1993

Process Knowledge Based Rapid Prototyping for Requirements Engineering

Ramesh, Balasubramaniam; Luqi

Process Knowledge Based Rapid Prototyping For Requirements Engineering

Balasubramaniam Ramesh

Code AS/RA
Naval Postgraduate School
Monterey, CA 93943
E-mail: ramesh@is1.as.nps.navy.mil

Luqi

Code CS/LQ
Naval Postgraduate School
Monterey, CA 93943
E-mail: luqi@cs.nps.navy.mil

Abstract

Rapid prototyping offers an iterative approach to requirements engineering to alleviate the problems such as uncertainty, ambiguity, and inconsistency inherent in the process. Further, as the systems development process is characterized by changing requirements and assumptions, involving multiple stakeholders with often differing viewpoints, it will be beneficial to capture in a structured manner the history of the development process. In this paper, we describe how CAPS (Computer Aided Prototyping System) as a prototyping tool, augmented with REMAP (Representation and Maintenance of Process Knowledge) framework for reasoning with process knowledge captured during requirement engineering, helps firm up software requirements through iterative negotiations between various customers and designers via examination of executable prototypes in the context of evolving requirements.

1 Introduction

A major problem with the traditional waterfall lifecycle approach is the lack of any guarantee that the resulting product will meet the customer's needs. Often users will be able to identify true requirements only by observing the operation of the system. To alleviate the problems inherent in requirements determination for large, parallel, distributed, real-time, or knowledge-based systems, current research suggests a revised software development life cycle based on rapid prototyping [1],[2]. CAPS (Computer Aided Rapid Prototyping System) has been built to help software engineers rapidly construct prototypes of proposed software systems.

During the rapid prototyping process, users modify, refine and elaborate requirements by observing prototypes. This iterative process is characterized by 'what if' analysis of requirements and assumptions. The user may use the feedback obtained by observing the performance of the system under a variety of potential scenarios in shaping final requirements. It will be extremely beneficial to capture the process knowledge characterizing the evolving nature of requirements and assumptions during the rapid prototyping process so that the users may *replay* various scenarios in firming up the requirements or understand the repercussions of changing requirements and assumptions. The process of arriving at specifications involving multiple stakeholders with differing viewpoints and assumptions can be aided by REMAP (Representation and MAintenance of Process Knowledge), a tool for capturing, maintaining and reasoning with process knowledge. REMAP is based on a conceptual model of the requirements engineering process and has been developed based on an empirical study in the context of requirements engineering [3].

It should be noted that, in this paper, the term *design* is used to refer to any activity that leads to the creation of artifacts including those even the early phases of the systems development process. Further, the term *process knowledge* is used to refer to the rationales behind decisions rather than to an algorithmetic program to describe the process of developing a system (as used in the work on *process programming*[4]).

2 Rapid Prototyping with CAPS

The problem with requirements engineering is amplified in the case of hard real-time systems, where the potential for inconsistencies is greater [5],[6]. For these systems, requirements are difficult for the user to pro-

vide and for the analysts to determine. Toward this end, an integrated set of software engineering tools, the CAPS [7], has been designed to support quick prototyping of such complex systems by using easy to understand visual graphics [8] mapped to a tight specification language, which in turn automatically generates executable Ada [9] code.

2.1 Computer Aided Prototyping System (CAPS)

The main components of CAPS are the prototype system description language (PSDL), user interface, software database system, and execution support system.

The prototype system description language (PSDL): PSDL [10] serves as an executable prototyping language at a specification or design level and has special features for real-time system design. The PSDL model is based on data flow under real-time constraints and uses an enhanced data flow diagram that includes non-procedural control and timing constraints.

User Interface: The graphics editor, in the User Interface, is a tool which permits the user/software engineer to construct a prototype for the intended system using graphical objects to represent the system [11], [12]. The graphical objects presented to the designer include operators, inputs, outputs, data flows, and operator loops. A browser allows the analyst to view reusable components in the software base. An expert system provides the capability to generate English text descriptions of PSDL specifications. Together, these tools facilitate common understanding of PSDL components by users and software engineers alike, thereby reducing design errors.

Software Database System: The software database system provides reusable software components for realizing given functional (PSDL) specifications. It consists of a design database containing PSDL prototype descriptions for all projects, software base containing PSDL descriptions and implementations of all reusable software components, and software design management system that manages and retrieves the versions, refinements and alternatives of the prototypes in the design database, as well as the reusable components in the software base.

Execution Support System: The execution support system [13] consists of a translator, a static scheduler, a dynamic scheduler, and a debugger. The translator generates code that binds together the reusable components extracted from the software base. If the static scheduler fails to find a valid schedule, it provides diagnostic information useful for determining the cause of the difficulty and whether or not the difficulty can be solved by adding more processors. As execution proceeds, the dynamic scheduler invokes operators without real-time constraints in the time slots not used by operators with real- time constraints [14]. The debugger allows the designer to interact with the execution support system.

3 REMAP Model

REMAP model relates design rationale to the artifacts created during the systems development process. This model was developed using an empirical study of problem solving behavior of individual and groups of systems analysts engaged in a simulated requirements engineering exercise.

Figure 1 shows the primitives of the REMAP conceptual model. This model is described in detail in [3] and can explained in brief as follows: The systems design deliberation process involves discussion and resolution of issues or concerns that must be addressed to satisfy user requirements. Requirements represent the *goals* to be satisfied by the design process. The design process involves the refinement, elaboration and modification of the initial requirements. This process involves argumentation among various stakeholders in the form of discussion and resolution of issues. This argumentation process is modeled by extending the Issue Based Information Systems [15] (IBIS) model. The primitives in the IBIS model are issues, positions and arguments and relationships among them. These are shown in the dotted segment in Figure 1. Issues are questions or concerns, positions are alternatives that address an issue, and arguments either support or object to positions. In addition to IBIS primitives, the assumptions that underly arguments are also explicitly represented. Further, decisions that represent the resolution of issues by selecting one or more positions are also part of the REMAP model. In the context of systems design, the decisions lead to constraints which define design artifacts. The REMAP model represents the iterative nature of the design process. These "output oriented" primitives in REMAP are explicitly related to the argumentation (or design rationale) and requirements refinement process. These explicit linkages facilitate refinement and modification of the "outputs" in the context of changing rationale, requirements and assumptions.

A prototype environment has been developed to capture and reason with process knowledge.

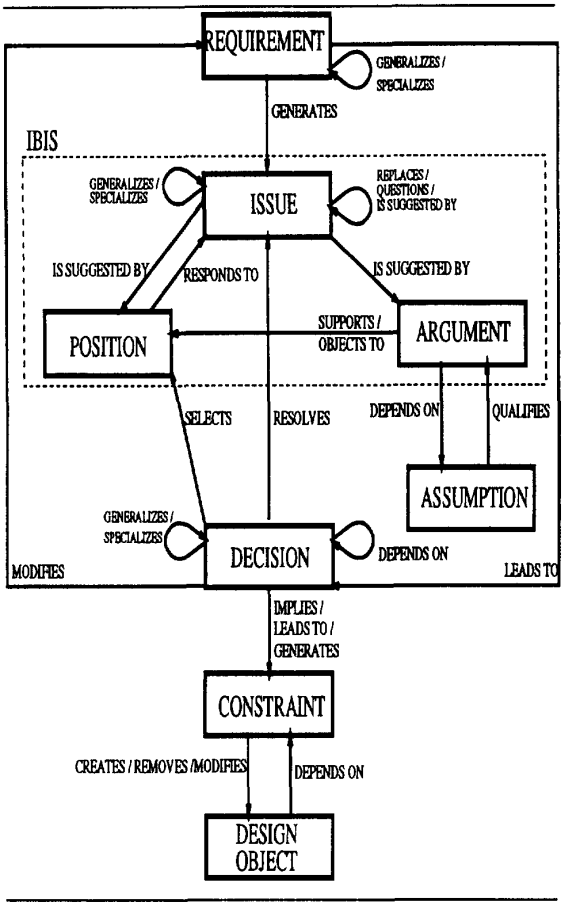

Figure 1: Conceptual Model

REMAP environment provides facilities for instantiating, querying and modifying objects interactively, thereby facilitating incremental acquisition of process knowledge. Further, a hypertext-like browsing facilitates convenient traversal of the knowledgebase. A reason maintenance system maintains the dependencies among various primitives of our model. Lubars [16] has investigated the use of IBIS style representation for representing dependencies. It should be noted that his research uses the IBIS primitives to represent dependencies among artifacts, whereas our research focuses on representing the dependencies among the extended-IBIS primitives using the Reason Maintenance framework.

4 Process Knowledge Based Rapid Prototyping

4.1 Prototyping

The Computer Aided Prototyping System (CAPS) is used to create software prototypes, which are mechanically processable and executable descriptions of simplified models of proposed software systems. It is also used to modify these models frequently in an iterative prototype evolution process for the purpose of firming up the requirements. The prototyping process consists of two basic operations: prototype construction/modification based on evolving requirements and code generation [17].

Prototype construction is an iterative process that starts out with the user defining the requirements for the critical aspects of the envisioned system. Based on these requirements, the designer then constructs a model or prototype of the system in a high-level, prototype description language and examines the execution of this prototype with the user. If the prototype fails to execute properly, the user then redefines the requirements and the prototype is modified accordingly. This process continues until the user determines that the prototype successfully meets the critical aspects of the envisioned system. Following this validation, the designer uses the validated requirements as a basis for the design of the production software.

The code generation stage focuses on transforming and augmenting the prototype to generate the production code. Prototypes are built to gain information to guide analysis and design, and support automatic generation of the production code.

Prototype construction and modification is an evolutionary and exploratory activity. If the target system is intended to satisfy multiple users with different and often conflicting viewpoints and needs, the requirements engineering exercise involves arriving at a consensus or compromise among the participants. Also, the users may begin with requirements that can not be satisfied within the constraints on available resources or may make assumptions that are either invalid or not shared by other participants. In such situations, a mechanism to capture the evolution of requirements will facilitate arriving at a compromise on requirements and consensus on assumptions that guide the design process. The process of modification of the requirements can be aided by mechanisms for the representation of and reasoning with process knowledge.

A prototype may not implement all of the functions of the proposed system, since the prototyping

effort is focused on the aspects of the requirements that are unknown or uncertain. After the requirements have stabilized, the design and the structure of the prototype must be augmented to account for these additional functions. These augmentations can be expressed in the prototyping language to provide an early check on the adequacy of the final version of the system structure. Even if the additional requirements can not be formalized in the prototyping language, an informal representation can be maintained using REMAP model primitives and related to the specifications.

A prototype may not meet all of the performance requirements, or may not operate in the same hardware and software environments as the proposed system. REMAP provides a mechanism to represent the assumptions under which the prototype and the target system will operate and to identify the ramifications of changes in these assumptions. This approach may provide the benefits of rapid prototyping in both the requirements analysis and system maintenance activities when assumptions change.

4.2 Domain Specificity and Requirements Traceability

Using CAPS to engineer requirements offers clear advantages over determining requirements manually. The prototype system description language is focussed on the domain of hard real-time systems and as such offers a common baseline from which users and software engineers describe requirements. Defining requirements in a domain specific language results in more efficiency and fewer errors because it constrains the way users and engineers can describe a particular requirement. In addition, the interpretations of requirements stated in a domain specific language such as PSDL are unambiguous, whereas requirements stated in English are often misunderstood.

Requirements traceability is essential to accurately map changed requirements into the implementation. CAPS offers basic requirements traceability through the "BY REQUIREMENTS" statement in the PSDL grammar. This statement allows software engineers to associate actual requirements with the definitions of module interfaces and constraints by annotating the interface or constraint definition with an identifier.

In addition to maintaining traceability of requirements to design artifacts, it is important to capture information about where the requirements come from[18]. REMAP aids this objective by providing a mechanism to capture the history of the requirements evolution process.

4.3 Process Knowledge in Requirements Engineering

The requirements for a software system are expressed at different levels of abstraction and with different degrees of formality. The highest level requirements are usually informal and imprecise, but they are understood best by the customers. The lower levels are more technical and more precise, are better suited for the needs of the system analysts and designers, but they are further removed from the users' experiences and less well understood by the customers. Because of the differences in the kinds of descriptions needed by customers and developers, it is not likely that any single representation for requirements can be the "best" one for supporting the entire prototyping process. CAPS augmented with REMAP provides a wide spectrum of such representations from informal requirements to prototyping language specifications.

In the context of prototyping, the requirements are used as a means for bridging between the informal terms in which users and customers communicate and the formal structures comprising a prototype. We believe that a useful representation for this information is a hierarchical goal structure, where informal customer goals are refined and defined by several levels of increasingly formal and precise subgoals, with different notations used at different levels. The elaboration, refinement and modification of requirements in the form of a hierarchy can be represented both at the level of informal specifications as well as at the level of formal specifications. In REMAP, the discussion and resolution of issues may lead to other issues to be resolved, leading to a hierarchy of issues. The goal directed nature of the design process, therefore, can be represented as a hierarchy of issues to be resolved with attendant rationale behind the process of their resolution. The application of this structure to prototyping is illustrated by example in Section 5.

4.4 Design Replay

During various phases of the life cycle, facilities to retrace the progression of steps that the design process went through can be beneficial. Such a facility is essential if corrective action is to be taken to redo parts of the design process that may have led to dead ends or undesirable solutions. Further, design history information is useful in *replaying* the steps to facilitate understanding the *evolution* of the system as well as identifying the *choice points* where alternative decisions could lead to different solution paths.

In our representation of design history, we maintain the belief times of assertions in the knowledge base. For each assertion in the knowledgebase, information on the time period during which it was believed, (i.e., it was a part of the current knowledge base) is maintained. This information can be used to trace the steps in the evolution of the design history or *design replay.* It can also be used to categorize decisions, assumptions etc, according to the phase of the development cycle. During design replay, additional positions that resolve an issue could be defined leading to a different set of alternatives to be considered. Also, reevaluation of the already defined positions based on different criteria may lead to different decisions.

Design replay can be chronological or dependency directed. The dependency information maintained by REMAP can be used to identify the decisions, and it turn the issues that lead to the creation of a specific design object. Then the issue could be reexamined to generate alternate solutions. Similarly, using dependency directed backtracking, the set of assumptions that a design object depends on can identified. The assumptions could be reevaluated to revise beliefs in them. Such a revision of beliefs could lead to a different design solution.

5 Example

To illustrate the concepts described above, we describe a small sample application generated in CAPS. The purpose of the exercise is to verify the requirements for a robot control system by creating a prototype software system using CAPS.

Figure 2 is an extended dataflow model of the basic operators in the example. The operators (shown as circles) in the left are time non-critical and the ones in the right are time critical. The numbers above the operators denote the Maximum Execution Time of the operators. This representation is part of a PSDL design which is translated by CAPS into an executable prototype. An important step in the process is to ensure that the specification is feasible with respect to the timing constraints given in the PSDL. The CAPS static scheduler performs an analysis of the timing constraints and reports when a schedule is not feasible. After the user has filled in the additional details into the PSDL specification, Ada code is automatically generated to meet the specification.

The following scenario describes a design situation in which discussion about a requirement is conducted during the prototyping process in arriving at the above specification. REMAP supports the capture of such

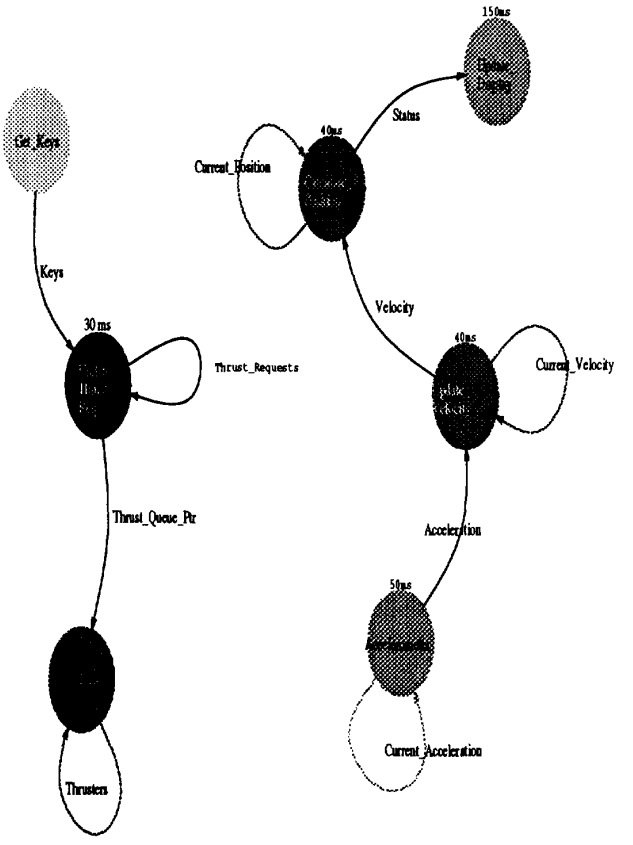

Figure 2: Operator Specification

deliberations by providing mechanisms for designers and users to conduct their *conversations* using the primitives of the conceptual model. As the design deliberation proceeds, various issues, positions, arguments, assumptions, decisions etc. are defined by the users.

Figure 3 illustrates the results of a deliberation session. The prototyping team is engaged in the deliberations of the requirement *temporal operations definition.* The team identifies *timing constraints* as a major issue or question to be resolved before the requirement can be satisfied; i.e., determining the maximum execution times of various operators as in Figure 2. The first position proposed is option1. The Editor window in Figure 3 shows the details of option1 which include the timing constraints on the maximum execution times of four operators: calculate position, update display, up-

date velocity and accelerometer. These assignments are supported by the argument that this is the only feasible option based on their experimentation with other possible timing constraints. A group member points out that the argument is based on the assumptions that the system will use only a single processor and that the rate at which the display mechanism refreshes is quick (i.e., the display can be updated every period in which a data stream from the sensor is received). In this example, these two assumptions have been ascertained as valid (denoted by their white background). The effect of these assumptions is propagated by a reason maintenance system. This propagation leads to valid belief status of the argument *feasible option* and which makes *option1* a valid position. The design team also recognizes the fact that another position (option2) may be more appropriate during deployment of the system. This position is supported by the argument that tighter timing constraints are needed to satisfy the high performance need in a real-life situation. This argument is based on the assumptions that a multi processor environment will be used to support the design and that a quick display mechanism is available. It should be noted that the assumption *multi processor* negates the assumption *single processor*. As the assumption *single processor* has been declared valid earlier, the assumption *multi processor* will automatically be declared invalid. It has further been defined that an argument gets valid support only when *all* the assumptions qualifying it have valid support. In this context, therefore, the argument *high performance need* and hence *option2* do not have valid support. Finally, the net effect of the evaluation of assumptions in this context is that specification1 (which corresponds to Figure 2) is declared valid and hence a prototype is generated by CAPS satisfying this specification. Further specification2 is declared invalid.

5.1 Maintenance of Changing Requirements

Changing requirements and assumptions necessitate changes to software systems. In our model, changes to design rationale will automatically trigger changes in the belief status of design solutions thereby suggesting redesign [19]. Since various components of the process knowledge that lead to the design solution are tightly related, changes to the constraint set resulting out of changed assumptions, decisions or requirements can initiate the synthesis of a new design solution in CAPS. Thus the process of maintenance will occur as changes in the process knowledge.

In the above example, a change in the validity of the assumption *single processor* would trigger changes in belief status /validity of relevant arguments, positions, decisions etc. leading to specification1 losing its support and specification2 gaining support.

5.2 Replay

During a prototyping session, the users may want to replay the various scenarios that are valid under various sets of assumptions to identify the most desirable solution (i.e., specification). During such replay, the users may want to evaluate assumptions and ascertain the ramifications of such changes.

In our example, the users may start with the identification of assumptions on which specification1 derives its validity. A careful evaluation of the assumptions may suggest that the available display mechanisms are not quick enough to handle all the data streams from the sensors. As the display mechanism can not handle the rate at which input data arrives, it is wasteful to try to update the display every cycle in which sensor data is received. Instead, the display could be updated once in, say 3 or 4 processing cycles, depending on its capacity. When the assumption *quick rate of display* is retracted, the assumption *slow rate of display* that negates it gets valid support. This may lead to another position, say Option3, obtaining valid support. The end product of these evaluations will be a new specification, which can be used by CAPS to generate an executable prototype.

5.3 Decision Support

Depending on the nature of decisions, a decision process as above can be supported with varying degrees of automation by REMAP. Dependency directed backtracking can be invoked by the user to identify the assumptions on which a specification gets its validity. Further, maintaining *degrees of belief* in assumptions can help decide which of the assumptions should be retracted. Then, the system can be instructed to revoke the assumption(s) in which the users have the least degree of confidence. Also, the system can be provided with domain knowledge to validate assumptions as assertions are made into the knowledgebase. For instance, the evaluation of whether a display is *quick* or *slow* can be made by the system if the rate of display of the display mechanism is asserted into the knowledgebase and the evaluation of the assumption is performed by a deductive rule. As the facts about the display rate changes in different situations, REMAP can automatically evaluate assumption that depend

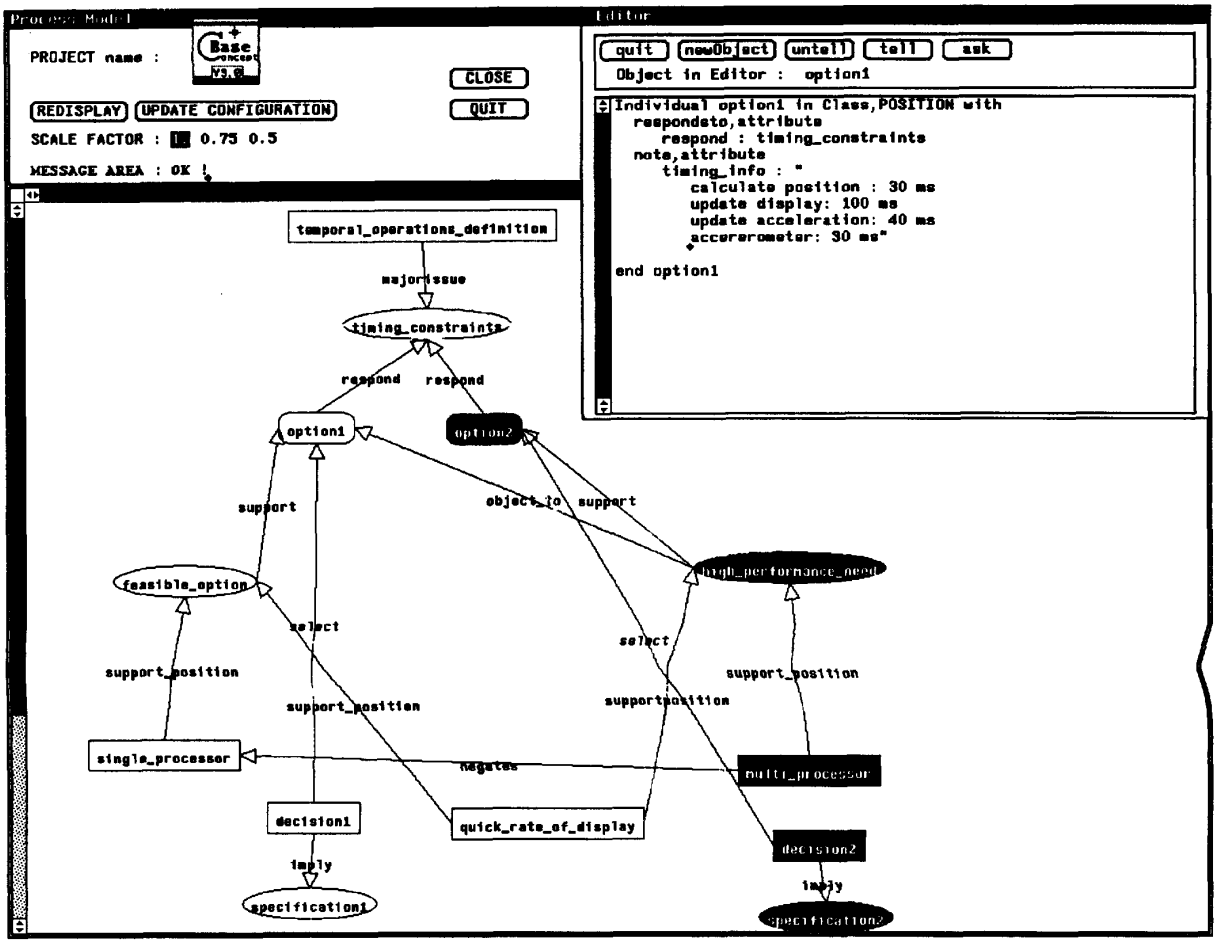

Figure 3: Design Deliberation Instance

on this information. Thus, besides providing support for storing and tracking design alternatives, the system can provide active support in the resolution of issues. Finally, mechanisms for computing belief status of an assumption based on the beliefs of individual team members can be incorporated.

The users may obtain feedback from CAPS by the execution of a prototype. This, in turn, could lead to the definition of additional issues and deliberations to resolve them.

The dependencies of the components of the software base on the specifications can be maintained by the reason maintenance system. Then, the components to be modified by the changed belief status in assumptions can be readily identified, improving the efficiency of the prototype reconstruction process.

6 Conclusion

Rapid prototyping offers an iterative approach to requirements engineering to alleviate the problems of uncertainty, ambiguity, and inconsistency inherent in the process. CAPS (the Computer Aided Prototyping System) has been built to help software engineers rapidly construct software prototypes of proposed software systems. CAPS augmented with REMAP helps firm up software requirements through iterative negotiations between customers and designers via the maintenance of process history and examination of executable prototypes.

Especially in the case of many interacting assumptions and validation/verification efforts by multiple agents, automated tracking of the dependencies be-

tween assumptions, issues, and design decisions is very valuable. Keeping track of arguments that did not prevail is useful because at the early stages of development, information is highly uncertain. Users are quite likely to pose a position, retract it when some unpleasant consequences surface, and then go back to the original position when even more unpleasant consequences surface for a rival position. Some current challenges are automating assessment of the relative importance of competing goals, using this information to resolve conflicts between arguments supporting different positions, and automating the materialization of the design objects corresponding to different combinations of design decisions.

Using a prototype system description language enables engineers and users to quickly focus on the pertinent requirements of their system resulting in increased efficiency and fewer requirement errors. Using process knowledge, users can resolve any conflicts in their viewpoints, surface and evaluate assumptions behind these viewpoints and evaluate the ramifications of changes in such assumptions.

References

[1] V. Berzins and Luqi, "Rapid prototyping real-time systems," *IEEE Software*, September 1988.

[2] M. Tanik and R. Yeh, "The role of rapid prototyping in software development," *IEEE computer*, vol. 22, pp. 9–10, may 1989.

[3] B. Ramesh and V. Dhar, "Supporting systems development using knowledge captured during requirements engineering," *IEEE Transactions on Software Engineering*, June 1992.

[4] L. Osterweil, "Software processes are software too," in *Proceedings of the 9th International Conference on Software Engineering*, (Monterey, CA), pp. 2–13, April 1987.

[5] W. Beam, *Command, control and communications engineering.* McGraw-Hill, 1989.

[6] J. Stankovik and K. Ramamritham, *Hard Real-time systems tutorial.* computer soceity press, 1988.

[7] Luqi and M. Ketabchi, "A computer aided prototyping system," *IEEE transactions on software engineering*, october 1988.

[8] N. tactical interoperability support activity, "Pdw 120-s-00533 (rev. b, change 4), over the horizon targeting (oth-t) gold reporting format." Report, June 30 1989.

[9] G. Booch, *software engineering with ada.* benjamin/cummings publishing company, inc., 1987.

[10] Luqi, V. Berzins, and R. Yeh, "a prototyping language for real-time software," *IEEE transactions on software engineering*, october 1988.

[11] M. Linton, J. Vlissides, and P. Calder, "Composing user interfaces with interviews," *IEEE computer*, february 1989.

[12] N. Aeronatics and space administration, "Transportable applications environment (tae) plus," 1990 january.

[13] E. Borison, "Program changes and cost of selective recompilation," Technical Report CMU-CS-89-205, Carnegie-Mellon University, Computer Science Dept., July 1989.

[14] A. Mok, "a graph baed computational model for real-time systems," in *proceedings of the IEEE international conference on parallel processing*, (pennsylvania state university), 1985.

[15] J. Conklin and M. Begeman, "gibis: A hypertext tool for exploratory policy discussion," *ACM Transactions on Office Information Systems*, vol. 6, pp. 303–331, October 1988.

[16] M. D. Lubars, "Representing design depencies in an issue-based style," *IEEE Software*, pp. 81–89, July 1991.

[17] Luqi, "Software evolution through rapid prototyping," *IEEE computer*, may 1989.

[18] A. Finkelstein, "Tracing back from requirements," in *Proceedings of the colloquium on tools and techniques for maintaining traceability during design*, (Savoy place, UK.), IEE, December 1991. IEE Digest No. 1991/180.

[19] V. Dhar and M. Jarke, "Dependency directed reasoning and learning in systems maintenance support," *IEEE Transactions on Software Engineering*, vol. 14, pp. 211–227, February 1988.