(2)

Honeywell ProtoTech Phase I Final Technical Report

Contract N00014-90-C-0015
ARPA order Number 6983
Dec 1989 - Jun 1992

John Kimball, Tim King, Aaron Larson, Chris Miller, Jon Ward
Honeywell Systems and Research Center
612/782-7308
alarson@src.honeywell.com

June 17, 1992

Honeywell ProtoTech Phase I Final Technical Report

Contract N00014-90-C-0015
ARPA order Number 6983
Dec 1989 – Jun 1992

John Kimball, Tim King, Aaron Larson, Chris Miller, Jon Ward
Honeywell Systems and Research Center
612/782-7308
alarson@src.honeywell.com

June 17, 1992

Technical Report CS-C92-002

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE**	**3. REPORT TYPE AND DATES COVERED** Final Technical Report/Dec 1989-June 199

4. TITLE AND SUBTITLE
Honeywell ProtoTech Phase I Final Technical Report

5. FUNDING NUMBERS
Contract
N00014-90-C-0015

6. AUTHOR(S)
John Kimball, Tim King, Aaron Larson, Chris Miller, Jon Ward

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Honeywell Systems & Research Center
3660 Technology Drive
Mpls, MN 55418
MN65-2100

8. PERFORMING ORGANIZATION REPORT NUMBER
CS-C92-002

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report describes the Environment for Software Prototyping (ESP). Within ESP, prototyping is viewed as an experimental process whose results drive the evolution of a domain theory in the context of a megaprogramming environment.

ESP is used to evaluate the nature and efficiency of technolgies that address requirements unique to prototyping. ESP consists of a component shelf containing reusable components (i.e., a component consists of a specification and a set of implementations), a repository for reusable software architectures (i.e., an architecture defines the structural characteristics of a system), and a workbench (i.e., a workspace for conducting prototyping experiments). ESP provides an environment for bootstrapping a megaprogramming capability.

14. SUBJECT TERMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**

Contents

Honeywell
ProtoTech Phase I Final Technical Report

John Kimball, Tim King, Aaron Larson, Jon Ward
Honeywell Systems and Research Center
(contact alarson@src.honeywell.com)

June 26, 1992

1 Introduction

This document is the final report for the Honeywell ProtoTech[1] (Prototyping Technologies) Phase I contract[2]. In this document, we define our understanding of ProtoTech's mission, and we outline the work undertaken by the Honeywell/U. of Maryland team addressing the overall ProtoTech goals.

2 ProtoTech Objectives

Formal DARPA ProtoTech contracts were started in early 1990 based loosely around the requirements specified in [1], which separated the requirements into those for a prototyping language and environment. The original five ProtoTech teams have worked as a community to address both language and environment concerns, develop a common understanding of what prototyping is, how it is used, and have evolved each of the teams' baseline technologies. Collectively we have defined prototyping as:

> Prototyping is an experimental activity intended to expose properties of a potential product before design decisions are carved in stone.
>
> The purpose of our program is to help users understand what experimental questions to ask, to streamline acquisitions of their answers, and to organize use of this information in developing and evolving the product.[17]

This definition recognizes that prototyping is not an end goal in software engineering, but rather contributes toward the attainment of another goal. Prototyping is part of many

[1]ProtoTech was previously known as the Common Prototyping Languages (CPL) program.
[2]This work sponsored by DARPA via ONR contract N00014-90-C-0015.

software engineering methodologies, and is practiced in different phases of the software lifecycle, even though it tends to be most heavily used in the earlier tasks (i.e., requirements, specification, and design). Therefore, prototyping must fit within the software engineering lifecycle that it is to support. With this fact in mind, it is helpful to restrict the domain of interest to a subset of the many possible software engineering processes. Such a restriction was formally provided in January 1992 when the ProtoTech community was tasked with determining:

> "What should be the ubiquitous elements of the future directly supporting components-based software engineering?" — *Bill Scherlis*[17]

Components-based software engineering (a.k.a. megaprogramming or reuse) is the disciplined approach of codifying an understanding of domain-specific knowledge into software artifacts, some of which are executable, (e.g., architectures and components), with the expressed intent that the components be usable in multiple contexts. The cost for developing a particular artifact is amortized by increasing the artifact's reusability and thereby increasing its value. Megaprogramming not only requires that software engineering be cost-effective, but suggests the approach of using hierarchical decomposition and architecture specification to achieve this result.

Thus, the ProtoTech goal is to maximize the effectiveness of prototyping within a megaprogramming-based software engineering lifecycle, not only in the application of prototyping to the construction of component-based systems, but also in the construction of components for such systems, and in the maintenance and evolution of both the components and the composite systems.

An interesting perspective is to view megaprogramming as an effort to capitalize[21] the results of software-intensive concurrent engineering. The current DARPA "Domain Specific Software Architectures" (DSSA) contracts are clear examples of this approach. The DSSA program includes, for example, multiple, multi-disciplinary teams addressing realization of control theory in actual operating products. The teams consist of not only domain specialists and software engineers, but also people doing foundational work.

3 Assumptions and Interpretation of Goals

In this section, we describe Honeywell's perspective on the ProtoTech goals, and explain why prototyping is an important aspect of megaprogramming.

From Honeywell's perspective, there are two undeniable facts:

1. Software is expensive to construct.
2. Customers don't buy software, they buy functionality.

The goal then is to minimize the amount of n w software construction needed to achieve a given level of functionality. This simple reasoning is the basis of megaprogramming — *assemble* systems, don't *construct* them. There are however two more undeniable facts:

3. Product functionality that is competitive today, isn't tomorrow!
4. Software components that weren't designed to work together, don't!

The design and architecture of a system (or group of systems) must be amenable to change, and must satisfy a sufficiently large group of users to support the cost of production. The inevitability of change, coupled with the nearly insatiable customer demand for increased functionality, pushes vendors to be increasingly responsive. The large amount of detail needed to specify a unit of user-level functionality requires good designs to partition concerns, and an economy of scale to justify the investment in the construction of the system. This can be achieved either by a very large single-product market, or more likely, by a market for a related set of products — a *product family* — sharing a large fraction of their artifacts (architectures and components), and hence amortizing the aggregate cost across products and time.

The primary obstacle to constructing architectures supporting product families is that architectural decisions usually require broad understanding of a domain, but products must exist prior to the accumulation of this knowledge. This situation exists because the demand for functionality almost always precedes the in-depth analysis necessary to understand the domain — airplane manufacturers want to use computer-based flight control systems before there is a universally accepted theory of how the control system will interoperate with the remainder of the functions on the airplane[3]. The result is that systems usually provide only partial solutions to domain problems. Increasing a software system's coverage of a problem domain requires not only further analysis of the domain, but realization of the increased understanding in software. Furthermore, the less well-understood and refined a system architecture is (i.e., immaturity), the higher the likely cost of making changes[4].

The most significant prototyping effort will be expended in contexts where architectures are immature. If the domain was mature, then analytical methods with a higher degree of coverage would exist. It is therefore the nature of prototyping to be used in less well-understood application domains, where change is more rapid. It is also an implication that prototyping is not likely to result, by itself, in broad problem domain solutions. It is therefore important to realize that prototyping should be focussed at addressing specific issues within a larger problem solving/reasoning activity.

Domain theories are necessary to architect product families since variability is one of the additional attributes of an architecture that megaprogramming requires. It should be noted

[3]They of course will only fly the plane if they are sure the system will do what they want, and will not interfere with other onboard systems, but how it will integrate with the gate scheduler is not known, and hence left to human control.

[4]Note that architecture maturity and system maturity are not equivalent.

```
Ad hoc                          Components    Automatic Code
                                 Based        Generators
----------------------------------------------------------->
```

Figure 1: Domain Maturity

that during the time when prototyping is used most heavily, namely during initial lifecycle phases, we don't expect the domain theory to be represented in a particularly formal language. Rather, domain theory will likely be represented in informal conceptual structures that model the domain. We expect the representation to be primarily structured text, primarily because prototyping will yield results which will only be understood and placed into context (in a megaprogramming sense) after some period of reflection. We strongly believe that information capture is necessary to this activity.

Thus, prototyping should be properly viewed as an experimental process whose results drive the evolution of a domain theory. A difficulty in the state of the practice is that prototyping is often viewed as aiming to answer isolated questions relevant to an individual product – the results of prototyping are not being folded into an evolving domain theory which supports a family of products. The prototyping activity, or experiment, should be as focused as possible to minimize the expense, and to enable accurate analysis of the results when they are obtained. Simply stated, software prototyping is an application of the scientific method to the development of domain theory for software-intensive systems. Following from the scientific method, experiments should only be carried out to refute, or to confirm, stated hypotheses. Repeatability of experiments, to the extent that the resulting analysis can be repeated, (i.e., capturing the environment in which it was performed) is important for future analysis since by definition, the results will only be provisionally valid.

The primary goal of megaprogramming within a given domain is to permit problems in the domain to be resolved with a minimum of programming effort. To achieve this result, megaprogramming must be based on a codified set of artifacts (i.e., domain-specific architectures and components) which are derived from fundamental, common domain principles. Domain specificity places bounds on the context in which these artifacts may be reused, thereby increasing the likelihood that they can be specified in a reusable manner.

The goals of ProtoTech can thus be stated in terms of a domain's "maturity model" (see Figure 1). The maturity of a particular domain is measured by the degree to which there is common understanding of the principles governing the domain, and the degree to which there is general agreement about solutions for broad classes of problems in the domain, and the degree to which these decisions have been codified in reusable software architectures and components.

Within the maturity model, the main prototyping effort will be performed for problems in domains which are closer to the "Ad hoc" end of the maturity scale, as typified by one-of-a-kind or unprecedented implementations. The lessons learned during a prototyping exercise should be useful not only in resolving the current problem (i.e., feeding the results into the main line of development), but should also aid in the movement of the domain theory towards the "Components-based" end of the maturity scale, where the principles of the domain have been codified as a set of architectures and components.

Previously, we stated that the ProtoTech goal is to maximize the effectiveness of prototyping within a megaprogramming software engineering lifecycle. From the maturity model perspective, the goal of prototyping can be restated as accelerating the "rightward" motion in figure 1 per unit of energy (e.g. dollars) spent.

4 Honeywell ProtoTech Strategy

In this section, we provide a historical perspective on this project, describe abstractly how various users of a megaprogramming environment could benefit from prototyping technology, and show how our technology thrusts will aid them. Finally we present our plans for future work.

4.1 Historical Perspective

From the start of our ProtoTech contract, our basic belief was that what distinguished prototyping from "software development" was the process undertaken. Clearly, a prototyping activity satisfied a different set of constraints than a development activity, since prototyping's purpose was information gathering. We assumed that this information would primarily be embodied in a MIF representation of the prototyping "results" garnered from an experimental prototyping platform. The results would be integrated into the mainline software development process through the configuration management engine controlling the project development. In this way, the incremental nature of the results would be apparent, and traceability of prototyping-based decisions would be maintained. We postulated an environment to support this activity, specifically the Environment for Software Prototyping (ESP), shown in figure 2. Prototyping would be carried out concurrently with the development activity, in an "accelerated development" mode, i.e. a prototyping experiment would be carried out fast enough that results could be fed back to the main line of development in a timely manner.

This view was not workable for a number of reasons. First, few prototyping efforts have a clean "question goes in, answer comes back" call/return style interface with the larger development effort. There will be a frequent exchange of data and control between the two. The question furnished to the prototypers must be embedded in a context of previous decisions, discoveries, and constraints, and the answer which is exposed will be part of a network of answers to derived questions and a history of efforts to answer those questions.

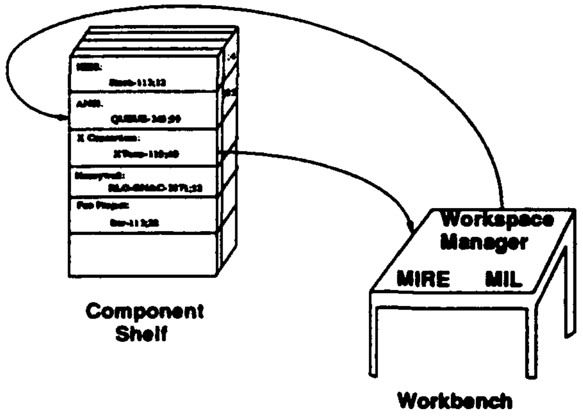

Figure 2: Early ESP conceptual overview

Secondly, viewing the results of an experiment as being communicated by the configuration manager's representation of the software artifacts misses the fact that most of the results must be incorporated into the evolving design, of which only one representation is the collection of software artifacts. A simple record of the evolution of the prototyping apparatus (the prototype) is inadequate, even if it records the transformations from a specification to an executable. A large amount of the knowledge accumulated is in the form of deliberations: questions, problems, proposed solutions, arguments about those solutions, and the application of issue-resolution methods (e.g. prototyping) to choose solutions. Some representation of the issues and their resolutions is necessary to facilitate reasonable traceability between software artifacts, designs, specifications, and requirements, and to communicate questions and results between the prototypers and the developers. Thirdly, although prototyping processes have characteristics which distinguish them from non-prototyping processes — most importantly their goals — and consequently require different sorts of controls, it is no less important that the integrity of the results be maintained. Having process support in place to facilitate the management of the complex issue resolution process appears to be a key point in successfully managing prototyping efforts.

5 Our approach

We have chosen to address prototyping in a megaprogramming environment via four cooperating technology areas:

1. Module Interconnection Formalism (MIF)

   A way of expressing architectural and interface concerns in a highly parameterisable manner, including type parameters, functional parameters, and constraint parameters. The formalism must be both hierarchical and incrementally specifiable.

2. Lab Notebook (LN)

   A mechanism to foster more complete analysis of design decisions, and to record them in a manner that facilitates communication. The LN embodies a disciplined (i.e., structured) approach to the scientific method which organizes information and manages the myriad details that often increase the cost of prototyping and effective information interchange.

3. Process Management (PM)

   A two pronged approach to providing computer assistance during system engineering. Part one is a set of tailorable process definitions which provide a methodology for integrated experimentation, domain analysis, and component/architecture development. Part two is a subsystem which provides assistance, guidance, and measurement to facilitate cooperative work via computer mediated activity coordination.

4. Change and Configuration Management (CCM)

   A substrate technology linking the other three. The MIF interface specifications define configurations. LN provides an abstracted view of changes through a system's lifecycle and aids in change notification. Software process guides system builders through the steps necessary in order to migrate systems from one set of configurations to another.

   CCM is also the basis for doing technology transfer. The most explicit software process, record capture, and artifact management capabilities come from an organizations CCM system. If the technology is to be successfully transmitted to users, it must be integrated into their basic support infrastructure.

These four capabilities, when taken together, provide for the representation of engineering decisions in an executable format (MIF), in a format suitable to human understanding of decision interplay and system evolution (LN/CCM), and provide the necessary active guidance to increase the likelihood that it all will actually be used (PM).

The following sections describe each of the above technologies in somewhat greater detail, point out the relationships between them, and provide references to working documents containing our latest thoughts on the items.

5.1 Module Interconnection Formalisms (MIF)

Two complementary camps have emerged within the ProtoTech Community with respect to MIF research and technology: the architecture representation language (ARL) camp and the configuration packager camp. Those in the ARL camp are focusing on the theory of architectures and design of notations/languages in which software system architectures can be formally described. Those in the configuration packager camp are focusing on technology necessary to automate the translation of fully instantiated architectures into source code. Of course, the line between these two camps is often fuzzy as they are addressing complementary aspects of a more general problem we refer to as MIF in the ProtoTech Community.

The Honeywell/University of Maryland (UMD) Team has members representing both camps. Honeywell is primarily a member of the ARL camp and has been investigating the requirements and use of ARLs with the Honeywell and IBM DSSA teams and David Garlan of CMU.

UMD, as founding members of the configuration packager camp, have been developing MIF technology that will provide a basis for implementation of ARL-based design/development tool sets. The UMD software bus technology developed to date has demonstrated the viability of the module interconnection language and configuration packaging concepts and supporting run-time system[18].

5.1.1 Approach

Our Perspective of the MIF Problem A generally anemic aspect of software engineering is that of software architecture; the art/science of planning and building software systems. General principles and theory of software architecture simply do not exist. Lacking formal architectural principles, methods and tool support, we build software systems based on ad hoc and implicit architectures. The results are as expected.

An architecture is a representation of a set of decisions which purport to solve a set of issues/constraints raised by a system construction which leaves, to the largest extent possible, other issues open for later resolution with a minimum of additional constraints. The set of resolved issues and the constraints imposed by the architecture are a critical component of the architecture.

A software architecture is a hierarchical structure representing levels of design decisions which organize the set of software components that comprise the system and expresses constraints on the components and interactions between them.

We observe the following symptoms in Honeywell's software development businesses and attribute these to the lack of a formal basis for software architectures:

- Generic CASE architecture tools are applied to the requirements and design phase for describing software architectures but the results are typically under whelming; these

CASE tools have been abandoned by some in favor of generic graphics tools due to their inflexibility, lack of expressiveness or inappropriateness with respect to the tasks at hand.

In particular, the lack of domain specificity has been cited by divisional software engineers as a reason not to use tools such as STP[5], Teamwork[6], etc. That is, they don't map cleanly onto the problem domain and the gap between a CASE-specified design and the eventual software architecture is too great.

- Regardless of how software blueprints have been rendered (i.e., back-of-the-envelope sketches verses more formal software architecture drawings/descriptions produced using commercial CASE tools), the translation of these software architecture descriptions to software designs is largely viewed as a black art. Software engineers do not share a common understanding of the process of translating software blueprints into designs. The low fidelity of the translation process is evidenced by multiple, inconsistent and often incorrect translations of the same architectural concepts even within the same system.

- The mapping between the software blueprint and other representations of the system (requirements, detailed design, source code, regression tests, etc.) deteriorates over time. The source code itself becomes the only authoritative document of the system's architecture as the system evolves. Any existing software blueprints act merely as hints as to what one might expect to find in the source code for the system. Therefore the software blueprint is rarely consulted nor kept up to date over time lending to further deterioration of the mapping between this and other representations of the system.

  And even when blueprints are kept up to date, they contain only information about the end product itself; nothing about why the design is the way it is as opposed to other ways.

The problems of architecture representation, interpretation and maintenance must be addressed and the impact of solutions to these problems may be profound. Point solutions, focusing on pieces of these problems are already being developed by various groups within Honeywell. Application generation technology is being applied within the displays domain of several Honeywell avionics divisions. A group within the Air Transport Systems Division developing avionics simulation software is developing code generation capabilities for a commercial CASE tool used for specifying software architectures.

The DARPA Domain Specific Software Architecture (DSSA) Community is also working on solutions to these problem; each DSSA team is responsible for developing architectures for their respective application domains. [7]

[5]STP is a registered trademark of IDE

[6]Teamwork is a registered trademark of CADRE

[7]The DSSA Infrastructure Tiger Team has developed a four-layer model of architecture concerns: architecture schema, architecture representation language, the reference architecture and product architecture.

Our Approach to the Software Architecture Problem Our approach to solving this problem involves the design of an architecture representation language (ARL) based on a formal theory for software architectures, the development of an effective engineering methodology based on ARLs, and a minimal tool set which supports the ARL-based methodology.

An ARL is a language for describing architectural elements: components, component-to-component connections, component subarchitectures (i.e., hierarchical composition relationships) and constraints on these objects.[8]

The design of ARLs is a collaborative effort currently in progress. We are working with members of the DSSA community to develop requirements for ARLs. Based on the requirements derived from this effort, we will be either adopting or designing an appropriate ARL. It is not clear at this time as to whether a single ARL can be designed or exists which satisfies the entire set of requirements. Current wisdom suggests that multiple ARLs will be required and that existing specification languages must be extended[5, 7].

An ARL-based engineering methodology is being developed by the Honeywell ProtoTech Team. There are several related projects at Honeywell SRC from which elements of this methodology are being derived; in particular, the Honeywell DSSA project and an internal software reuse initiative focused on the domain of avionics flight management systems. Both projects focus heavily on software architecture issues and suggest the need for a formal methodological basis[2].

A collection of integrated tools will be required to support the ARL-based engineering methodology. The tool set must minimally provide the following capabilities:

- Generate/synthesize systems from architecture representations. The architecture will contain unbound parameters for which the user must supply arguments to generate a system.
- Support architecture analysis (i.e., architectures as formal, machine processable entities). Architecture analysis tools apply analytic techniques to architecture representations.
- Support reuse of architecture objects (i.e., support for persistent objects in the ARL).
- Support user-definable architecture objects

It is our goal to prototype a system with these capabilities and test the effectiveness of the supported methodology. Much of technology required to build this prototype already exists in the DARPA community (e.g., persistent object base and graphical user interface technology) allowing us to focus primarily on ARL design issues.

[8]The architectural elements described here were identified by the DSSA ARL Tiger Team. This group is currently analysing requirements for ARLs.

Having accomplished this goal, we can then turn to establishing integration requirements for MIF technology and tools with the other elements of our prototyping environment, namely the lab notebook, process management and change and configuration management systems.

5.1.2 Future Plans

Our current plans for MIF research include the following action items:

- Publish a summary of DSSA ARL Tiger Team ARL Requirements Analysis to the DSSA and ProtoTech Communities.
- Continue research on ARLs. We will be leveraging the efforts of other researchers in the DARPA Community showing progress on this research topic.
- Develop and refine our ARL-based software engineering methodology.
- Prototype and demonstrate tools supporting the ARL-based software engineering methodology. The prototype demonstration will:
    1. Test the prototype tools internally on toy prototyping problems.
    2. Show the applicability of the prototype to the Honeywell DSSA system, replacing their current approach for representing architectures.
    3. Identify a divisional project in which the MIF technology may be further tested and refined.
- Continue the MIF dialogue with the MIF Working Group of the ProtoTech Community.

5.2 Lab Notebook

The ProtoTech community's characterization of prototyping as "an experimental activity intended to expose properties of a potential product before design decisions are carved in stone" has two key aspects. First, prototyping is an experimental method for *learning* in the absence of more formal analytic methods. Second, since the results of a prototyping effort typically drive future design decisions, these results must be effectively *communicated* to those who will use the information.

The ESP Lab Notebook (LN) supports the learning and communication activities involved in prototyping. In particular, the LN provides two main functions. First, it captures a design rationale which records the artifacts produced by a prototyping effort (i.e., *what*) and the deliberative process which produced those artifacts (i.e., *why*). Second, it provides a means of viewing that rationale. Increasingly, evidence suggests that knowledge of why a system does something contributes more to understanding than knowledge of what a system does, and that deep understanding requires knowledge of both [22, 12]. A LN must manage the following complexities:

- Since prototyping is often carried out by a group, the LN must facilitate the learning activity in a group setting.
- Learning during prototyping (both in group and individual settings) emerges from a *series of deliberative processes during which positions* are taken, methods for gathering data are selected, data is interpreted, and decision criteria are selected.
- Deliberative processes give rise to extremely complex webs of discourse which obey regular structures (e.g., [11, 14, 3, 20]).
- Learning may be facilitated by managing and preserving these deliberative discourse structures (e.g., identifying when a topic has been left unresolved, when arguments are being repeated, etc.)
- Communication may be facilitated by transferring these deliberative discourse structures, perhaps edited for different needs, to later users of the information. The structure itself provides a means for navigating within the information, while the information captured during design deliberation may be expected to be much richer than that captured after the fact (as in traditional technical reports).
- To effectively communicate design rationale, the LN must support multiple views at various levels of abstraction (e.g., [3] describes what happens without such a capability).
- The LN must strike a careful balance between human usability, machine usability, and expressiveness, all of which are interrelated [10].

- Above all, the LN system must get used if it is to fulfill its functions. This means that it must provide some utility and value added both to the extended design team (if it is to communicate), and to the local prototyping team (if it is facilitate learning and capture design rationale in the first place).

5.2.1 Activities

To date, LN activities have primarily focused on a literature search to identify related work. We have recently begun to produce a concepts document which will more fully describe a LN capability within the context of ESP.

5.2.2 Approach

Prototyping as Deliberation Prototyping, and system design in general, is inherently a deliberative process. Issues are raised, alternatives are defined, lines of reasoning are formed and debated, and ultimately, decisions are made. Experiments or trials are run and data is collected, but even this process is fraught with debate about the appropriate questions to be asked, the appropriate methods for testing hypotheses, the interpretation of results and how results should be integrated into the overall design. Such a process provides an effective means of learning about the problem domain and the solution space, and for communicating results. Since the deliberative process is the *means* by which decisions are made, a record of it shows how the decision was arrived at.

Deliberative processes of this nature obey a loose, but nevertheless real structure known as a *discourse grammar* [19, 13]. Just like the more familiar verbal grammars, any given speaker may violate grammar rules (either intentionally or unintentionally) with any given utterance, but communication is facilitated by proper use of the grammar. A variety of researchers [3, 20, 15] have proposed models of deliberative grammar structures which, we believe, can be used to structure a tool for capturing the deliberations which occur as a part of the prototyping process.

There are at least five benefits to providing a tool for structuring and recording deliberative processes and the design rationale which results from them:

- A structured deliberative process helps identify incompleteness or inconsistency in a line of reasoning (i.e., gaps, missing arguments in the grammar structure).
- A structured deliberative process helps focus discussions on relevant issues and helps expose fallacious arguments (i.e., by identifying issues over which there is dissension and arguments which are logically inconsistent).
- A structured deliberative process is likely to both be and to appear to be more "fair" to all participants. Everyone's issues are recorded and addressed. Reasons for decisions are explicit and retrievable. Records are kept of decisions and thus, memory for

group decisions is less subject to individual interpretation. All of these factors should contribute to greater group satisfaction with the decision process and, hence greater efficiency.

- Design rationale captured during the deliberative process serves as a group memory which facilitates recovery of previous discussions and the context in which they were held. It also enables users to determine why a decision was made without replaying the discussion.
- Design rationale is useful (and necessary) across the entire project lifecycle and helps users understand both why a system is constructed a particular way and why it is not constructed another way.

In short, a structured deliberative process leads to a quicker, more complete, and less error prone understanding of a problem domain and potential solutions and a design rationale serves as an effective vehicle for communicating results.

LN as a Structured Deliberation Support Tool We, therefore, conceptualize LN as a tool which supports and captures deliberations which occur during the prototyping process. LN will provide and encourage the use of a deliberative structure. By so doing, it will improve the quality of the deliberative process. LN will capture the process of a particular deliberation (or series of deliberations) during prototyping by instantiating a deliberative structure. This instantiated structure thus becomes a record of the deliberative process. LN will enable inspection of the instantiated deliberative structure, thereby improving communication between groups and "memory" (i.e., retrieval of information) within a group. We will also explore methods of translating or summarizing information captured during the deliberation process for transfer to other users, as well as methods for using the structure itself to manage and critique the deliberative process.

General Characteristics We have identified six dimensions which characterize all existing and proposed design record capture tools. These are not orthogonal dimensions, but are related in complex ways which we are beginning to understand. We believe that any LN tool will be characterized by a set of tradeoffs along these dimensions and our future work will be directed at identifying the optimal set of tradeoffs for utility in the ProtoTech environment.

- **Structure** — The degree to which interactions with the LN must obey a pre-determined format. There are two different sources of structure: discourse structure and vocabulary structure.

  Discourse structure refers to control over when and (generally) what information is input, but not necessarily over how it is input. Systems which use a question and answer format are generally imposing some sort of discourse structure, but those that permit answers to questions in free text are generally imposing only discourse structure and not vocabulary structure. An important aspect of discourse structure is the explicit representation of goals, assumptions, constraints and their interrelationships which serve to describe expectations about the context in which a prototype is developed [14]. They also serve to prune the solution space.

Vocabulary structure refers to control over how information is input (i.e., control over the vocabulary of input). Systems which use a feature set language to input new information strictly control "language" structure. Vocabulary structure is important in preserving a common frame of reference.

Generally, increased structure facilitates the input of information by 1) constraining the space of needed inputs to a manageable set, 2) implementing a framework within which input is expected, and 3) making inputs incremental (rather than writing the final report at the end of the project). It also facilitates retrieval of information by imposing a navigable framework through which information can be traced later. However, structure can also interfere with design by imposing restrictive levels of order too soon on the sometimes nebulous creative process, or by limiting the vocabulary with which ideas can be discussed, thereby increasing the difficulty of recording of early, ill-formed thoughts.

- Domain knowledge — The degree to which the LN uses domain-specific knowledge in order to perform its functions. We refer here to knowledge about the domain of design, rather than knowledge about deliberative interactions, or knowledge about design rationale capture. In a sense, structure (as defined above — particularly discourse structure) may be regarded as the syntax of an interaction in which design rationale is captured, while domain knowledge provides the semantics for the interaction. A feature language vocabulary structure would require domain knowledge.

  The presence of adequate structured domain knowledge enables a tool to be "smarter" in interactions in that domain. Both discourse and vocabulary structures can be sensitive to semantic aspects of the domain, rather than to the simple syntax of interaction. Critiquing (see below) can be done on semantic rather than syntactic levels as well. Incorporation of new knowledge (see below) can be done in a more sophisticated fashion, since a knowledge organization already exists. However, domain knowledge has huge overhead associated with it. It is difficult to capture, represent, manage and utilize. Further, particularly for prototyping endeavors where the problem is poorly understood initially, relevant domain knowledge may be unavailable.

- Critique — The degree to which the LN can offer evaluations of the information provided by a user. Critiquing can be done on both syntactic (e.g., structural) and semantic (e.g., domain knowledge) levels. Clearly, the kinds of critique the LN can provide depends of the available structure and domain knowledge.

  Critiquing can provide a source of value added for LN users (and hence an incentive for using the system), and can lead to better design and more accurate and complete record capture overall. Critiquing is arguably the best way to go about incorporating new domain knowledge into the knowledge base of the tool as the LN asks questions to flesh out its own knowledge base (see below). However, critiquing also incurs the overhead of additional structure and domain knowledge, in addition to the overhead of designing and constructing the critiquing functions themselves. In addition, unsolicited critiquing is often intrusive and can disrupt ongoing thought processes (see below) thereby stifling the creative process. Further, incorrect or inappropriate critiquing is annoying, and critiquing without incorporation (i.e., learning), where the tool is wrong and stays wrong, is especially annoying. Users often avoid or ignore annoying criticism and may miss valuable feedback as a result.

- **Intrusiveness** — The degree to which the LN intrudes on the design process and the thoughts of users as opposed to being a passive observer or off-line receptacle. A traditional electronic lab notebook (i.e., an electronic medium for recording ad-hoc notes, etc.) is the ultimate in non-intrusiveness. Any type of structured dialog (e.g., Q&A) is necessarily intrusive to some degree, but the dialog phase itself can be relegated to a specific time and place (e.g., enter the LN and sit through a Q&A session) or can be ongoing and omnipresent (e.g., an interrogator subroutine running on a CAD design tool). Critiquing is also necessarily intrusive, but again, it can be request-driven as opposed to unsolicited.

  Intrusion probably obeys some inverted "U-shaped" function with regards to amount of information captured (i.e., there is a point of diminishing returns). No intrusion is more or less the way documentation and reports are written now — wait until the end of the project and put down whatever you can remember as quickly and painlessly as the customer and documentation tool will allow. Up to a point, a system which frequently asks for information is more likely to get it than one that doesn't and a system that asks for information at or near the time the information is being used/considered is more likely to get more of that information than one which waits until the time is past. However, the act of intruding on thought processes is inherently disruptive — and therefore annoying. Ideas may be lost as the result of ill-timed questions and enough disruptiveness may cause users to avoid the system. Of course, the ability to actively disrupt thought processes means the system must be present and "conscious" during those processes (i.e., it must be "looking over the shoulder" of the designers). This implies a much more complicated tool, with associated overhead. The only practical application of this technique to date seems to be to make the tool a part of another tool which is itself a part of the design process (e.g., a CAD tool, or an authoring tool).

- **Incorporation** — The degree to which the LN learns or redefines itself on the basis of the information it captures during a design instance. Primarily, incorporation deals with learning new domain knowledge, but the LN could learn structural knowledge as well, maybe on a individual differences basis. For example, the fact that a particular person is prone to bad counterfactual reasoning constructions may guide the type of advice or critique provided by the LN. Individual differences might also be important for more semantic knowledge — knowing what side a person takes on an issue is certainly important to how listeners evaluate his/her reporting of facts. Various ways a system may "become smarter" include the growing of a design decision tree which, minimally, helps to position later information, learning new domain knowledge for later knowledge-based critiquing, and learning new keywords and/or links between topics for later cross-referencing.

Incorporation is essentially a learning task and, therefore, many LN approaches using incorporation would suffer the traditional machine learning problems. It is generally difficult for a system to automatically determine the "correct" new information to incorporate. In human-assisted learning, users frequently will not take the time to "teach" a novice knowledge base what it needs to know. In either case, determining the effects of new information on existing knowledge structures is difficult. Without good information, the system will suffer from the problems described above (e.g., bad advice, etc.). In contrast, some types of incorporation would be reasonable straightforward. In a vocabulary-structured LN, for example, enabling the system to recognize and request definitions for new terms would be reasonably easy. By the same token, automatically incorporating cross reference links based on key words would be a trivial exercise in database management.

- **Integration** — The degree to which the LN is integrated across the development team and development process. To be maximally useful, the various artifacts captured in the LN should be interrelated. In particular, the software artifacts should be tied to relevant portions of the design rationale so that users have a direct link between an artifact and information that describes it. Without direct linkage, users will be less likely to lookup relevant information and the rationale will not effectively serve the purpose of communication.

  However, integration implies an information management overhead which, if carried too far, can become too costly. The need for integration must be balanced against the cost [6].

5.2.3 Future Plans

Future plans include the following:

- Finish up a concepts document that develops and refines the ideas presented here.
- Write an initial requirements document for the ESP LN that addresses the six dimensions described above.
- Spec out a LN capability and prototype selected features to learn about necessary features and the LN's ties to ESP and the prototyping process.

5.3 Change and Configuration Management

Change and Configuration Management (CCM) is the collection of tools, techniques, and facilities for managing change to complex systems of interdependent artifacts. CCM must cope with several complexities:

- The artifacts are both internally complex and highly interdependent. The trend is for the artifacts to be finer-grained, which will increase the complexity of the web of relationships and thus the interdependence.
- The history of such an artifact is typically not a linear line of descent, but a DAG, with branches and merges.
- It is typically necessary to maintain more than just the "most current" version of an artifact. Non-leaf versions are frequently still in active use; also it is often necessary to backtrack to an interior version.
- Frequently, multiple threads of development are active simultaneously — multiple alternates are undergoing evolution.
- It is useful to be able to specify a composite artifact at multiple levels of concreteness, allowing the versions of components to vary in specified ways — all components may be specific versions of artifacts, or some may be version-generic references which may specify a rule for selecting an appropriate version of that artifact.
- Change is typically performed by multiple agents, working in teams and subteams; a logical change consists of a number of physical changes, and logical changes may be nested.

Prototyping presents specific challenges to CCM; agile, lightweight CCM will be a key support capability for prototyping. Prototypes evolve in complex ways. Due to the need to quickly explore the relevant problem domain and solution space, prototypes must evolve rapidly and radically. Even more so than "normal" software development, prototyping is an iterative and tentative activity. Frequently a thread of development is abandoned when the line of investigation fails, we backtrack to a previous version, and continue on in a different direction.

Similarly, normal software development frequently has multiple alternate threads of development ongoing simultaneously; this is even more significant in prototyping. Evolutionary prototyping effectively presents the challenge of multiple overlapping interdependent life-cycles — for instance, one version of the prototype may be undergoing experimentation, while a second is about to be fielded for the next round of experimentation, while a third is undergoing development. Multiple threads of development is also a significant feature of prototyping because it is often important to carry several designs forward - to explore development of several possible designs (in actual or simulated concurrency), rather than freezing design decisions too early.

5.3.1 Activities

- Based on previous work done at S&RC, we have produced some reference documents.
- In an attempt to leverage activities with the POB project, we participated in a OOODB sponsored CM workshop, and provided inputs on the CM model that they proposed. Unfortunately, there has been no followup activity on OOODB CM, and no significant progress to date.
- As an alternative, we have been attempting to get a copy of the Artifacts system from software options. Scheduling and release problems have resulted in some delays, but we are hopeful that we will be getting a copy soon.
- We have been investigating the possibility of using a layered file system strategy to incorporate object oriented file system capabilities into PM tools to facilitate process enactment with a minimal impact on existing user environments.

5.3.2 Approach

Although prototyping is tightly related to the larger effort which it supports (see Section 5.4), a process with significant experimental intent needs very different change-management policies from a pure production-intent process. The production-intent process, which is preparing a system for delivery to a customer, may have stringent requirements for okaying changes, involving input from all the stakeholders; it may have rigorous qualification requirements for promoting a version, etc. In a high experimental intent prototyping process, the decision to make a change will typically be much lighter-weight, in the hands of the prototypers, and guided by the deliberative nature of the process. The prototyping process must have the appropriate disciplines for capturing the knowledge developed during prototyping. The capture of the total evolution of engineering decisions is very significant in a question-answering method like prototyping. Classical design record requirements (e.g., 2167a) tend to assume pure production-intent processes (which may never exist); they specify that the design decisions at each level of abstraction must be recorded, but the tree of false starts, unapplied discoveries, and fine-grained rationales is not captured

Our approach is to assume the existence of a highly-configurable, lightweight CCM engine, which cooperates strongly with a process management (PM) engine. The CCM engine handles the complexities listed above. It captures the history of evolution of systems of artifacts in a recoverable way, provides the building blocks for nestable long transactions, and provides mechanisms for version-selection references using compatibility relationships. The PM engine provides guidance, monitoring, assistance, and metrics-collection. The process definition specify the appropriate CCM policies, which are implemented by the CCM and PM engines.

The CCM engine will rely on these PM engine capabilities:

- Mechanisms to implement various flavors of change management policies and procedures.
- Mechanisms to measure/monitor change processes and changing artifacts.
- Reified process state which provides knowledge of the intent of change events, to guide change propagation.

Contrariwise, the PM engine needs these capabilities from the CCM engine:

- The history of artifact evolution — status of artifacts, changes applied to them.
- Constructs for performing, manipulating, and reasoning about changes.
  - Constructs for controlled cooperation among individuals and teams — long transactions, workspaces, etc.
  - A open change propagation interface for implementing policy.
  - Mechanisms for evolving process definitions (artifact, agent, and activity templates).
  - Constructs for dealing with evolution, concepts for dealing with change. To reify process, one must reify change/evolution/history — one need terms representing change (change as a first-class object).

We hope to acquire most of such a CCM engine, either COTS (commercial off-the-shelf) or ROTS (research off-the-shelf), and use it in our experiments. Our definitions of prototyping processes will include specifications of appropriate CCM policies.

5.3.3 Future Plans

We hope to acquire either a pre-release of the OOODB POB, or a copy of Artifacts to enable more realistic experimentation with process enactment capabilities on top of a UNIX platform. Our goal is to provide a tech transfer vehicle for our other technologies by either sliding underneath existing file system support, or integrating with existing CCM system capabilities. We do not wish to expend large amounts of energy developing a full CCM system ourselves, but we are finding it very time consuming to acquire the necessary support capabilities for or PM and LN work. We have not yet decided on a plan of action to resolve this situation.

5.4 Software Process Management

Process Management (PM) is the support and improvement of software engineering processes through process definition, the use of a defined process to guide execution/enactment, automated support for process monitoring and measuring, automated support for process assistance/guidance, and the analysis of process metrics for project management and process improvement.

To support prototyping in a megaprogramming environment, we believe a two-pronged PM approach is needed. Part one is a set of tailorable prototyping process definitions which provide methodologies for integrated experimentation, domain analysis, and architecture development. Part two is an environment subsystem which provides assistance, guidance, and metrics-collection to facilitate and coordinate cooperative work according to those prototyping methodologies.

Prototyping is distinguished as a unique flavor of software development by its process. One cannot tell whether someone is prototyping by observing what tools he/she is using; rather, one must look at the process – in particular, the goals of that process. In prototyping processes, a significant goal driving the development and exercising of the software is the need to answer specific questions (experimental intent). We believe that the unique needs of prototyping will not be satisfied by new tools but rather by process improvements (which should be supported by tools!).

We have studied examples of prototyping within Honeywell, and available studies from other companies. In particular we have focused on "prototyping-in-the-large", where prototyping is used in the development of large long-lived real-world systems, typically involving multiple teams. We have observed several recurring problems faced by prototyping-in-the-large, which are candidates for process improvements:

Prototyping must be reasonably cheap and fast. As previously stated, development of a prototype is not an end in itself, Prototyping efforts are elements of other efforts, and have tight constraints on budgets and calendar time. Prototyping must occur quickly, to find the needed answers before unacceptably delaying the larger effort. Prototyping efforts are frequently difficult to scope, because they operate in areas with many unknowns, which lack analytical question-answering methods. Further, ongoing prototyping tasks may lose resources which get redirected to fight other fires in the larger effort. Prototypes typically deal with situations where there are many variables, but the state of the practice is that careful experimental design is underused in software prototyping.

Prototypes evolve in complex ways. As previously stated (Section 5.3), prototypes evolve rapidly and radically, with frequent backtracking, and with multiple threads of development active concurrently.

The behavior of software systems is often too complex to analyse except through empirical means. However, some organizations shun prototyping due to the difficulty of scoping and managing prototyping efforts, preferring to shoulder the risk of the unknown design parameters that could have been revealed using prototyping techniques. One curious effect of the difficulty of scoping and managing prototyping efforts is the existence of "clandestine prototyping". Some organizations actually forbid prototyping because it cannot be accurately costed, and because their customer-mandated CCM system is so ponderous that prototyping is simply infeasible; changes cannot be performed rapidly – even changes to prototypes – since all changes must go through the slow CCM system. In such organizations, clandestine prototyping occurs – prototyping is used, but under another name, without the support of tooling or process.

The question may be poorly posed. Frequently the question which is to be answered by a prototyping effort is poorly posed. This is particularly a problem when the team posing the question (the development team) is distinct from the team answering it (the prototyping team). Often specific sorts of information are desired, but inadequate effort is spent expressing these as the goal of the prototyping effort; the goal is frequently expressed in terms of feasibility, leading to a million-dollar prototyping effort which emits one bit of data as its result: "yes, it is feasible", "no, it isn't".

The question statement may be too vague. It may not specify what kinds of information the posers actually desire as answers. It may lack success criteria, so that it impossible to judge when the effort is done and the answer should be used. It is often necessary to evolve the question statement based on input from the prototypers (and their initial exploration of the question), until it is properly posed. Poorly-posed questions frequently result in answers that are difficult to integrate into the larger effort and the product family's domain model.

The original question may be misplaced. Prototyping efforts are frequently working in less well-understood problem areas, where there are many unknowns, and the answer to one question depends on answering a raft of other questions. The state of the practice is that prototyping efforts often get off-track, concentrating on answering a derived or related question rather than the original question which prompted the effort.

Choosing which derived questions to pursue. Answering one question depends on answering others; further, in the process of answering a question one may discover other, higher-priority questions previously unconsidered. Further, as noted above, the specification of questions is often an iterative process – a question must be refined or reposed to more accurately express the need. Thus a prototyping effort typically deals with not one question but rather a growing network of questions that evolve over time; typically there are more questions than there are resources to answer them. Thus, the evolving network of questions and efforts to answer certain of them must be carefully and dynamically managed. Resources (people, budget, calendar time, hardware, etc) must be allocated to the most pressing questions. Priorities will change over time, as new questions are discovered and constraints inherited from the larger effort change; thus resources must be dynamically re-allocated. Prototyping processes thus must be carefully guided, and must react nimbly to changing circumstances.

Associated knowledge may be lost. A prototyping effort frequently develops a considerable body of relevant knowledge over and above the answer to the original question, in the process of answering that question. The state of the practice is that this associated knowledge is often lost, and effort must be spent later to rediscover it. The knowledge may be forgotten before the larger effort reaches the phase where that knowledge is applicable; the prototypers and the developers may be different teams, in which case the fact that that knowledge once existed may not be known to the developers; the knowledge may be recorded, but in a form that is so difficult to locate that the developers do a rediscovery effort anyway.

Resulting knowledge may be inadequately integrated back. At the end of one iteration of the canonical scientific method, the experimental results are related back to the theory/model which prompted the hypothesis; by denying or tending to confirm the hypothesis, the results enrich and extend the theory. This step is typically inadequately performed in software prototyping, particularly prototyping-in-the-large. The results should be fed into a domain analysis activity which uses them to enrich the evolving domain model. But the state of the practice is that the results are frequently seen as relevant to only one issue in only one product; a decision is made on that issue, the rationale is forgotten, and its relevance to the product family is not explored and recorded.

The knowledge loss which often occurs due to these problems is particularly troubling in a megaprogramming context, where the knowledge resulting from prototyping should be contributing to an expanding domain model.

To address these problems, we are postulating prototyping process models (supported by tooling) that are explicitly deliberative and that leverage MIF's ARL capabilities to incrementally fix design decisions. This is described below.

5.4.1 Activities

In addition to foundational investigations of process description, enactment, and process improvement, we also are involved in a number of other activities;

- We are working with the Boeing STARS effort to transfer process definition and enactment technology. A report of the work can be found in [8]. In addition, we plan to use this technology in our later prototyping process experiments.
- We have been active participants in the ProtoTech community Impact and Process Working Groups. In particular, we have participated in the efforts to explore the space of prototyping processes, since an understanding of prototyping processes is fundamental to our strategy of improving the process of prototyping.

- We have developed an "IBIS mode" utility under GNU Emacs; we have used it in two tasks in a related contract to exercise our model of deliberation-driven prototyping processes. The IBIS mode is used in an issue-resolution activity, which is the master activity driving other prototyping activities which coroutine with it. We will be producing a short paper describing the results of these experiments.
- We have been fostering Honeywell divisional activities to see how process technology can be transferred to divisional sites. We have been working with divisional CCM and Software Process groups, and have started ongoing discussions both face-to-face and via email with two other operating divisions.
- Finally, we have been investigating the difficulties involved in getting organizations to adopt process technology by instigating a set of strategic vision discussions within our own research group at S&RC (approximately 40 engineers). It is our perception that without a focused, process-oriented, strategic vision, organizations do not have the understanding nor motivation to adopt process technology. Since our role within Honeywell is largely to do prototyping-in-the-large, we have a testbed for real-world experimentation with the development and attempted adoption of our proposed prototyping processes. We have decided to attempt this locally first, to see what obstacles are likely to arise before attempting it in one of our operating divisions.
- We have been carrying out several small-scale process enactment experiments on UNIX, and atop DEC COHESION on VMS.

5.4.2 Approach

We believe that the recurring problems of prototyping-in-the-large can best be answered by prototyping processes that are explicitly deliberative and that leverage MIF's ARL capabilities to incrementally fix design decisions. Few process models for software development recognize its deliberative nature; only in the last few years is it becoming more common to admit that there is a degree of experimental intent in most software engineering processes. The long recognition of production intent has led to process components aimed at ensuring that the "ilities" required by production intent are produced – reliability, usability, etc. The recognition of experimental intent present (in varying degrees) in software engineering processes must lead to process components which are aimed at ensuring the well-structuredness of the experiments, the capture of the results, and the integration of the results to enrich the domain model.

The management and guidance of a process with significant experimental intent will leverage the reified deliberation structures. The process will be guided – at least in part – in terms of questions being answered and derived questions being developed and chosen for investigation. This contrasts with a purely production-intent process, where guidance is be in terms of the requirements placed on the system under construction (the prototype) by the final delivery context. An evolutionary prototyping process combines both sets of concerns.

In most existing software process models, the core artifacts are the specifications of the software product – requirements, design, code, test plans, etc. But the evolution of these artifacts occurs embedded in a web of deliberations; prototyping is used as one question-answering (issue-resolving) method in these deliberations. Reifying the deliberations provides the appropriate handles for monitoring and managing the prototyping process, and relates the development of individual products to the evolving domain theory of the product family.

In our vision, definitions of deliberative prototyping processes will provide the guidance for properly conducting prototyping experimentation and for integrating the resulting knowledge into the product family's domain model. The structures and tools of ESP will be fundamental components of these processes:

- MIF's component packaging capabilities will provide the breadboard for ESP, meeting the need for prototyping to be reasonably cheap and fast by moving toward a megaprogramming paradigm, where prototypes are constructed by assembling components rather than by writing code.

- MIF's ARL capabilities will provide the ability to incremental fix parts of a design – the architecture is used to record commitments, to capture the resolution of issues / answering of questions. The relationships between partially-determined versions of a design and the network of deliberations records the rationale for the architectural decisions made, and supports later evolution.

- The Lab Notebook will provide the structures to reify the deliberative process as a structured network of semi-formal text. As previously mentioned, we view prototyping as a question-answering activity; in ESP the Lab Notebook will provide the deliberative structures that support question refinement, development of an answer strategy, recording of an answer derivation (including alternatives considered and rejected), answer interpretation, and answer refinement.

- A sophisticated underlying CCM engine – providing highly configurable, agile, lightweight CCM – will support the rapid and radical evolution of prototypes, their multiple lines of development (carrying multiple designs forward), and the frequent backtracking. The CCM engine likewise will support the growing, evolving graph of deliberations which is tightly interwoven with the graph which is the evolution of the prototype. Agile CCM is particularly necessary for evolutionary prototyping-in-the-large, with it multiple interdependent overlapping lifecycles.

- A sophisticated underlying process management (PM) engine will provide guidance, monitoring, assistance, and metrics-capture as the processes are carried out. The structure of the LN directly reflects a process for question answering; process management guides these deliberations, via deliberation-driven prototyping process definitions informing a PM engine. PM is particularly important at the meta-control level, for switching between interrogative modes. Thus both simple guidance (based on defined processes for experimentation and deliberation) is helpful, as well as reactive guidance based on monitoring and measurement of the prototyping process.

Megaprogramming requires consistent and long term investment in a particular domain. Without adequate supporting information, it is difficult to maintain funding support, or to improve the development process. The continued application of prototyping to megaprogramming will require both these metrics, and the traceability support from CCM. When domains are on the "ad hoc" side of the domain maturity spectrum, there will be a larger number of unknowns, and consequently a much larger number of variables in the problem solving process. Tools will be needed to track the large number of open issues, and aid in systematic resolution procedures. Codification of this procedure should permit leveraged application of similar problem solving mechanisms to different domains.

Given the experimental nature of the proposed problem-resolution strategy, it is important to make sure that the experimental artifacts and apparatus is consistently recorded so that the necessary metrics can be acquired. When the problem resolution methods are appropriately codified, the "process specification" will be the basis for automation and assistance.

PM and CCM are critical support technologies to the long term success of megaprogramming in general, as well as to the effective use of prototyping as a question-answering method within megaprogramming.

We have developed a specific characterization of the ways prototyping processes could be used within a megaprogramming-based development. We have been discussing these ideas both internally, and with the ProtoTech Impact working group. The results are encouraging, but preliminary. A sketch of our current notes can be found in Appendix A.

5.4.3 Future Plans

Future plans include the following:

- Produce a document that records initial insights from use of deliberation-based prototyping on the two tasks in another contract. Continue to experiment with deliberation-based prototyping process models on SRC projects. (It will probably be necessary to move beyond the quick-and-dirty IBIS mode to more powerful support for deliberation, possibly COTS or ROTS; the deliberation facility must be integrated in some way with the PM engine being produced for STARS.)
- Continue interaction with divisional Software Process and CCM people, to validate our insights against their experiences.

- Continue exploration of the space of prototyping processes in the Impact Working Group. Continue to mutually share insights about process support infrastructure with the Process Working Group.

5.5 Relationships

5.5.1 Megaprogramming and ...

... MIF From one perspective, megaprogramming is simply a new word for an old idea, software reuse. However, software reuse has mostly failed to yield the expected benefits forcing software engineering practitioners and theoreticians to rethink software reuse issues. From this perspective, megaprogramming can be viewed as a refinement of software reuse: a broad class of reuse methodologies based on techniques that facilitate programming at the software component level rather than programming language statement level.

Megaprogramming requires a software engineering paradigm shift. Historically, software engineering has largely been viewed as a task of specifying a system's behavior, a line at a time using a programming language. Megaprogramming requires a systems engineering approach, in which systems engineers (megaprogrammers) specify a system's structure in architectural terms and its behavior in terms of component selection rules. Software engineering, in the megaprogramming paradigm, can be viewed as the application of software engineering principles to construction of reusable software components, as opposed to just software systems.

The result of the paradigm shift to megaprogramming is that both software architectures and components become reusable patterns in the development of software systems. The elements of a software architecture model define standard infrastructure for interconnecting and composing components and standard component interfaces. Software architectures become first class objects that can be reasoned about and analyzed separately from the actual components that comprise a system. As explicit objects, software architectures form a natural framework for automatically generating and synthesizing a system from a set of software components.

The current wisdom on software reuse suggests that reuse methods are most successful when applied to specific or narrow domains. For megaprogramming, this implies that the notion of software architectures must also be domain specific. The DARPA DSSA program reflects this understanding. While the elements of an architecture model may be unique per application domain, there exist underlying concepts that are common to all software architecture models. ProtoTech MIF research addresses the common elements of domain specific architecture models.

... LN To a large degree, effective megaprogramming relies on a deep understanding of the principles underlying a given domain and the codification of those principles in an appropriate form (i.e., a domain model). Domain analysis is an evolutionary process of identifying and organizing a domain model for a particular domain with the purpose of making that information reusable. The results of prototyping activities are one source of input to domain analysis. That is, as new aspects of a domain are prototyped, the results help to uncover new domain-specific information which is then used to flesh out an appropriate domain model.

In this context, the LN primarily serves as a communication link between the prototyper and the domain analyst. The LN conveys an understanding of the prototyping activity by including a description of the particular question under consideration, the context within which the question was asked, the methods used to produce a result, alternatives considered, and rationale for why particular alternatives were accepted or rejected. This type of information is valuable input to a domain analyst who is tasked with identifying and organizing the common principles underlying a given domain.

... CCM CCM is a crucial support technology for megaprogramming. Megaprogramming is characterized by long-term evolution of large systems, and heavy reuse of artifacts between threads of development. In megaprogramming, complex webs of artifacts are being evolved over a very long-term period of time. Multiple threads of development are ongoing, and backtracking occurs. Components and architectures produced in one thread – and still evolving there – are used in other threads, which are also evolving over time. The domain theory and the deliberations linking it to the software artifacts are similarly evolving in complex ways. There are complex derivation, equivalence, and compatibility relationships among all of these to be tracked. Support for version-generic references and version-selection rules, to define abstract configurations, is key for specifying architecture in the context of continuous evolution.

The CCM support for megaprogramming must be configurable, lightweight and agile, and capable of handling complex networks of fine-grained artifacts with complex internal structure.

... PM The efficient and effective development and utilization of product families will depend on a focus on quality and on continuous process improvement[4, 16]. We are investigating the specific PM needs of megaprogramming. Our results are very preliminary at this time. While the development of process models and process definitions for megaprogramming will be important, our focus will continue to be on developing models and definitions for prototyping processes occurring in the context of megaprogramming. Similarly, we will focus on enactment support for prototyping. The overall context that we are assuming for megaprogramming is described in [9].

5.5.2 Prototyping and ...

... MIF The Honeywell ProtoTech team is exploring the relationship between prototyping and megaprogramming processes. We see several potential benefits of applying MIF technology to prototyping. First, we envision architecture prototyping as a means of improving the requirements and design phases of software development. If during these phases of software development, the system's architecture can be rapidly assembled and analyzed, it would provide the customer an opportunity to map the system's requirements to a tangible model of the desired system and judge the consistency of the developer's and customer's understanding of the requirements.

Second, applying MIF technology to software experimentation holds potential for reducing the cost and time required for creating executable prototypes. The behavior of software systems is often too complex to analyze except through empirical means. However, software developers often shun prototyping due to schedule and resource constraints preferring to shoulder the risk of the unknown design parameters that could have been revealed using prototyping techniques. MIF-based tools for specifying architectures and components, component selection, code generation and synthesis should simplify and accelerate software experimentation.

... LN As an experimental activity, we view prototyping as essentially a question answering activity. That is, individuals resort to prototyping when they have a question they cannot answer via other means (e.g., reference, simulation, analysis, etc.). The LN provides a prescriptive structure for guiding this process and recording the results. In particular, the LN supports question refinement, development of an answer strategy, recording of an answer derivation (including alternatives considered and rejected), answer interpretation, and answer refinement. Emphasis is placed on capturing the rationale for decisions made during this process. This structure helps ensure that the question is answered appropriately and that the results are communicated back to the question asker in an effective manner.

... CCM Prototyping has strenuous change management needs. Prototype systems evolve rapidly, to return the answer in a timely fashion (under time and resource constraints), to react expeditiously to the growth of the graph of questions and derived questions, and to backtrack in response to the failure of lines of investigation. In addition, multiple threads of development are typically going on concurrently. CCM will be a crucial support technology for prototyping.

... PM Prototyping processes have subtly different goals and drivers from purely production-intent development processes, and operate under tight time and resource constraints. Thus careful guidance of the process, with nimble reactions to changing circumstances, are crucial. Prototyping must occur rapidly; the situation typically changes rapidly, as answers are accumulated, derived questions are uncovered, and the needs of the broader development process change. Monitoring and guidance of the prototyping process is thus crucial, including tracking of resources expended, allocation of priorities to the nodes in the graph of questions, and feedback from process measures. Prototyping plays a variety of roles during the various stages of system development, and plays different roles depending on the maturity of the application domain. There are thus a variety of prototyping processes; we are participating in an exploration of the space of prototyping processes, to characterise commonalities and determine the key needs of prototyping.

5.5.3 LN and ...

... CCM Given that prototyping is an experimental question answering activity, one expects that prototypers will frequently explore multiple paths, possibly due to a risk reduction strategy or due to encountering deadends. In the process of exploring alternatives, the LN must manage the results of the prototyping process which are captured as a complex, evolving network of interrelated artifacts such as questions, hypotheses, goals, assumptions, deliberations, decisions, designs, experimental apparatus, and experiment results. In this context, an answer, recorded as a LN entry, consists of a set of configurations where each configuration represents an alternative line of thought leading to a partial or complete answer to a given question. That is, an answer is really a set of potential answers each of which is represented as a configuration of "the answer". The LN will rely heavily on CCM capabilities to manage these artifacts.

... PM As previously mentioned, we view prototypmg as a question answering activity in which the LN supports question refinement, development of an answer strategy, recording of an answer derivation (including alternatives considered and rejected), answer interpretation, and answer refinement. That is, the structure of the LN directly reflects a process for question answering. The PM guides these deliberations. PM is particularly important at the meta-control level, for switching between interrogative modes. Thus both simple guidance (based on defined processes for experimentation and deliberation) is helpful, as well as reactive guidance based on monitoring and measurement of the prototyping process.

5.5.4 CCM and PM

CCM and PM have complementary concerns. CCM is primarily concerned with characteristics of the product — specifically, with ensuring the consistency of the product. PM is primarily concerned with characteristics of the process — specifically, with the consistency, results, and variability of the process.

CCM and Process Management (PM) are quite interdependent; see Section 5.3.

6 Summary

The DARPA ProtoTech program consists of a community of academic and industrial teams, whose collective goal is to identify and develop supporting technologies for prototyping processes within a megaprogramming-based software engineering lifecycle.

This report presents Honeywell's view of software prototyping as an experimental process whose results drive the evolution of a domain theory, the principles governing the engineering of software within a particular application area (e.g., avionics flight management systems or cockpit simulators).

Current software engineering methods and tools provide inadequate support for prototyping. System specification methods focus on modeling "what" the system is intended to do in terms of a requirements model and "how" the system implements the requirements in terms of an architecture model. No current system specification techniques include formal methods for organizing the rationale for the system's requirements and architecture (i.e., "why" the system requirements and architecture are organized as they are.). Design rationale is an essential part of the results generated from a prototyping experiment.

Architectural specifications are also a critical aspect of an evolving domain theory. A system's architecture represents a set of design decisions in which the requirements have been packaged to achieve the desired system behavior. Prototypes are often built to answer questions based on incomplete sets of requirements or a lack of understanding of the available technology for implementing the requirements. The result of such prototyping is a refined or extended set of requirements and a set of architectural considerations. Current design methodologies provide little or no support for expressing or evolving architectural concerns.

Prototyping processes are inherently experimental in nature. System development processes from which prototyping processes are usually spawned must be rigid and well-controlled. The need exists for an integrated model of these two types of processes to control flow of information between the two.

The Environment for Software Prototyping (ESP), a prototyping environment model, is used to evaluate mechanisms to address these problems. The ESP model consists of a component shelf, a repository for architectures, components and implementations, and the workbench, a workspace for conducting prototyping experiments.

Within the ESP framework, standard engineering methods and models are extended to address requirements unique to prototyping experiments.

The Lab Notebook is a structured deliberation support tool for guiding and capturing deliberations which occur during prototyping processes. The methodology it supports addresses the problem of recording design rationale and integrating such information with the requirements and architecture models.

The architecture representation language (ARL) provides the prototyper a notation for describing the structure of experiments and generating and evaluating executable representations on a breadboard, the software bus.

The process manager is loaded with a model of the chosen prototyping process and guides the experiment, ensuring completeness and consistency of the software artifacts. The process engine also supports transition of information between the system development and prototyping process models.

ESP is being used to evaluate the efficacy of these technologies. System development processes and design methodologies used in an avionics systems development environment are being modeled and adapted to test the effectiveness of these solutions to the problems sighted.

A PM notes on Experimental vs. Production Intent

A sketch of how experimental intent can be used as a distinguishing factor in characterizing prototyping processes. This is part of an ongoing discussion in the ProtoTech Impact working group. It is, in part, a response to presentations from Dennis Heimbigner (U Colorado), Bill Carlson (Intermetrics), and Leon Osterweil (UC Irvine) made at the Winter '92 ProtoTech community meeting.

```
Elaborated thoughts on the characteristics of prototyping processes.  The
handles are:  consider prototyping's goals;  consider the constraints on
prototyping processes.

o Definition of prototyping:  (proposed new definition)

  Construction of and experimentation with a model of a software system to
  answer questions about the system's application domain (problem domain
  and/or solution space).

    o Model: shares some characteristics with the deliverable system.  May
      *be* the deliverable system itself -- e.g., in the case of an
      evolutionary prototype.
    o Construction: this is a no-op if the prototype already exists.
    o Experimentation -- does prototyping *necessarily* involve the
      execution of the model and use of results?  Probably so.
    o Problem domain  and/or solution space -- the prototyping may be
      exposing properties of a proposed system, or learning about the
      domain / problem space that system addresses.  It may be difficult or
      impossible to tease apart the problem domain and the solution space
      (e.g., in ''wicked'' problems.)

o One dimension for characterizing prototyping processes: degree of
  interrogative intent.

          |100% ------------------ 0%|  production intent
          |0% ------------------ 100%|  experimental intent
           -------------------------     -
          | \___                    |    |
          |     \__                 |    |
          |        \_  experimental |    |
          |          \_    intent   |   proportion
          |production  \_           |   of intent
      |   intent       \__           |   |
      |                   \___       |   |
      |______________________\___|   -
        ^           ^          ^


      |--------------|
       tends to be used
      in well-understood
          domain
                 |--------------|
                  tends to be used
                in not well-understood
               domain, or 'wicked' problem

        ^           ^          ^

       (1)         (2)        (3)
Consider three example processes, plotted along the X-axis --
  - process (1) is producing a precedented system.
  - process (2) is an "evolutionary" prototype -- the plan is to
     deliver the system under construction, but also use it as a
     learning vehicle, via iterative development.
  - process (3) is a "throwaway" prototyping -- the plan is to discard
     the system under construction (or most of it).

  o Production Intent: intend to deliver the system to customers.
      o Building the system for customers to use it.
      o Deliver to customer:  the system.
      o Important system characteristics: robustness, usability,
        maintainability, documented.
      o Need: coding standards, documentation standards, relatively
        tight controls on changes.

  o Interrogative Intent: intend to use the system to answer questions.
      o Building the system to learn something before commitment.
      o Deliver to customer: an answer, and its associated knowledge.
      o Important system characteristics:  quick to construct,
        estimable cost, answers the question, captures the associated
        knowledge, reduces costs/risks of development.
      o Need: scientific method support, ERC/communication support
        (deliberative), quick-construction support, instrumentation
        support, loose controls on changes.
```

- Mixed Intent: intend to answer questions, and to deliver it.
  - Building system to deliver, but also using it as a platform for learning something before commitment -- typically iteratively.
  - Deliver to customer: the system; capture the answers/decisions and associated knowledge, too.
  - Important system characteristics: same as production intent, plus: support for periodic restructuring, captures the associated knowledge, reduces costs/risks of development, evolvability, well-architected.
  - Need: refinement languages? separate functionality from optimization? lightweight CCM?

- Note that the proportion of intent of a process may change over time! I will often start an effort over near the high-experimental-intent end, and move toward the high-production-intent end as we progress through the lifecycle.

- Note that a process may be at one place on the scale, but one of its subprocesses may be at another!
  - For instance, a large process with high production intent may have a subprocess (dealing with one subassembly) that has mixed intent; and it may apply several small throwaway prototypes to resolve certain issues (subroutine-calls to high-experimental-intent subprocesses).
  - So to say that process P is at point N on the scale doesn't mean that its subprocesses P.Q, P.R are necessarily at that point, nor that the process which P is a part of is at that point on the scale.

- Note that I may choose to use an evolutionary strategy at any point along the intent dimension!
  - If I am doing a throwaway (don't intend to every give to customers), but intend to use it repeatedly over a long period, I may design my throwaway for evolvability, and evolve it over a long period before chucking it.
  - If I am doing a purely production-intent effort, I may choose to apply an incremental development strategy -- design for evolvability, gradually add functionality over time -- motivated by early testability, easier maintainability, etc.
  - And of course if I am doing a mixed-intent prototype, I will want some form of evolutionary strategy.

o Note that ''pure experimental-intent'' and ''pure production-intent'' processes hardly ever occur!

o Pure production-intent processes get replaced by application generators. If my design problem has *no* questions that need empirical evidence to resolve, then if I must do this sort of system often enough I'll write an application generator which ''cans'' the process.

o Pure experimental-intent processes probably only occur in Clueless Prototyping (see below), at the ''immature'' end of the Domain maturity model; in most other cases, I have some hope that part of the spec, design, or code will be used in a product later.

o Most software engineering processes are thus a mix. Thus protototyping support (appropriate when you have experimental intent) will be widely applicable.

o One dimension for characterizing prototyping processes: relation to domain model.

```
 -------------------------
|                         |
| prototype               |
|                         |
|            DELIVERABLE  |
|                SYSTEM   |
|  Domain                 |
|  Model                  |
|_________________________|
```

How do these three things interrelate, for the given prototyping process?

Three categories of prototyping processes:

o Clueless Prototyping. I don't have a domain model at all. I am exploring the domain to try to create an initial model.

o Getting a seat-of-the-pants impression.

o In the why-won't-the-car-start example, this is the strategy which is (initially) appropriate for someone who's never even opened the hood before. Poke around, try things randomly, think about what happens.

o (This example is a debugging rather than a construction example -- it's weak in that way.)

o I may be prototyping to formulate the problem rather than to

```
        find the solution -- in that case, it's likely that the
        requirements and specification won't be known until the
        system is pretty far along.
      o This occurs more frequently at the 'Concept Definition /
        Feasibility Analysis' far-front-end of the lifecycle.
      o Needs: deliberation support (argumentative process is
        appropriate, since I have no way to judge right/wrong, yet).

  o Trial-and-Error Prototyping. I have a domain model, and I think I
    have an idea about the answer, and I am checking it out.
      o E.G., many "feasibility prototypes".
      o In the why-won't-the-car-start example, this could be "I know
        it's the fuel or the air or the fire, hmmm, let's check the
        fire ..."
      o This is undisciplined, when compared with Scientific
        Prototyping.
          o It is reasonable to do this if (a) I am pretty darn sure
            of my supposed answer;  (b) the cost of prototyping is
            reasonably cheap.
      o This is used _too_frequently_.  I am neither (1) carefully
        guiding my steps (and recording them) according to the domain
        model, nor am I (2) carefully relating my discoveries back to
        the domain model, to enhance it.  I.E., I am being sloppy.

  o Scientific Prototyping.  I have a domain model, and I am doing
    disciplined hypothesis-testing with it.
      o I may be doing the scientific method for one of two reasons:
          o 1. I am using SM to develop/extend/validate a theory/model.
            I am enhancing the domain model, learning more about the
            general.
              o E.G., in the why-won't-the-car-start example, I am
                improving my domain model by testing if there are
                other reasons why a car may not start.
              o This is ''science'', as opposed to ''mature engineering''.
              o My model need not be too good, but given an initial
                model -- e.g. from Clueless Prototyping -- I can use
                SM to enhance it (bootstrapping).
          o 2. Alternately, I am using SM to answer a question / solve a
            problem about a specific situation -- learning more about
            the specific.
              o E.G., in the why-won't-the-car-start example, I am
                carefully generating hypotheses as to why this car
```

won't start, I am testing them out and using the (recorded) results to guide the generation of new hypotheses (pruning the tree of possibilities).

- o This is engineering, as opposed to science.
- o I am systematically leveraging the domain model to attack the current problem.

o Needs: scientific method support -- experimental design, focusing, support for integrating results back into domain model.

- o Note that a different set of deliberative structures may be appropriate for Scientific Prototyping versus Clueless Prototyping. Specifically, the rule for selecting a position as a winner is different. In Clueless Prototyping, I have an unconstrained/unstructured problem, and I need argumentation, and must use judgment as the selection rule. In Scientific Prototyping, I use hypothesis testing as the selection rule -- I use the experimental results to confirm or refute the position.

o This relates to the Domain Maturity Model. Clueless Prototyping is appropriate in immature domains, which lack domain models. As the domain matures, Scientific Prototyping (and Trial-and-Error Prototyping) become applicable and appropriate. To move toward maturity, and away from ad-hoc, Scientific Prototyping is a key tool.

o We need to explore how both the ''intent'' dimension and the ''relation to domain model'' dimension relate to the global structure of the prototyping process (coroutine, call/return, etc).

- o Experimental intent: question goes one way, answer comes back. Production intent: spec goes one way (''build this''), system comes back (''here it is'').

o Prototyping as Question-Answering. Prototyping is a question-answering procedure. We may conceive of a generic question-answering routine

```
procedure answerQuestion (q) is
   case type-of(q) ...
     alreadyKnowAnswer :
       ... ;
     lookItUp :
       ... ;
     askSomeone :
       ... ;
```

```
        cluelessPrototyping :
          ... ;
        trialAndErrorPrototyping :
          ... ;
        scientificPrototyping :
```

```
o An interesting issue is the meta-control level:  when do I switch
  from one question-answering strategy to another?
    o If I ask N people and no one knows, I don't keep doing that.
    o At some point it becomes unreasonable to keep doing
      cluelessPrototyping -- my model is good enough to switch
      to scientificPrototyping.
    o At some point it becomes unreasonable to keep doing
      trialAndErrorPrototyping -- I'm getting nowhere, it's time to
      bring in the big guns.
```

References

[1] Robert Balzer, Richard P. Gabriel, Frank Belz, Robert Dewar, David Fisher, John Guttag, Paul Hudak, and Mitchell Wand. Draft report on requirements for a common prototyping system. Technical report, University of Southern California/Information Sciences Institute, 1988.

[2] "Rx for Software: Domains + Architectures + Components = Reuse", Technical Report CS-R92-010, Honeywell Systems & Research Center, Minneapolis, MN, 1992.

[3] Jeff Conklin and Michael L. Begeman. gIBIS: A hypertext tool for exploratory policy discussion. *ACM Transactions on Office Information Systems*, 1988.

[4] W. Edwards Deming, "Out of the Crisis", MIT Center for Advanced Engineering Study, Cambridge, MA, 1986.

[5] Garlan, David, "Language Requirements for Software Architecture," Rough draft circulated for discussion at Jackson Hole, WY. DSSA Meeting, March 23-25, 1992.

[6] Goodhue, D., "The Economics of Data Integration", unpublished, Carlson School of Management, Univ. of Minnesota, May 1989.

[7] Gougen, Joseph A., "Principles of Parameterized Programming," Chapter 7, Software Reusability, Volume 1, Concepts and Models, Ed. Ted J. Biggerstaff and Alan J. Perlis, ACM Press, 1989.

[8] J. Kimball, K. Thelen. "Automated Support for Process Enactment Using PREIS and AAA", Honeywell SRC Technical Report CS-R92-014.

[9] Tim King. Software reuse: Concepts and issues. Research Report CS-R92-007, Honeywell Systems & Research Center, April 1992.

[10] Jintae Lee. Extending the Potts and Bruns model for recording design rationale. In *Proc of 13th Intl. Conf. on Software Engineering*, pages 114–125. IEEE, May 1991.

[11] Jintae Lee and Kum-Yew Lai. A comparative analysis of design rationale representations. Technical Report CCS TR 121, MIT, Massachusetts Institute of Technology
Sloan School of Management
Cambridge, Mass, May 1991.

[12] Letovsky, S., E. Solloway, "Delocalized Plans and Program Comprehension", *IEEE Software*, 3(3), 5/86, pp. 41-49.

[13] Christopher A. Miller. *Learning to Disagree: Argumentative Reasoning Skill in Development*. PhD thesis, University of Chicago, 1991.

[14] Jack Mostow. Toward better models of the design process. *AI Magazine*, spring:44–57, 1985.

[15] R. Nickerson. *Reflections on Reasoning.* Lawrence Erlbaum Associates, Publishers, 1986.

[16] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, Edward L. Averill, Judy Bamberger, Timothy C. Kase, Mike Konrad, Jeffrey R. Perdue, Charles V. Weber, and James V. Withey. Capability maturity model for software. Tech Report CMU/SEI-91-TR-24, ESD-TR-91-24, Software Engineering Institute, Carnegie Mellon University, Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213, August 1991.

[17] *Proceedings: Winter 1992 Community Meeting, Snowbird Utah*, January 1992.

[18] Purtilo, J., "The Polylith Software Bus," To appear, ACM Transaction on Programming Languages and Systems. Currently available as University of Maryland Technical Report 2469.

[19] Nancy L. Stein and Christopher A. Miller. *Advances in Instructional Psychology*, "The Development of Memory and Reasoning Skill in Argumentative Contexts: Evaluating, Explaining and Generating Evidence".

[20] Stephen E. Toulmin. *The Uses of Argument.* Cambridge University Press, 1958.

[21] Wegner, Peter, "Capital-Intensive Software Technology," IEEE Software, Vol.1, No. 3, July, 1984.

[22] K.C. Burgess Yakemovic and E. Jeffrey Conklin. Report on a development project use of an issue-based information system. In *Proc. of CSCW 90*, pages 105–118, October 1990.