Towards a Manageable Solution to the Iterative Development of Embedded Knowledge-Based Systems

Thomas P. Gathmann  Donna Halker

Rockwell International Corporation
Satellite & Space Electronics Division
Seal Beach, CA

ABSTRACT
The development of knowledge-based systems (KBSs) is a more complex variant of the traditional software development lifecycle. This paper specifically addresses the project management, knowledge acquisition, and testing traceability of a KBS. Conventional software engineering mechanisms are utilized and woven into a process that encompasses the entire development lifecycle. The knowledge acquisition phase is crucial to a successful project, and as such, has been the focus of our efforts.

INTRODUCTION
Problem Statement
We have addressed the problem of tracking system requirements to support knowledge based system (KBS) software design, development, testing, and project management. Many issues that are associated with conventional software development apply to KBS projects as well. Both types of projects involve the translation of complex system functions into precise, measurable requirements. Requirements must be testable and traceable to the software design and implementation. Well-defined requirements establish the criteria for determining if the project is on-schedule, and when it is complete. Management methods and tools for managing conventional systems differ from KBS due primarily to differences in the development approach.

Traditional software systems are frequently developed using the waterfall approach, especially for embedded systems. This methodology insists first on fully defined requirements, then the system is designed to meet the requirements, and finally, the implemented software is tested against the established requirements. Design and implementation solutions can be stated in specific algorithms and step-by-step procedures that guarantee the right conclusion will be reached when the correct data is entered. During the conventional software development life cycle, requirements are further clarified and a few may be added or deleted, but the original requirement set is not significantly changed.

The problems associated with managing KBSs are compounded by several factors. The iterative nature of KBSs implies that detailed requirements evolve during the development life cycle, consequently, standard procedures for development of conventional software are difficult to apply to KBSs. Today, very few automated tools exist for life-cycle management of conventional software, much less KBSs. Our approach for reducing the problems of KBS management incorporates the use of available tools and manual techniques. The main issues that we address in this paper include the mechanisms for managing the project's schedule and budget, and a concise method for requirements definition.

Background
The KBS development cycle for the eventual generation of flight-worthy code can be viewed as a subset of the traditional software cycle. Figure 1 shows the relationships between the conventional software development cycle and the one that we have successfully employed on many projects. The objective of this process is the development of a consistent requirements specification along with an associated top-level design. Knowledge acquisition is similar to convention requirements phase and also establishes portions of
top-level design. The prototyping activities further define the design and clarify requirements. Prototype testing verifies requirement and design compatibility. Conclusion of the prototype testing phase occurs when a high degree of confidence in requirements definition and top-level design exists. From this point on, the traditional methodology may be employed for detailed design, implementation, and testing of the code.

The Approach section of this paper provides a brief introduction of each tool and method we employed, and explains the overall process. In the Tool Applications section, we answer the same questions for each tool or method: what are the capabilities, when the tool is utilized, and how did it help resolve the problems associated with knowledge acquisition, KBS testing, and project management.

A suite of generally available tools, along with the associated methodology, can be defined to define a process in which the management of KBS is consistent with accepted conventional software techniques. Such a methodology has been developed over the course of several projects and is continuing to be used and refined. However, due to the immaturity in conventional software environments regarding project management, this methodology also lacks complete procedures for the development process and metrics reporting. Automated project management tools are needed to reduce costly and error-prone 'man-in-the-loop' bookkeeping tasks.

### APPROACH

The problem, summarized above, is the conventional software engineering project management task does not have a repeatable solution. The case was also made that development is typically a much more stressing environment for the manager as well as the engineer. Developing a flight-worthy KBS is even a more stringent requirement. The methodology presented in this paper adequately handles these factors. The main problems present on our KBS projects included:

- requirements were needed to bound the problem so we could scope the task in terms of budget and schedule
- a vehicle was needed to capture requirements that represent design logic and a method of representing the requirements that is easy to create, maintain, and be reviewed by experts
- a system was required to track the relationship of the expert system rules to the decision logic and, ultimately, to the system requirements
- a method was needed to maintain...
requirements as they evolved to contain further detail.

We have addressed these problems over a variety of projects and have identified various options which are available for application in the KBS environment. The chosen tools each provide support in a particular portion of the KBS development cycle (see Figure 2). The utilization of the four distinct tools provides full coverage of the entire cycle and allows derivation of many project management metrics as well.

The software engineering domain has floundered in the development of custom software [1]. The need to improve the productivity of developers and implementors (i.e., software engineers) has become the focus of many programs and organizations. Implicit in the problem of software engineering is lack of consistent and accepted nomenclature and procedures. The identification of the phases in the software engineering process has been agreed upon for many years, however, their sequence and relationships have been in a state of flux [2]. Essential to phase transition, regardless of the sequence, is the utilization of a common device which supports each of the phases. This is the basic motivation behind current Computer-Aided Software Engineering (CASE) tools. Their intent is correct, however, they typically enforce a particular development methodology. No known commercial CASE tools acceptably embrace the KBS development lifecycle.

A tool to assist in the acquisition of knowledge and information from experts and other sources was required. This need was emphasized when combined with demanding schedule and budget constraints. A tool was necessary that would help automate portions of the knowledge acquisition phase as well as ease the transition from that phase to the design, implementation, and testing phases of the overall development lifecycle.

The lack of available tools to address the KBS domain has necessitated the use of narrowly-focused tools each supporting a portion of the overall lifecycle. The transition from requirements to design is typically accomplished through the allocation of requirements to high-level definitions (i.e., modules). We have utilized a relational database management tool to document the requirements and their allocation to modules within our system. The use of this tool provides a flexible medium for the documenting of the requirements and their relationship to the end-product (code). The database may be tailored to include information that is pertinent to a particular phase of the lifecycle (e.g., module identifier associated with requirements to support the transition from requirement definition to system design).

Implementation segments (e.g., rules and data structures) may contain pointers to design information. We have capitalized upon this opportunity and combined it with the desire for visibility into the implementation phase. The utilization of standardized headers to be included with each rule and data structure was enforced. These headers contain information pertaining to the requirements and
design information handled by gIBIS (graphical Issue Based Information System) in addition to more traditional software engineering tracking data. These include the name of the entity, item authorship information, and item revision history. The utilization of these headers provides a framework for the transition and traceability between the requirements, design, and implementation phases of the development lifecycle for KBSs. The use of such transitional devices enforces the need for well-structured knowledge bases. Structured knowledge bases provide the software engineer a more modular and familiar interface for the implementation phase [3].

The need for a common repository of project-related information is satisfied by the use of a traditional software engineering mechanism, the Software Development File (SDF). This entity contains the requirements, design, source code, test cases and results, and metrics data for use in project management. The SDF is the mechanism for determining current status, as well as, reviewing historical data. The SDF represents both a snapshot of the project at a given moment in time and a history of the incremental effort associated with producing the finished product.

Closely tied to the SDF are the software metrics data. These parameters quantify measures of project complexity, scope, completeness, and volatility, to name a few. These parameters are used in conjunction with historical data and management techniques to assess project 'health'. Recommendations may also stem from the metrics data when taken in a historical perspective. Metrics tie together engineering performance, schedule, and budget into a cohesive picture (if interpreted correctly!).

Figure 3 provides a pictorial view of our approach in terms of the tools that we employ in KBS development projects. The system definition is focused through the use of gIBIS. gIBIS provides a cohesive thread throughout the entire development cycle. The requirements specification is a direct descendant of gIBIS products. The relational database tool ties together the requirements, design, implementation, and test status of the project. Rule headers are a tangible and visual cue within the source code itself. They provide information tying the rules to the requirements they are satisfying. In addition, historical data is provided and can be used in concert with the test data to maintain version consistency. All products of the tools during the lifecycle are stored in the SDF. The SDF contains the information for any final, deliverable products.

![Figure 3. Four Basic Tools Provide a Consistent Foundation for the Development of Knowledge-Based Systems.](image-url)
 TOOL APPLICATIONS

gIBIS (Graphical Issue Based Information System)

One of the challenges associated with knowledge acquisition is finding a systematic approach for capturing and documenting the decision logic, design rationale, algorithms and procedural dependencies. The mechanism needs to be easy to use, accommodate frequent revision of information, and provide precise terminology that can be shared by domain experts and software engineers. The tool selected to manage knowledge acquisition was gIBIS, an acronym for graphical Issue Based Information System.

The gIBIS tool is an implementation of the Issue Based Information System (IBIS) rhetorical model and was created by researchers at Microelectronics and Computer Technology Corporation (MCC). The IBIS model provides a method of documenting a "conversation" between many participating members. IBIS supplies a set of typed messages and a set of specific communications among the messages. A "conversation" evolves from posting an initial message. The message is the seed from which other pertinent pieces of data relate.

gIBIS is a hypertext-based tool that provides a graphical representation of an IBIS network which the user can create and modify [4]. The tool supports the message types specified in IBIS as well as associated relationships between messages. A hypertext node is assigned to each message and/or relationship. Each node consists of the graphical image and the associated textual attributes. Information is represented as a network comprised of messages connected by relationships.

As Figure 4 shows, the basic gIBIS interface is divided into four major windows: a graphical

![Figure 4. The gIBIS Graphical Interface Provides an Intuitive Medium for Both the Developer and Expert.](image-url)
browser, a structured node index, a control panel, and the inspection window. The user builds a gIBIS network in the browser window using rectangular icons to symbolize messages, and arrows to represent relationships. Message or relationship node attributes may be viewed or modified in the inspection window upon user request. The index window displays a table containing node number, type, and a label which is also displayed under the node icon. The control panel provides the user command interface.

For our applications, the IBIS model was appropriate for the planning and design activities associated with knowledge acquisition. Information contained within the gIBIS network facilitated the transition from requirement definition to knowledge base design and implementation. In addition, the graphical structure of the network allows software engineers to represent decision logic coupled with top-level software design.

In our knowledge acquisition sessions, we constrained message definitions to Issue, Position, and Argument. The set of legal message types and their relationships is shown in Figure 5. The central message type is an Issue which is used for stating a question or problem. Follow-up messages for Issues are Positions which 'respond to' the Issue. Follow-up messages for Positions are Arguments which 'support' the Position, or another Issue that 'is suggested by' the Position. For documentation purposes, we added Expert Rule, Reference, and Message message types. Expert Rule messages, represented as an AI icon, are used to explicitly state an 'if-then' rule associated with an Argument. The Reference messages document assumptions and Message messages identify open issues. A typical gIBIS network is shown in Figure 6.

![Figure 5](image1)

**Figure 5.** The Tailoring of the IBIS Method Provides A Common Data Representation.

The gIBIS networks provide a systematic method for formalizing knowledge obtained from expert interviews and requirement documentation. After a knowledge acquisition session, decision logic and procedural information are used to create or modify a gIBIS network. Then, the software engineers and domain experts use the gIBIS network to focus review meetings. The graphical representation of knowledge is intuitive for non-software professionals to understand. Clarification of requirements was simplified due to precise knowledge representations. One unexpected benefit to using gIBIS was that it also provides a method for prototyping the design logic and paved the way for transition to knowledge base implementation.

![Figure 6](image2)

**Figure 6.** gIBIS Networks Are Easily Understood By The Domain Expert and Easily Modified By the Software Engineer.

gIBIS is used to support the program management tasks by providing a convenient method to document requirement traceability and establish complexity measures. The structure of the knowledge base strongly parallels the gIBIS network, so requirement traceability to rules was accomplished by indicating which gIBIS node was satisfied. Development met-
rics, such as number of nodes or change history, is extracted by using the integrated gIBIS database query language to collect data from the node attribute information.

gIBIS has proven to be a valuable tool during the capture and documentation of system requirements and decision logic. Not only is gIBIS used to document design rationale, it also provides a means of gathering design metrics data. gIBIS is the tool that bridges the gap between requirement analysis and design.

Rule Headers/Software Organization

Lack of clear traceability of design to requirements has plagued project managers, configuration control, and (perhaps most disastrous) the customer. Often the reason for delivery schedule slips is mismanagement of requirement traceability. This is a particularly difficult problem for a KBS because requirements are constantly being elaborated. A traceability approach needs to be concise, easy to maintain, and mandated prior to software development. We choose to document requirement traceability in the header of each rule and data record definition. The header contains fields for identification, traceability, and revision history.

To stay in synchronicity with changing requirements, traceability was achieved via gIBIS. Due to the reasons stated previously, the primary vehicle for documenting requirements changes was the gIBIS network. Therefore, tracing to gIBIS nodes maintains concurrency between requirements and implementation.

During the prototype, our software engineers adhere to the policy of consistently appending headers to rule and data definition records. A positive side effect to this approach is that the headers provide visual separators for rules. The knowledge-base was also partitioned into modules, defined as a set of related rules. Module headers were added to describe the processing of the module.

The profit resulting from descriptive headers was apparent in the test phase and furnished a method to obtain project metric data. This is illustrated in Figure 7. Test data and development metrics could be extracted from the headers with support utilities.

![Figure 7. Rule Headers Maintain Requirement Traceability, Modification History, and A Useful Visual Cue Within the Software.](image-url)
stability, was derived from the revision history. Requirements traceability to design components is maintained within the database so the method of determining which components required retest due to code revisions was easy implemented.

Our database tool of choice is a widely known PC-based database management system (DBMS) that supplies a sophisticated relational database manager, flexible development environment, powerful user interface, and built-in report generator. Overall, this product provided all the features needed to easily maintain metric data and generate project management reports.

In terms of the KBS development cycle, the requirements were not reasonably stable until several iterations through every development phase. As indicated in Figure 8, the initial database was generated after the requirements document was published. When requirements were not "testable" or too general, derived requirements were defined to further clarify the original ones. The information provided for each requirement includes identification, description, revision history, requirement parentage, and design allocation.

The database is used as the central repository for the system development statistics. It also provides an automatic method of generating critical development progress reports.

The use of the DBMS has been expanded to include the utility of generating the actual code for input/output (I/O) data. The interfaces to most of our systems are critical to their success; we have focused on their early definition to ease downstream development problems. The requirements are traced to the I/O items directly which makes the limited code generation capability possible. This allows the code to remain consistent with a volatile set of requirements.

**Software Development File**
Most projects require a team of engineers working together to reach the goals of the project. During all development phases new information is surfacing, existing information is changing. Product delivery schedules are stringent, therefore, the development team must employ a method of communication that ensures pertinent data are readily available and reflect the current state of project data. Our ap-

![Figure 8. The Requirements Database Maintains Project Status In Terms of Schedule and Completeness.](image)
proach addresses the issues of information currency and accessibility by utilizing a conventional software mechanism: the Software Development File (SDF).

The SDF is a book(s) shared by all software team members which contains all informal requirement analysis data, proposed design solutions, test cases and results, source code and support file listings. Each software team member is responsible for storing the latest copy of his products in the SDF as soon as they are "stable". Figure 9 shows the organization of the SDF and illustrates the type of products that are generated during each development phase.

The SDF is established early in the project to organize the knowledge acquisition activity. Notes from interviews with the experts, and the resulting gIBIS diagrams are stored in the requirements section of the SDF. The design section maintains all data definition formats plus top-level views of the software structure. The code section contains the latest operational version of the system plus previous operational versions. The testing section of the SDF contains all the test cases along with the results. The test cases are organized functionally to aid in retrieval for reuse. Project schedules, status reports, etc. are kept in the development status section.

The SDF provides insight to where you have been, and helps establish what remains to be completed. By faithfully storing the latest
information in the SDF, the communication among team members is enhanced. Development progress metrics can be derived for the current and future projects.

Since the SDF contains all development history data, we can analyze the development steps. This analysis may be executed with respect to schedule time to see if the task was performed at the correct point in the development process. An example of this is the determination of whether the requirements specifications data was stable enough to justify a commitment to code. Process inspection is facilitated and bottlenecks or other areas may be candidates for improvement. The SDF is key in the continual improvement of KBS development lifecycle. Future projects should, and are, based on history by retrospectively examining the SDF.

**CONCLUSIONS/ LESSONS LEARNED**

Over the course of several KBS development projects, we have created a process and selected a suite of tools that assist in its execution. The projects have ranged from typical interactive-type systems to embedded and flight-worthy code.

We have found that knowledge acquisition is key to the KBS lifecycle. As such, we have employed a sophisticated tool, gIBIS, to combat this problem. This tool provides a visual and expedient methodology for the categorization of data, its representation and relationships, intuitive interface, and high-level prototyping capabilities. Indeed, gIBIS is the cornerstone of our process.

We have coupled traditional software engineering mechanisms to gIBIS to create a flexible environment for the development and management of KBSSs. Rule headers are used to document traceability to requirements and utilization evolution. They contain information regarding the purpose and intent of the code along with information useful in system testing. A relational database management system is used to formally and efficiently document the requirements for the project. This tool also bridges the gap between the requirements definition and implementation phases in the overall lifecycle. Requirements are allocated to software modules and tracked in their completeness. Source code is generated for the I/O interfaces. The Software Development File maintains the "corporate legacy" for the project.

The process we have developed to manage knowledge-base systems is still evolving and suffers from the lack of a more automated and integrated tool suite. The weakest link is still the man-in-the-loop handling of bookkeeping tasks. We believe gIBIS should be the basis for this tool since it already encompasses many of the desired attributes. Flexibility and utility should still take precedence over current CASE environments. Methodologies should be inherent in the system while not being overpowering. gIBIS provides the best of both worlds.

**REFERENCES**