification languages (see previous sections); interface specifications will be separated from implementation details; the various VHLL's will eventually be rich enough to express all manner of complex details such as timing and I/O requirements, so that one may expect to have available the design of a complete system (hardware and software). At this time, many of our current problems will boil down to checking the compatibiliy of module interface specifications.

This section proposes a spectrum of KBSA facilities that provide assistance in determining compatibility of modules. These components could be developed in the short- and mid-term phases of the project and could become useful tools in production-quality programming support environments in the mid term. The long-term focus is to develop support technology for automating aids to the general modular interface compatibility problem. The long-term tools evolving from this effort will provide components for other KBSA facilities in requirements, validation, and testing.

Short-Term Goal

- An Automated Structured Walk-Through System for Software Portability

  This capability will help check the transportability of software packages between different installations and machines. It will accept and record information about various computer installations, and give advice on the system constraints on software currently in force at a particular installation when asked.

  The capacity for ensuring portability will accept interface specifications about various computer installations. Initially, these interface specifications will be highly restrictive, but they will include I/O requirements and limits, file access and privacy conventions, memory limitations, and run-time scheduler interface specifications. Later, more complete and formal interface specifications will be used. Based on such information about an installation, the assistant will build up a set of constraints to which a program running on that installation must conform. If a software specialist is tailoring a module for that installation and requests the portability walk through, he will get a checklist of constraints that his program must meet in order to run there. Items on the checklist will be displayed one at a time and will require an answer. The actual sequence of items displayed will probably depend on his previous answers.

  Such portability assistants could be very useful in the short term and could probably be implemented using very simple facts about installations and rather simple rule-based reasoning to generate sequences of constraint checks.

Mid-Term Goals

Construction of a sophisticated mid-term version of a portability facet should focus research on (and take advantage of) several basic technology areas:

- **Knowledge domains** – Facts about an installation that affect the running of a software package will have to be represented together with dependencies.
- **Fault models** – The assistant, in some versions, may use a model of previous experience reports in reasoning about portability.
- **Specification languages** – As program design languages become more powerful, the information required by the portability facet will become precise and well defined, depending only on the formal specification of an installation (operating system and hardware). Construction of portability facets should promote research on specification in a precise high-level specification language of conventional operating systems in particular, and complete installations in general.

Mid-term development of portability facets should therefore take advantage of advances in specification languages, the existence of more complete modular specifications of systems (installations), and the development of complete glossaries of keyword concepts affecting portability (together with logical interrelations between those concepts expressed in a form suitable for automated reasoning including rules, special purpose

deduction packages, etc.). The mid-term assistance would also be in the form of generated checklists. However, these would encompass a much more complete set of parameters affecting portability. Using associations with analysis of installation interface parameters, the assistant may also track histories of previous reports of software portability attempts to the installation in question. It may then issue advice during a portability walk through, e.g., who to contact about a particular interface requirement.

Long-Term Goals

Long-term development of a sophisticated reusability facet may involve a highly integrated KBSA. Reusability could be made a factor in requirements planning and refinement, module histories and documentation, and in activities coordination during system implementation. The reusability facet would track particular facts relevant to flexible use of a module and specification changes of system components.

3.4 KBSA Support System

This subsection identifies the lower-level utilities needed in the support system for the development, evolution, and eventual integration of the KBSA facets. Higher-level support utilities that require more sophistication, such as inference systems or symbolic evaluators, are discussed in Section 4, "Supporting Technology Areas."

The KBSA support system will be an integrated programming support environment that provides facilities for a number of agents to pursue a variety of simultaneous activities concerned with program development, testing, and maintenance, as well as with project management. The environment will be integrated in the sense that several policies must be adhered to by each of the many tools available in the environment; these policies are enforced through a set of system utilities. The two most important policies are those of version control and access control:

- Version control–

  The version control policy derives from a desire to minimize the amount of work each tool must do in order to account for the changes made since the last time that tool did the same job. To implement this, each program entity (for example, a procedure, a type, a data object, a fragment of documentation, a collection of program entities) will bear a version number. The version number will change only when that entity changes, thus enabling a tool like a symbolic evaluator to know what has (and what has not) changed since the last time it did an analysis of some collection of entitities. In addition, each program module that is derived by some tool will have a derivation history that relates it to the particular version of the parent modules and the particular version of the tool that contributed to its derivation.

- Access control –

  Access to all the elements of the KBSA development environment will be strictly

controlled. The user (whether involved in developing/maintaining some software product or in modifying or augmenting the KBSA development environment itself) will be constrained to deal with the environment through the activity coordinator, which will ensure that any action that is taken is appropriately authorized.

The KBSA support system has three major components: the data base, the tool set, and the user interface; we discuss these in turn.

- Data base–

  The data base maintained by the KBSA development environment consists of three functionally distinct major components: the administrative data base, the software data base, and the knowledge base. The administrative data base will be a data base that contains a variety of information to do with the agents and organizations that are known to the KBSA. Various administrative and management agents will be able to query and update this data base in order that the relationships among the organizations and personnel currently engaged in the projects under way are correctly reflected.

  The software data base will contain a set of modules–collections of program entities that, together, embody all aspects of the set of products currently being developed or maintained by some instance of a KBSA development environment. The creation and manipulation of the program modules is done by various tools, whose use is mediated by the activity coordinator.

  The knowledge base contains all the various kinds of knowledge acquired by and available to the collection of knowledge-based facets that will be integrated into the KBSA environment as well as by the activity coordinator.

  A set of data-base utilities will be provided by the KBSA to deal with the addition and deletion of various data-base elements, with backup and archiving, and with organizing and reorganizing the various contained databases to ensure a timely response to a query or update.

- Tool set–

  The set of tools available in the KBSA development environment will grow as new tools that provide assistance in various aspects of the life cycle of some software product are integrated into the KBSA. Initially, however, there will be a basic set of (standard) tools including tools for editing, compiling, and program transformation; debugging aids; tools for analysis, query, project management; for creating, dispatching, and responding to messages; for data base management and so on. It is assumed that the initial tool set will be developed by modifying various existing tools to adhere to the version and access control policies that are enforced by the KBSA. This initial tool set will later be superseded by the KBSA facets.

- User Interface–

  The user interface to the KBSA will be through the activities coordinator. It is

assumed that this interface will be realized through a work station that provides high-resolution graphic output plus a keyboard and various kinds of pointing mechanisms for input. It is further assumed that, at any time, there will be a number of windows into displays concerned with various aspects of one or more activities in which each agent is engaged.

A number of utilities will be provided to enable various tools to create and manage a variety of displays and to permit the user to control the positions, size, and other aspects of the windows currently open.

§4 SUPPORTING TECHNOLOGY AREAS

The KBSA facets must support problem-solving activities at all stages in the software life cycle. These automated facets depend on the application of different technologies, which we call supporting technology areas. These areas fall principally within software technology and artificial intelligence technology. For example, within software technology, the development of machine processable languages for formalizing programming activities and knowledge is a supporting technology area. Within artificial intelligence, the area of knowledge-based expert systems is a supporting technology area. The ultimate success of the KBSA depends very strongly on the development of the supporting technology areas.

In the past several years, the relevant supporting technology areas have been developing rapidly. There has, for example, been much research activity in the areas of requirements languages, knowledge-based expert systems, automated program verification, and sophisticated program management systems. Some of this activity has led to prototype experimental tools and in some cases to commercially applicable products.

These recent advances in the relevant supporting technology areas have created a sound foundation for the short-term goals of the proposed KBSA plan. However, further advances are required in these supporting technology areas to achieve the KBSA's mid- and long-term goals. It is expected that the relationship between the KBSA plan and the supporting technology will be symbiotic. The KBSA effort will produce growth in the supporting areas, and innovations in the supporting areas will contribute directly to KBSA. The overall effect should be a vigorous program of technology development followed by prototyping of and experimentation with advanced software support tools.

In this section, we discuss the major areas of technology required to support the KBSA.

4.1 Wide-Spectrum Languages

Work on all aspects of programming languages and high-level specification languages needs to be strongly encouraged. Language design and underlying formal semantics should be particularly emphasized. Languages concerned with distributed and parallel processing require special attention. It should be noted that our use of "language" here is intended to cover graphic and schematic representations of systems as well as conventional written representation. It is intended to cover the spectrum from requirements to implementation, from management to maintenance.

The importance of language design lies in providing 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 for their intended use in the software development process, and the existing support tools for one language have not been designed with a view to

interfacing with the tools for any other language or system. The aim of this support technology area should be development of a wide-spectrum language suitable for all stages.

At this time, research in the design of very-high-level formal specification languages should be emphasized. There is an immediate demand for specification languages that extend and complement current programming languages, for use as program design languages (PDL's) as well as for formal documentation. Later on, specification languages can be expected to provide the stepping stone for better systems design languages. By the mid-term period, new specification languages may well be developing as the new programming languages of the future. At that time, it should be possible to formulate a detailed research and development plan to produce a wide spectrum language suitable for KBSA needs.

4.1.1 Formal Semantics

Languages need a formal semantics to provide a basis for the construction of tools supporting activities such as error checking, consistency and compatibility analysis, and program transformation.

A great deal of progress has been made in the past few years in developing formal semantic models for programming languages, but much still remains to be done. For example, many of the conventional sequential high-level programming language constructs are quite reasonably modelled with denotational or Hoare-style models. Others, such as various aspects of memory management, are not yet modelled completely satisfactorily. As we consider adding certain very-high-level constructs and various notions of parallelism to our programming languages, we must extend the formal semantic models to encompass these new constructs.

4.1.2 Advanced Systems Analysis Tools

Support tools for activities carried out in a wide spectrum language must be developed. These tools will be based on the formal semantics of the wide spectrum language, and on expert systems techniques. This support area must be developed in conjunction with the language design effort. Current approaches to advanced program analysis tools can be placed in two broad categories. The first, often termed "smart compilation," seeks to gather certain facts about a program – use/set information about program variables (essentially syntactic) common subexpression information, dead/live regions of program flow for variables and so on. Much of this work is reasonably ad hoc and the mechanisms for doing it are usually embedded deep within a compiler with the information neither saved nor available to any other tool. This area of smart compilation tools needs to be expanded and the analyses they perform made available to other tools in standardized form. It can be expected to produce useful products in the short to mid term.

The second category is often termed "inference-based" and includes symbolic evaluators, transformation of specifications into run time error checks, and program verifiers. Tools in this area utilize not only the semantics of the underlying language but also user-supplied knowledge about the system itself (e.g., formal specifications, knowledge about the problem domain of the system). Development in this area of expert analysis tools needs to be vigorously encouraged and the results of these analyses made available in standardized form. This area will produce sophisticated KBSA facets in the mid to long term.

4.2 General Inferential Systems

A general inferential system is a system that supports automated inference from user-supplied inference rules applied to the modeled semantic properties of a user-defined data base. (This generally includes first-order logical operators but may also include other structural elements, e.g., operators and connectives from modal or temporal logics.) Such systems are applicable to all problem domains. This support technology area needs to be strongly supported with special emphasis placed on KBSA needs.

Important aspects in the implementation of such systems are:

1. Efficient implementation of inference rules and data representation for general logics. The inference rules are usually derived from the semantics of the language in which the data are represented. Such logics include first-order logic but also extensions and variants that may be useful in reasoning about programs such as first-order logics of partial functions, time logics, and nonmonotonic logics.
2. Structural modular facilities for expansion to include domain specific inference modules or efficient decision procedures.
3. User interface facilities specifically focused on making the system useful in practical applications. For example, an area where general inferential systems have lacked development so far is provision of facilities for explaining why a statement cannot be derived from the given data.
4. Efficient decision procedures for subclasses of logical formulas.

Much of this work will depend on theoretical advances and basic research in such areas as inference systems for various first-order logics, time logics, and decidable first-order theories. This basic research should be encouraged whenever it is relevant to KBSA facets.

4.3 Domain-Specific Inferential Systems

A major thrust of the KBSA is to provide facets that aid in the reasoning required concerning specific problem domains at various stages in the software production and

maintenance process. There is a wide variety of specific problem domains for which the development of automated reasoning support is essential to KBSA facets. These domains include the domain of programming languages, application-specific domains, and the domain of project organization and the coordination of the activities ongoing within a software project.

For each individual problem domain, special rules of reasoning and solution finding will apply. Inferential systems based on the special rules are much more efficient than the application of general inferential systems to an encoding of special domain within, say, first-order logic. Specialized inferential and problem-solving systems will be essential components of individual KBSA facets. Therefore, research in this area focused on special problem domains related to proposed KBSA facets should be strongly supported. This research can be divided into three areas outlined in the subsections below.

4.3.1 Formal Semantic Models

A formal semantic model for a problem domain is a system of definitions and rules that permit a human being to reason about the objects in that domain, and their interrelations. Such a model is a most desirable, and perhaps necessary, precursor to any techniques for mechanical reasoning and problem solving in that domain.

4.3.2 Knowledge Representation and Management

The knowledge concerning each domain must, at least conceptually, be available in a knowledge base that is used by the various tools reasoning about that domain. This knowledge is represented in a fashion appropriate for external use and is also represented internally in such a way that it can be accessed, updated, and efficiently maintained. Several external representations will often be desired. For example, the form in which an expert in the domain presents knowledge to the knowledge base may differ drastically from the form in which we wish the system to represent this information to someone who is not a domain expert – a user, programmer, or manager, for example. For the nonexpert, we typically wish to explain, in lay terms, some aspect of the knowledge about certain objects or situations.

Data (knowledge) about a problem domain may be of various forms. Some data may be applicable to the knowledge base; these are generally called (inference) rules since their function is to deduce (new) facts about the domain from the existing data. Other data may take the form of heuristics for deciding when rules can be usefully applied.

Knowledge management concerns analysis of the knowledge data base. The knowledge base for a problem domain may change as a function of the activity in that domain. For example, a project progresses from a prototyping stage, during which the coordination

may be mostly informal, to a production stage and then a maintenance stage during which the coordination may be highly formalized.

Support technology for knowledge management in KBSA must address the problems of change and explanation. For example, in KBSA, the following examples of knowledge management aids will be required:

- There must be mechanisms to "explain" the rules (appropriate to some situation) to, for example, managers who are not particularly adept at dealing directly with first-order logic or some variant thereof.
- It must be easy to add, remove, and modify rules. The use of relations in a relational data base to represent certain (ground) rules and a reasonable management system for the relational data base would be most helpful here.
- There should be mechanisms for checking the consistency of the data, noting redundant facts, and so on.

4.3.3 Specialized Inference Systems

Efficient inference systems for a given problem domain will need to be developed based on (and in conjunction with) the semantic model and data representation for that domain. Such systems may be close enough to standard general inference systems (resolution, equational rules, implicational rules, PROLOG schemes, etc.) to be implementable using general techniques. However we must also encourage research into specialized inference systems in order to explore all possibilities for efficiency. (This situation is somewhat analogous to a current situation where, in the context of current Von Neumann machines and programming languages built on top of them, research in alternative data flow machines is being pursued.) Efficient inference systems for a given problem domain are likely to be realized by building much of the inference mechanisms into the data representation.

4.4 Integration Technology

A fundamental premise in the KBSA plan is that various facets can be integrated into uniform (from the user's point of view) environments in the mid to long term. The ability to achieve this goal will depend on development of a supporting technology for integrating separate facets. Integration technology covers both the underlying KBSA support system itself (Section 3.4) and the integration of separately developed facets.

4.4.1 KBSA Support System Technology

Adequate technology for implementing the basic facilities required of the KBSA support system itself (Section 3.4) must be developed. This may be categorized as basic management facilities. While technology in this area is already well developed so that implementation of initial KBSA support systems can be undertaken now, experience will almost certainly demonstrate weaknesses and areas where further research and development is required.

4.4.2 Interfaces and Standards

As the KBSA progresses, our experience with initial integration experiments should be used to develop guidelines and standards for interfaces between KBSA facets. Experience has already shown that such standards are difficult undertakings . Research must be encouraged in investigating such support areas as

1. Definition of standard abstract data structure representations for internal forms of broad spectrum languages (Section 4.1).
2. Standard interface facilities to be supplied by knowledge data bases.
3. A universal user command language for all KBSA facets.

§5 PROJECT PLAN

The KBSA effort involves a fundamental shift from the current informal person-based software paradigm to a formalized computer-assisted paradigm and the creation of the knowledge-based software assistant required to support it.

This ambitious undertaking cannot be achieved by a single effort in one giant step. Rather, it must be approached through a series of small steps over an extended period. This recognition has been the major influence on our plan. This plan consists of a set of coordinated parallel efforts which will be periodically integrated into a succession of usable, and increasingly comprehensive, KBSAs.

The primary strengths of this plan lie in the identification and enunciation of the KBSA paradigm as the solution to the problems besetting current software efforts, and in the identification of a technical framework for fostering the distributed, gradual creation of KBSAs.

The heart of this technical framework is the combination of the activities coordinator and the knowledge-base manager which enables each life-cycle activity to be incrementally formalized and fitted into the matrix of other activities. It forces consistency among these activities and imposes standards. Most importantly, by formalizing the activities and by managing the knowledge involved, it provides the basis for knowledge-based support of both the individual activities and the software development as a whole.

Thus, two major tasks in the plan are the creation of suitable KBSA frameworks and the incremental formalization and knowledge-based support of the individual facets of the software life cycle. The individual facets identified in Section 3, and their development, should be taken as illustrative and suggestive of the possibilities afforded by the KBSA paradigm and must not limit further insights.

A third major task is the periodic integration of the evolving individual KBSA facets into a succession of more comprehensive KBSAs. These integration efforts will be used as demonstrable measures of progress toward a complete KBSA. They will also serve as the basis for the production engineering and documentation needed to transfer the KBSA technology and approach to actually impact the software development process. This technology transfer is the fourth major task.

The final major task is the supporting technologies identified in Section 4, such as knowledge-base management and inference techniques, which, while not specific to either the framework or any individual KBSA facet, are crucial to the overall development of KBSA capabilities.

Thus, planning of the KBSA project has been divided into five major tasks. The interactions and dependencies between them over the 15-year time scale are described below. Achievement of these tasks is discussed in terms of milestones (short-term = 3-5 years, mid-term = 7-10 years, long-term = 10-15 years). Specific recommendations aimed at achieving the planned developmental steps are also given.

A steering committee will oversee the implementation of the plan and advise the funding agency on progress. The committee is to undertake a number of specific tasks in addition to general planning. These include the definition of conventions, establishment of guidelines and techniques for integration and technology transfer, and the selection of facet capabilities to include in the integration efforts.

5.1 Outline

The overall plan is to develop individual KBSA facets simultaneously by supporting parallel development efforts. The individual facets are to be integrated into prototype KBSA systems.

Each individual facet development effort will be planned to produce intermediate products, which may have immediate practical application. The intermediate products represent developmental stages toward the final goal, and should be planned for the short- and mid-term periods. Construction of prototype KBSA systems by integration of facets or intermediate products is also planned as separate efforts in parallel with further facet development.

Technology transfer efforts will begin near the end of the short term and continue through the end of the project. Support technology efforts will begin immediately and continue through the end of the mid-term.

The plan contains three stages of milestones:

- The short-term milestones (3-5 years) are: first, the development and demonstration of individual facets as specified in detail in Section 3.3 and collated in Section 5.4; second, the demonstration of a framework for supporting the KBSA, including a working activity coordinator and at least one facet, and integrated systems consisting of some of the individual facets or intermediate products; and third, a set of guidelines and standards to facilitate integration of facets as defined by successful frameworks. Some technology transfer is also expected in this stage, and a preliminary set of guidelines for technology transfer will be defined.
- The mid-term stage (5-10 years) requires facets to conform to the integration guidelines. Milestones of this stage are: first, further development of more advanced stages of facets as specified in Section 5.4; second, demonstration of integrated systems consisting of many facets; and third, further successful technology transfer efforts.
- The long-term milestone is the construction of prototype KBSA systems that integrate all facets.

It is clear that a planned effort of this nature requires careful monitoring by the steering committee, particularly with regard to definition of individual facet efforts, production

of guidelines and standards for integration and technology transfer, and selection of facets for technology transfer.

5.2 Tasks

The KBSA plan is structured into the following five tasks:

1. Definition and implementation of KBSA framework (i.e., an activities coordinator and knowledge-base manager).
2. Definition and implementation of individual prototype facets with specific capabilities. (Sample facets are described in Section 3.)
3. Integration (i.e., development of integrated prototype KBSA systems that include several coordinated facets).
4. Technology transfer of facets (short term through mid term) and of integrated systems. (long term)
5. Development of the technology support base (described in Section 4).

These tasks and their interactions over the 15 year time scale are illustrated in Figure 1. Note especially the close "organic" relationship between the framework and facet tasks. The tasks are described below:

1. Definition and Implementation of KBSA Framework

The KBSA framework provides the basis for the development and integration of facets. It consists of an activities coordinator, a knowledge-base manager, a wide-spectrum language for representing multiple levels of software development knowledge within a facet, and a set of support utilities (user interface, inference engine, etc.).

The magnitude of the conceptual and system-building efforts required to construct such a KBSA framework necessitated our identifying it as one of the major KBSA tasks. However, we cannot conceive of it being undertaken except in conjunction with concurrent facet development.

Framework precursors will necessarily arise (in piecemeal, ad-hoc form) from early facet efforts. Some of these ad-hoc capabilities may be enhanced into comprehensive and well-founded KBSA frameworks. But this can only be done in the context of existing and planned facets. As these frameworks mature, other development efforts wishing to focus on individual facets can do so by adopting one of the emerging frameworks on which to build.

2. Definition and Implementation of Specific Facets and Capabilities

Section 3 gives a sample list of proposed KBSA facets, i.e., assistants for requirement, specification, development, testing, performance, reusability, and project management. The actual set of facets will probably include some not explicitly mentioned here.

Each individual facet effort will be required to contain a development plan that is structured into short-, mid-, and long-term stages. This plan will include both short-term and longer-term milestones leading to development of a sophisticated knowledge-based facet.

Although the development plan will differ somewhat from facet to facet, we expect it generally to follow the staged knowledge-based development model outlined in Section 5.3. Thus, a typical facet effort will progress from the formalization of properties relevant to the activities of the facet, to inferences that can be drawn on such properties, to actions that can be explicitly invoked to modify and/or maintain those properties; to the formalization of goals which, via planning and problem-solving, can implicitly invoke such actions; and finally, for some facets, to the inclusion of knowledge acquisition facilities that will further enhance the capabilities of the facet. During this extended staged development of knowledge-based facet capabilities, periodic (every 3 years) demonstrations of working prototypes of the individual facet will be produced.

Some of these parallel individual facet efforts will be selected for inclusion in an integrated KBSA, as described in Task 3 below. Such integration will be in addition to the continued development of the facet and not a replacement for it. As the framework activity matures, a set of standards and conventions will emerge that will guide further facet development.

3. Integration

This task requires that the different facets and capabilities developed and implemented in the course of the planned effort be integrated into existing KBSA frameworks as demonstrable working laboratory prototypes. The first integration prototypes should be planned to start toward the end of the short-term period (5 years). This task is preliminary to the building of production versions (described in Task 4 below).

A secondary goal of the integration effort will be the definition of guidelines and techniques for integration of facets. These guidelines and techniques will be available by the end of the short-term stage and will be a conformance standard for both facet and framework efforts during the mid-term stage.

4. Technology Transfer

Technology transfer of a facet or integrated KBSA system means the production engineering to make it available and usable by a broad segment of the software community.

Technology-transfer planning should be regarded as an extension of the integration efforts. It entails coordination with existing automated environments and human engineering for practical use. In order to plan technology-transfer activities, the steering committee will need to track and evaluate other software technology efforts, coordinate elements of the KBSA effort with outside efforts (when this is judged to be possible and timely), and either extend the integration guidelines or produce separate specific conventions for technology transfer.

Technology-transfer guidelines and conventions should be available during the mid-term phase. These should be compatible with the integration guidelines, and should be based on experience with three to five technology transfer efforts. Accordingly, some KBSA facets must be planned so that their short-term goals merit technology transfer.

Some technology transfer will occur after the short-term milestone, and a major transfer effort will begin after the mid-term milestone. It is recommended that conformance with the technology transfer guidelines be encouraged in the mid-term planning.

5. Technology Support Base

This is the technology that is needed to support the KBSA effort. We have discussed in Section 4 the areas of technology support needed. Such research should be supported on the basis of its relevance to the KBSA objectives.

5.3 Staged Development of KBSA Facets

A key feature of the KBSA project plan is the staged development of individual KBSA facets. As an aid to understanding, for those unfamiliar with knowledge-based systems, we present here one particular model for describing such incremental staged development of knowledge-based capabilities in the individual facets. We will also use this model to establish a framework to describe the relative maturity of each facet in the milestones presented in the following subsection. Naturally, actual knowledge-based systems, both existing and future, display great diversity in their staged development and may only follow this pedagogical model at the broadest level. Readers familiar with knowledge-based systems and their development can proceed directly to Section 5.4.

This section describes a possible sequence of stages through which each facet might logically progress. The similarity in the order of development for each facet is due to their common functional architecture (shown in Figure 3) and the logical dependencies between the elements of that architecture. Note, however, that the relative timing of similar stages will vary greatly between facets according to the difficulty of the development issues involved in each area.

Figure 3 shows all the functional elements that must be present to support the behavior of a mature KBSA facet like those described in Sections 3.3. First we will discuss the relationships and dependencies between these functional elements in the mature facet. We will then describe the typical order in which these elements are incrementally developed.

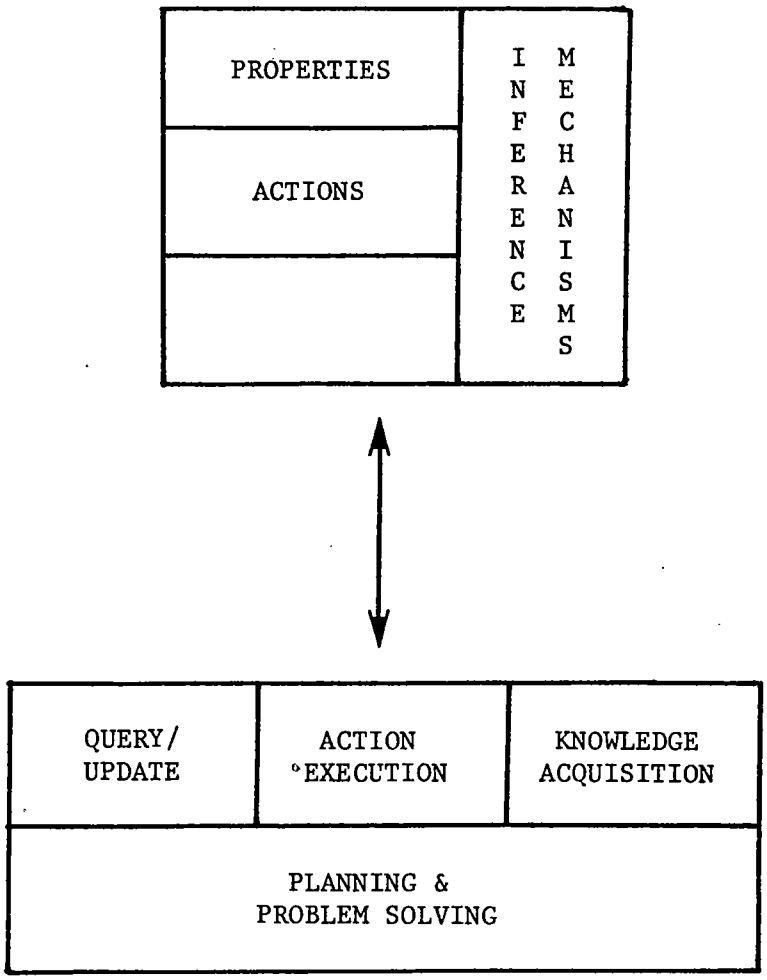

Figure 3. FUNCTIONAL ELEMENTS OF A MATURE FACET

The first-level functional decomposition in Figure 3 is between the active parts of the facet (bottom) and the knowledge base (top). Note that this division is not exact because in knowledge-based AI systems, a certain amount of active processing, called the inference mechanisms, is typically associated with the knowledge base itself.

The specific information in the knowledge base varies according to the domain of each individual facet. It is possible, however, to distinguish three general types of knowledge which are relevant in all domains: properties, actions, and reasons.

The most basic type of knowledge with which a facet is concerned is the properties of objects (e.g., requirements, specifications, code, test cases) in its domain. A property is typically a simple fact about the relationship between objects which may or may not be true at a given point in time. For example, a property of test cases might be the version number of the module on which it has most recently been executed.

A higher-level vocabulary within the knowledge base describes the actions of a particular domain. Actions are typically defined by a set of inputs (objects upon whose properties the action depends), a set of outputs (objects that are created by the action or whose properties are modified by the action), preconditions (properties that are expected to hold between inputs prior to execution of the action), and postconditions (properties that will hold between outputs and inputs due to the execution of the action). Program transformations are examples of actions in the implementation domain. Note that the action vocabulary builds on the property vocabulary because the preconditions and postconditions of the actions are expressed in terms of the properties.

In order to provide more advanced services to the user, a facet also needs to "understand" the reasons behind actions. The same action can be performed for different reasons. For example, the reason for changing an implementation decision might be either because of a change in specifications, or in order to increase efficiency.

In the active part of the facet, there is a related generic layering of functionality. The most fundamental service the facet can provide is to function as a data retrieval system, i.e., to provide query and update operations on the knowledge base. For example, it is useful simply to record the history of modifications to a module.

At a more advanced level, a facet has the ability to execute specific actions under direct command from the user. For example, the implementation facet will apply a program transformation chosen by the user.

With a fully mature facet, the user specifies his goals, and the facet, using planning and problem solving techniques, chooses and executes appropriate actions to achieve these goals. The user's goals then become the reasons that justify the actions taken by the facet. For example, the user will tell the testing facet that he wants a given module to be tested for release. It is then up to the facet to figure out what test cases to run.

A fully mature facet may also independently acquire new knowledge as a by-product of its other activities. For example, an implementation facet may discover new optimizations.

Figures 4 through 7 show the typical order of intermediate stages through which each facet will develop from first prototype to maturity. In each figure the newly added functionality at each stage is indicated by asterisks. As mentioned above, the relative timing of these stages will differ greatly between facets, but the order is essentially the same for each.

The first stage of development is shown in Figure 4. Work on each facet will begin by identifying the types of objects in the domain and the useful vocabulary of properties of those objects. A formal language and appropriate utilities are then provided for specifying and querying those properties.

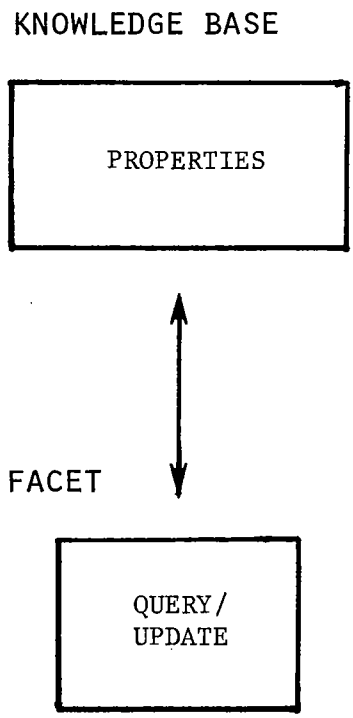

Figure 4. FIRST STAGE OF DEVELOPMENT: THE PROPERTY STAGE

The next major stage of development, shown in Figure 5, is typically to enhance the query and update capabilities of the facet by adding some inference mechanisms to the knowledge base. Initially, these inference mechanism deal only with properties – for example, inferring implicit properties from explicit ones, or detecting contradictions between stated properties. However, as the knowledge base is enhanced by adding actions and reasons, the inference mechanisms are also typically upgraded to deal with the new types of knowledge.

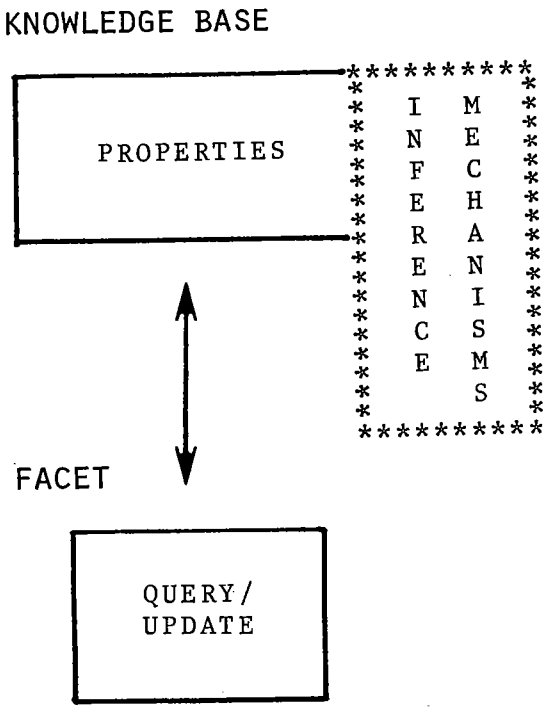

Figure 5. SECOND STAGE OF DEVELOPMENT: THE INFERENCE STAGE

The third stage of development, shown in Figure 6, is to introduce a representation for the actions in the facet's domain and the ability to execute these actions. The knowledge base at this stage may contain not only an enumeration of possible actions, but also the history of actions that were actually taken. Note that, at this stage, the facet takes no initiative; actions are performed only on direct command from the user. However, the facet can perform many useful monitoring and recording functions at this stage, such as verifying that the preconditions of an action hold before it is executed, or allowing the user to edit and replay a sequence of actions already performed.

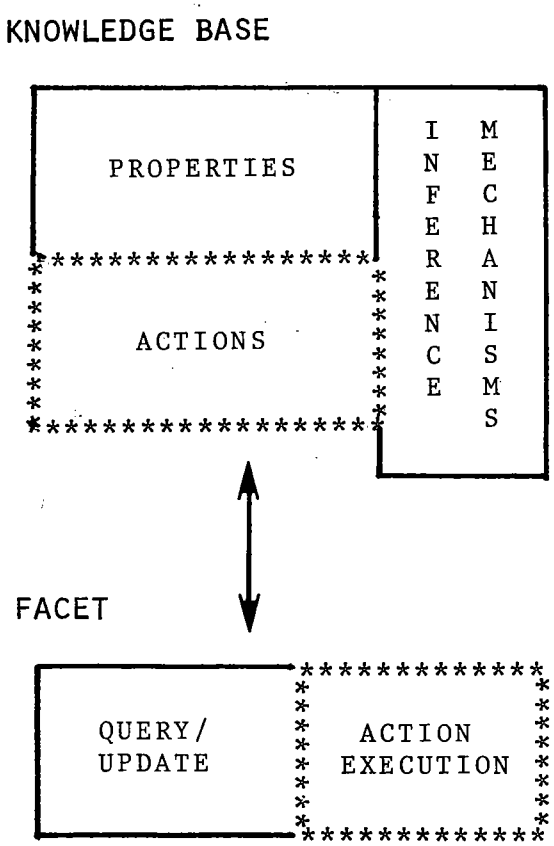

Figure 6. THIRD STAGE OF DEVELOPMENT: THE ACTION STAGE

The essence of the transition to the next stage of development, shown in Figure 7, is that the facet begins to assume some initiative. This stage is fundamentally based on developing a vocabulary of reasons (goals) in the domain. Given this vocabulary, the facet can use problem-solving techniques to plan sequences of actions to achieve goals that the user specifies.

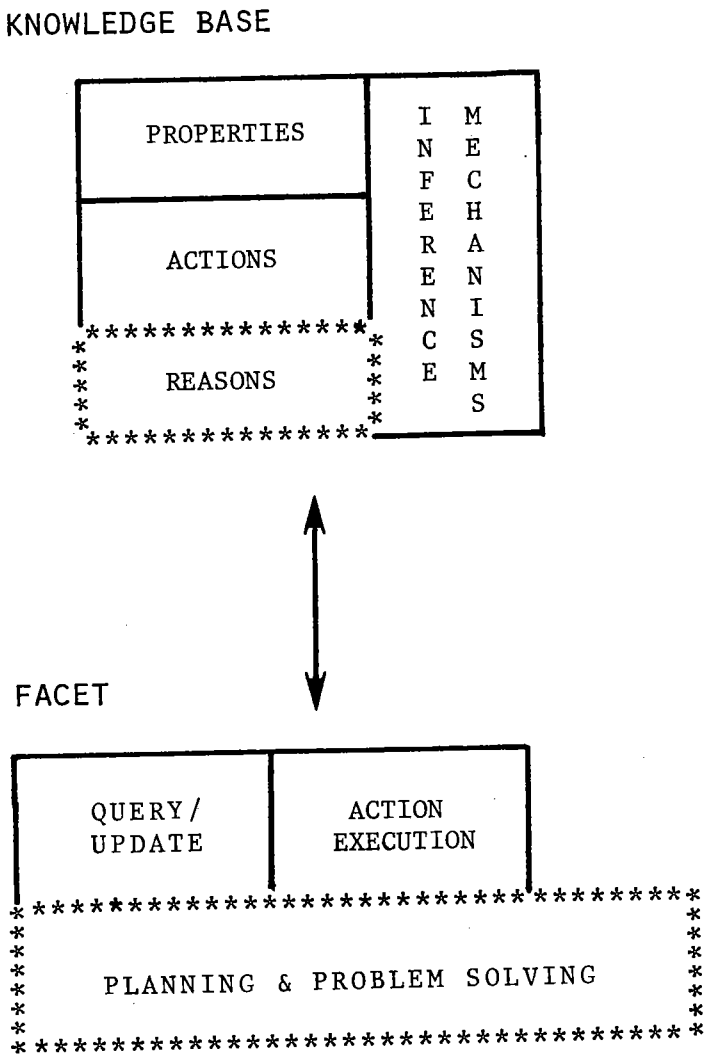

Figure 7. FOURTH STAGE OF DEVELOPMENT: THE PLANNING STAGE

Finally, some mature facets (Figure 3) may include knowledge acquisition capabilities. By this we mean not only the ability for the user to add new information, which is provided by the query/update facilties from the beginning, but also the ability of the facet to independently develop new vocabulary for itself to use to improve its own performance.

5.4 A Note on Limitations

We close this section by discussing our position on including, as goals for the KBSA, fully automatic program synthesis and comprehension of natural language specifications.

An important issue is that of what degree of automation in program synthesis is the goal for KBSA. The goal we have chosen for KBSA is machine-assisted program synthesis, rather than fully automated program synthesis.

We feel this will allow the KBSA to work with the highest level of languages, since at any point in time there may exist a gap between the level of language that is fully automatically compilable and the level of language that can be implemented by experts with machine assistance. Even though the level of language features that can be fully automatically compiled will steadily rise in time, our strategy will allow the level of specification and requirements languages to rise above that level.

By thus including the highest level constructs in the KBSA's formal specifications and requirements language, many forms of knowledge-based assistance can be applied at the highest level, even though the implementation of these languages will require human interaction.

The KBSA will be structured so that as advances occur in the area of fully automatic synthesis, they can be readily incorporated. Since we expect to keep the developer as well as the machine in the loop, provision of suitable interfaces will be made so that the developers and the KBSA can work effectively together.

Natural language specification was omitted as a goal because it is orthogonal to the KBSA approach. The KBSA approach is based on providing better and better formalisms for developers to use (and to assist them when they employ these formalisms), while natural language specification attempts to reexpress informal description in some formalism. Hence, it presupposes the existence of those formalisms. Thus, it cannot replace any of the proposed effort in defining those formalisms and providing assistance when they are used, but rather is a "user interface" to those formalisms.

While this would certainly be a useful addition to the KBSA, it seems to represent an unneeded dilution of focus and energy within our objectives of providing automated assistance to the software development cycle. Rather, it should be pursued separately as a research objective in its own right.

5.5 KBSA Milestones

Figures 8, 9, and 10 show the stage of development each facet of the KBSA is expected to reach in the short-, mid- and long-term time frames. In addition to characterizing each facet in terms of the generic stages introduced in Section 5.3, we also repeat below the major milestones for each facet defined (and more fully described) in Section 3.

Note that as the capabilities of any one facet of the KBSA grow, it naturally begins to overlap with the others. In each milestone, we point out some examples of integration between facets which become feasible in that time frame. In general, we expect an evolution of the overall structure of the KBSA from essentially separate facets in the short term, to a set of integrated, cooperating tools in the mid term, to in the long term, an assistant which is better viewed by the KBSA user as the single active agent described in Section 2.2.3 rather than as separate facets.

SHORT-TERM MILESTONE (Figure 8)

Project Management Facet

- Project management formalism
- Knowledge-base manager and message handler
- Task tracking

  (Initially, it will be useful just to have a record of the status of all tasks on-line with convenient update and query facilities.)

Requirements Facet

- Analysis of requirements planning method
- A formal requirements language
- Smart editing and managing of requirements
- Reviewing the requirements definition for the user

  (Similarly, these milestones for the requirements facet correspond roughly to the properties stage of development.)

Specification Validation Facet

- Executable specification language
- Specification wellformedness checking
- Specification testing
- Specification paraphraser

  (Checking the wellformedness of specifications implies the existence of some inference capabilities. Executing specifications is a kind of action execution.)

Development Facet

- Wide-spectrum language
- Transformation language
- Property language
- Interactive mechanical development
- Automated property proving

(Similarly, these milestones for the development facet imply the existence of both inference and action capabilities.)

Performance Facet

- Data structure analysis and advice
- Subroutine and module decomposition advice

(In the short term, the performance facet will need a representation for performance properties and the ability to reason about them, but will not be able to take any actions itself.)

Testing Facet

- Test-case maintenance assistant

(Test case maintenance, including automatically running test cases, will be feasible in the short term.)

Project Documentation Facet

- On-line documentation

(Similarly, in the short term all documentation should be at least on-line, even if there isn't a very deep understanding of it by the KBSA.)

Two examples of opportunities for integration between facets in the short term are shown in Figure 8. First, test cases should be treated by the project documentation facet as a form of documentation of the system being developed, and thus should be accessible through the same sort of interface. Similarly, decisions made by the development facet should also become part of the project documentation.

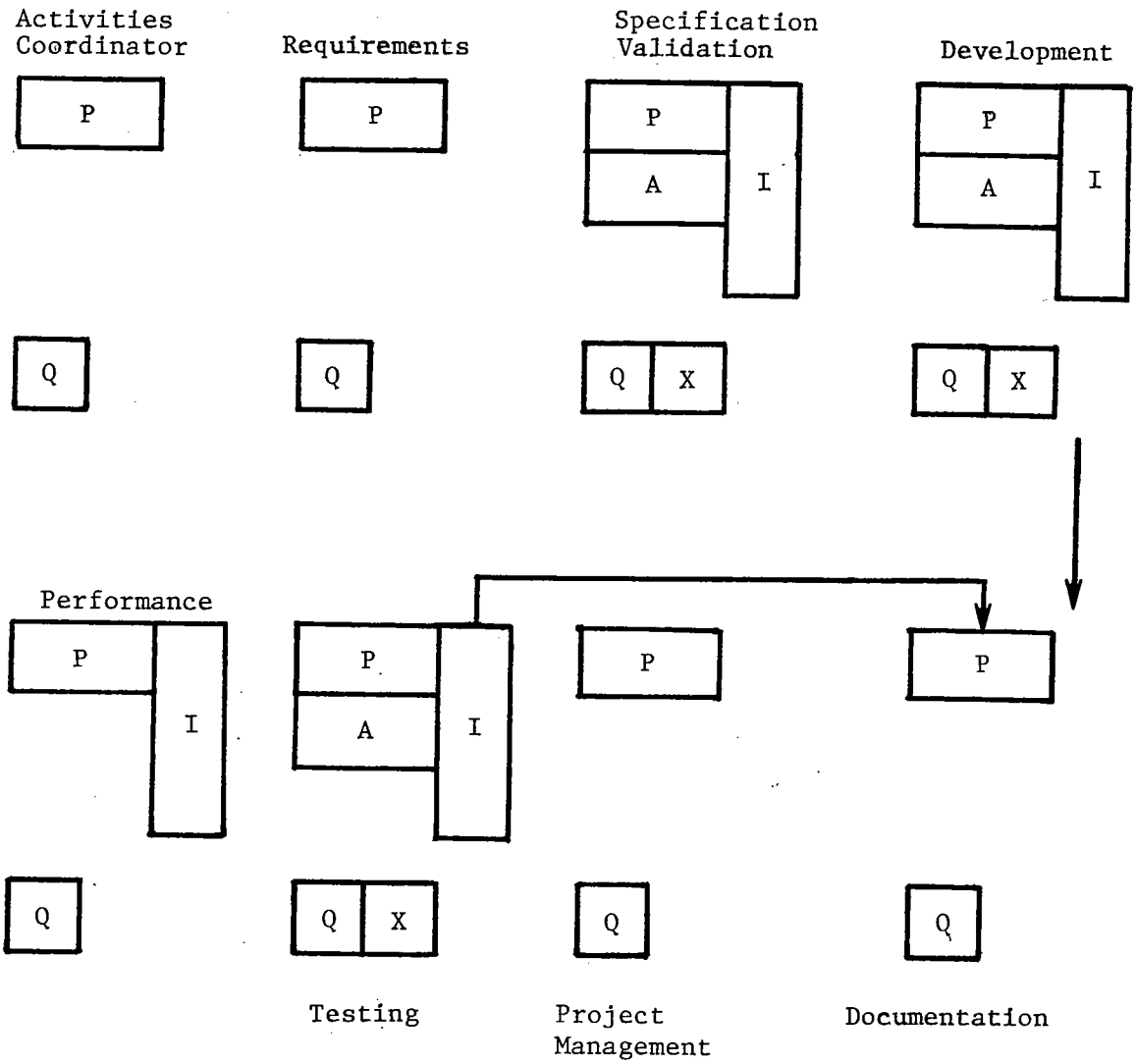

Figure 8. SHORT-TERM MILESTONE

MID-TERM MILESTONE (Figure 9)

Project Management Facet

- Suggesting simple management decisions
- Plan and procedure creation and modification
- Knowledge acquisition

(These milestones require the addition of inference and action capabilities.)

Requirements Facet

- Requirements transformation and refinement

(Similarly, this milestone requires the addition of inference and action capabilities).

Specification Validation Facet

- Rapid prototyping
- Behavior explanation

(Explanations of behavior require representation of the reasons behind elements of the specifications.)

Development Facet

- Automated development
- Automated replay

(Automated development and replay require planning and problem solving capabilities.)

Performance Facet

- Domain models for analysis
- Algorithm design analysis and advice
- Real-time performance advice

(These more advanced types of performance analysis will require fully developed problem-solving capabilities.)

Testing Facet

- Knowledge-based test generation

(At this point the testing facet will be fully mature, including the ability to independently define test sequences for a particular program using its general knowledge about the application domain. Planning abilities are the key new feature at this stage of development.)

Project Documentation Facet

- Partially formalized documentation

  (The formalization of documentation will allow a more powerful set of operations to be performed on it.)

A number of examples of integration between facets in the mid term are shown in Figure 9. The requirements facet should function as a source of test cases. The performance facet should be able to invoke the testing facet to run test cases for the purpose of measuring system performance (perhaps under different implementation decisions). The development facet should inform the testing facet of those parts of the program that do not need to be tested because they have been derived by correctness-preserving transformations. Finally, the specification validation facet may call upon the development facet for a rough implementation of specifications to be run to obtain user feedback.

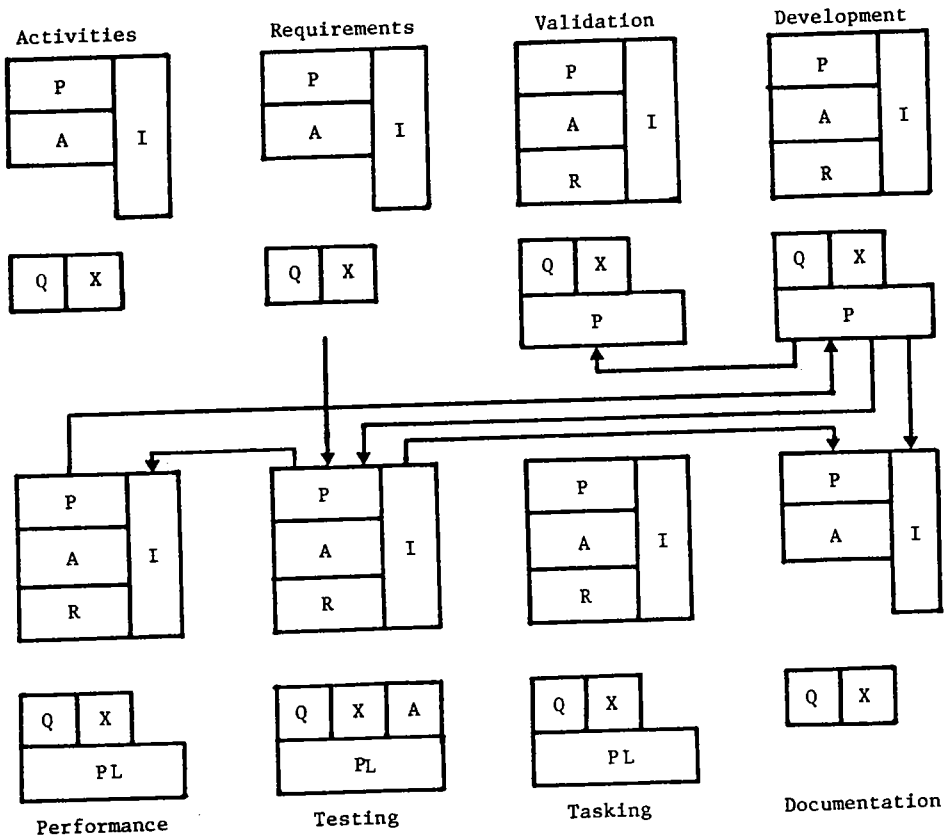

Figure 9. MID-TERM MILESTONE

LONG-TERM MILESTONE (Figure 10)

In the long term, all of the facets will reach their mature form, with the exception of knowledge acquisition capabilities in some areas, which are expected to remain very difficult artificial intelligence problems. The mature capabilities of the various facets are described in detail in Section 3. A new example of integration shown in Figure 10 is the invocation of the development facet by the requirements facet as an early filter on the technical feasibility of the user's requirements as they are being defined.

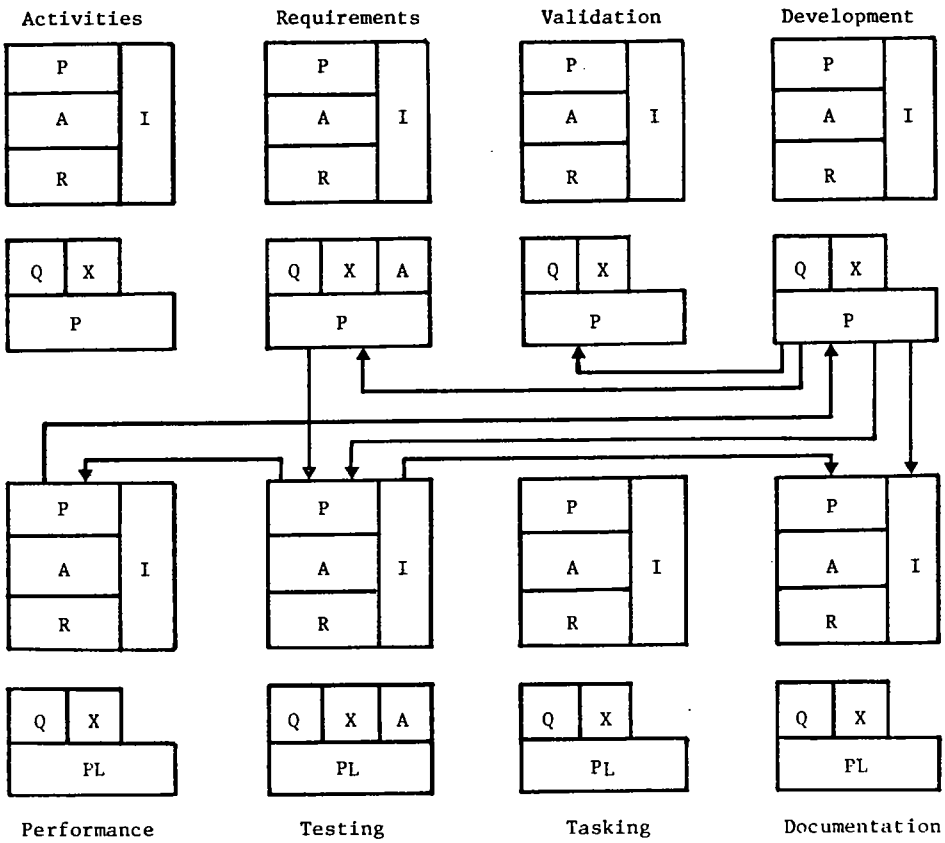

Figure 10. LONG-TERM MILESTONE

ACKNOWLEDGMENTS

Guidance and impetus for this study came from Northrup Fowler, Donald Roberts, Douglas White, Samuel DiNitto, and William Price. Larry Druffel, William Riddle, and Winston Royce all provided technical input to the study. Judy Tollner was administrator of this project. Assistance in formulating the facet descriptions was provided by Beverly Kedzierski and Elaine Kant. Carl Engelman helped formulate Section 1.

§6 REFERENCES

1. Software Technology for Adaptable Reliable Systems (STARS) Program Strategy: DoD Report, *ACM-SIGSOFT Engineering Notes, Vol.*8, No.2, April 1983, pp. 56-84.

2. Barr, Avron and Feigenbaum, Edward A., *The Handbook of Artificial Intelligence, Volume 2,* Chapter X, William Kaufmann, Inc., Los Altos, 1982, pp. 295-379.

3. Hünke, Horst (Ed.), *Software Engineering Environments,* North-Holland, New York, 1981.

4. Special Issue on Programming Environments, *IEEE Computer, Vol.*14, April 1981, pp. 7-45.