S3-31
N94-25196
p- 32

Abstract:

Conventional system architectures, development processes, and tool environments often produce systems which exceed cost expectations and are obsolete before they are fielded. This paper explores some of the reasons for this and provides recommendations for how we can do better. These recommendations are based on DoD and NASA system developments and on our exploration and development of system/software engineering tools.

1 Introduction

Over the past seven years our Signal Processing Center of Technology and in particular our Rapid Development Group (RDG) has been vigorously developing and applying engineering process approaches for complexity management and rapid development. The systems we target have both hardware and software components and include applications such as electronic countermeasures systems, signal classifiers, factory floor test equipment, and reaction jet drivers.

In this report the reader will find an analysis of flaws in conventional methodology, examples of innovation, and several recommendations for improvement. The key aspects of our multi-faceted approach build on architectural advances that enable hybrid system development (i.e., mixes of pre-existing subsystems and new development), process improvement, and tool developments addressing automation at higher and higher abstraction levels.

As a component of this thrust, we will report on our own prototype tools for requirements/specification engineering. Recently on the "Requirements/Specification Facet for KBSA" project, Lockheed Sanders and Information Sciences Institute built an experimental specification environment called ARIES [5] [1] which engineers may use to codify system specifications while profiting from extensive machine support for evaluation and reuse. ARIES is a product of the ongoing Knowledge-Based Software Assistant (KBSA) program. KBSA, as proposed in the 1983 report by the US Air Force's Rome Laboratories [3], was conceived as an integrated knowledge-based system to support all aspects of the software life cycle.

Historical Perspective

There are many opportunities for improving engineering processes, but several obstacles to be overcome as well. We begin by describing the current practice as a baseline. Figure 1 provides a top level view of "conventional" engineering activities. Engineers acquire requirements directly from discussions with end-users or through sponsor-authored documents. Engineers then line up appropriate data sets, extant or emergent algorithms, feasibility studies, and trade-off studies. They produce refined requirements which give sponsors confidence that

[1] ARIES stands for Acquisition of Requirements and Incremental Evolution of Specifications.

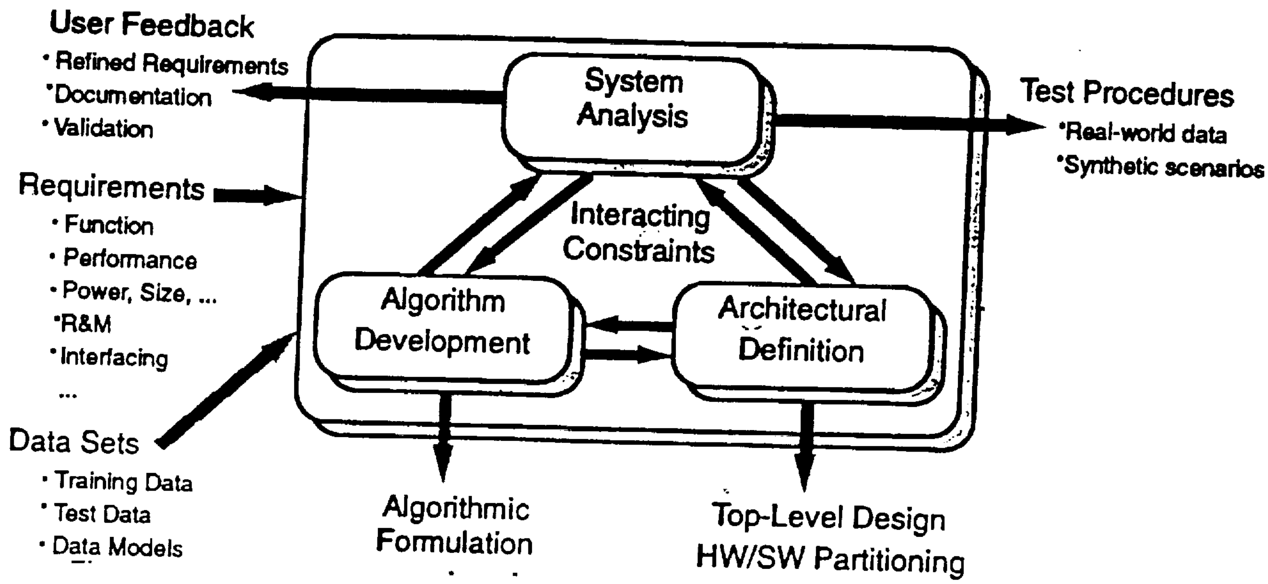

Figure 1: System Engineering Activities

the right solution will be built.

System analysis is the process of describing system functionality and managing constraints, but avoiding premature commitment to particular implementations. Engineers match functional and nonfunctional (e.g., performance, power, size, reliability) requirements against known system and component capabilities. Since the process today is largely informal, it is very difficult for engineers to avoid duplication of work (e.g., re-doing back-of-the-envelope trade-off calculations, re-inventing architectures and design solutions) or creating errors in a descriptions. Even well thought out specifications may contain missing references, ambiguous terminology, and other forms of inconsistency.

Engineers generate algorithmic formulations and top-level designs which are used to initiate downstream design, manufacture, and deployment. Additionally, they identify the real world data and synthetic scenarios necessary for conducting downstream system verification.

Engineers produce textual requirements documents, describing the characteristics of the system to be built. However, such documents are themselves but a means to achieve a more fundamental goal, namely communication of requirements to engineers and sponsors (end-users, procurement agents, etc.) and sponsors in related systems. In fact engineering media - diagrams, outlines - used along the way toward producing a written document can be extremely informative. Simulations in many forms and executable prototypes are another useful product, both to help communicate requirements and to validate the accuracy of those requirements.

Forced Early Commitments:

As we move into implementation phases, we note that the conventional development cycle is really a collection of discrete sequential steps (see Figure 2). Each step establishes a baseline and entails specific commitments. To manage large team coordination and to reduce schedule risk, engineers freeze implementation choices as early as possible - prior to partitioning of design tasks to members of a development team. For example, engineers may prematurely select a CPU, a sensor component, an algorithm, or an electronics part. Frequently, these implementation decisions are made before the system requirements have been fully analyzed and understood.

To ameliorate the effects of unforeseen, or poorly understood, requirements, system engineers impose design margins (e.g., extra memory, extra throughput, stringent power and size restrictions). The rationale behind these margins being that some physical components will exceed expectations and some unforeseen problems can be corrected by subsequent margin adjustments or, as a last resort, by writing new software whose functionality crosses physical system boundaries. Unfortunately, to achieve the margins mandated, engineers frequently introduce additional technical and schedule risk since now the required capabilities push even harder against the edge of achievable performance, power, and packaging.

Fundamental Problems:

There are several fundamental problems inherent in the "conventional" system development process. While the well-documented reasons for long development cycle times are many and varied, four significant flaws characterize the state of the practice: arms-length validation, rigidity in process and tool selection, isolated design activity, and performance-driven developments. Unfortunately, all contribute to long and costly development cycles.

- *Arms-length Validation*

  A key reason for end-user disappointment with a product is that during a long development cycle, end-users receive incomplete informal information on how the system will perform; once field and acceptance testing begins, end-users can be "surprised". Management can not directly observe development and hence institutionalizes control regimes which take on a life of their own. Unfortunately, in using these "arm's length" regimes, the best efforts of interested observers may fail to get at the real requirements - requirements that can only be accurately stated when end-users have the opportunity to interact with actual system implementations.

  If a surprise requirement is uncovered and a corrective action is taken (e.g., utilizing software that will achieve the design margins), this often occurs late in the development cycle when typically the program is fully staffed at the most expensive portion of its

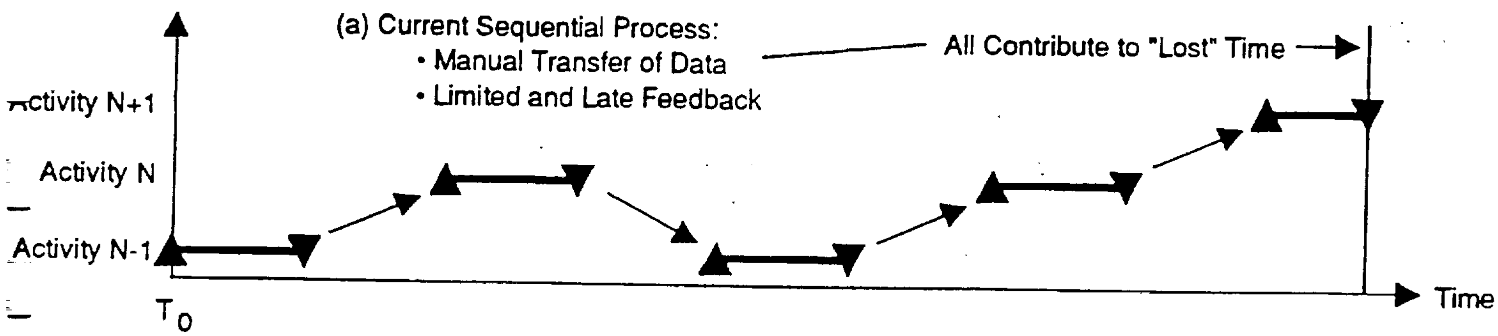

Figure 2: The conventional development cycle as a collection of discrete steps

costing profile. Consequently, even minor corrective actions have dramatic cost and schedule impacts.

- *Rigidity in Process and Tool Selection*

  Unfortunately, we often standardize on practices and/or tools and derive methodology from these standards rather than letting specific applications define their development methodology (and concomitant supporting practices and suite of appropriate tools). An effective process needs to "steer" through the design space avoiding obstacles and pitfalls as they arise.

- *Isolated Design Activities*

  Mid-phase design engineers are often isolated from system level requirements definition, system production, and fielded system maintenance/support/upgrade. For example, the feedback loop from design to manufacturing and back to design can take several days at best.

  Producibility guidelines, available on paper, and to a limited extent in advisor software packages, help engineers avoid only the most obvious pitfalls such as exceeding bounds on chip size.

  The cost estimation tools available today are not tightly integrated with the design process. These estimation tools derive cost from abstract parametric data (e.g., team

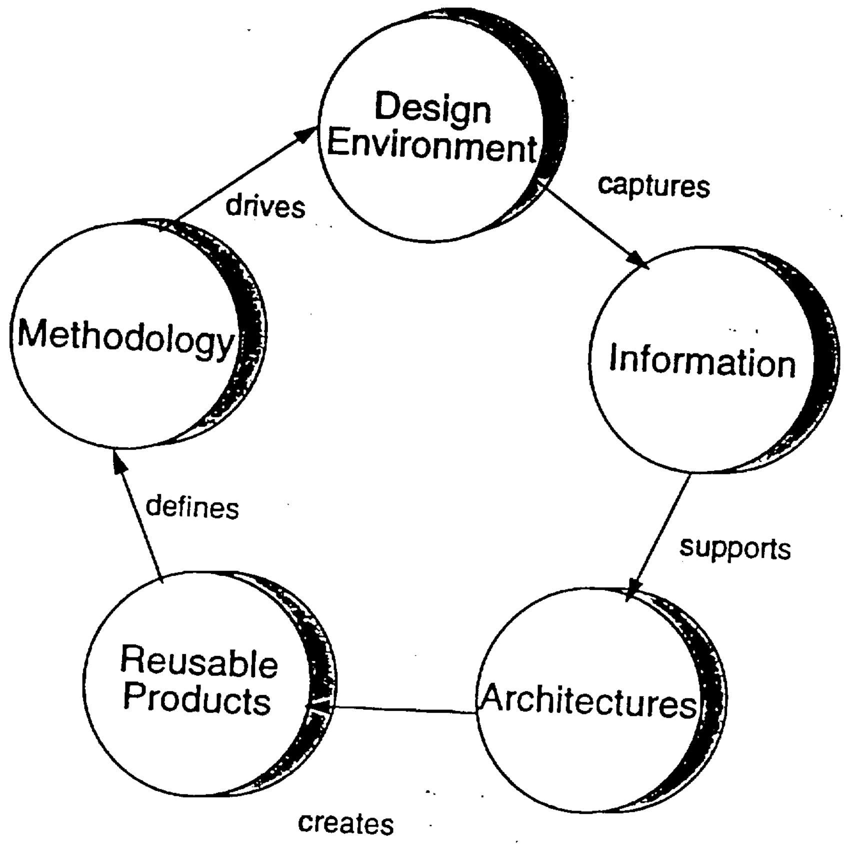

Figure 3: An engineer-centered model for the adaptation of architectures, processes, and design environments

Problem	Illustration of Approach (section 2)		Elements of the solution (section 3)		
	AIPS	RJD	Arch.	Process	Envir.
Validation	x	x			heterogeneous
Rigidity	x			flexible	
Isolation				cost-driven	information
Drivers		x	reuse		

Figure 4: Roadmap for the report

People initiate change from modifications at any point in the diagram. Architecture issues include availability of reusable components and careful attention to modularity (fire walls and internal health management) and interconnection issues.

Process requirements drive the need for modifications to the Design Environment (e.g., new point solution tools and new software capabilities). Conversely, at some point (the elusive paradigm shift), we find that the capabilities of new tools dramatically shift our conception of process.

Throughout the report, we will stress the importance of viewing change management as a design engineer's activity that is scheduled part part of the development process itself.

Overview of the report

Figure 4 cross references the problems above with descriptions of design data points and enabling technologies as they are described in the paper below.

In Section 2 , we provide case studies of two small effort emphasizing progress that is possible when we take prescriptive steps to avoid some of the common pitfalls. Section 2.3 presents a vision of the future through a scenario that is likely to occur within the next four to five years. Section 3 supports this position by describing our experience and observations on prevailing industry trends. Finally, in Section 4 we make several specific recommendations for process improvement.

2 Rapid Development Data Points

This section contains two data points in process improvement and a goal scenario.

2.1 AIPS: A Case Study in Rapid Development

AIPS is a completed digital hardware initiative which illustrates the opportunistic definition and employment of a heterogeneous design environment and the employment of a flexible process flow including the development of *virtual prototypes* for the target system. In this 1991 project, our RDG group implemented a radar pulse feature extractor system in less than six months. The system's sponsor required an advanced system operating at the 50MHz rate. An existing prototype board running at 12.5 MHz demonstrated needed functionality, but could not keep up with the required data rates. To bootstrap the effort, the sponsor furnished a real world data set useful for validating designs, an interface specification document, and only the schematic for the prototype board. Hence, RDG faced a severe reverse engineering task. In addition, scheduling constraints were very tight. The sponsor needed to have a fielded system within nine months. Final testing would only be possible when the system was integrated in the field.

During the first three months of the effort, RDG worked with sponsor system engineers to explore possible ECL, ASIC, and FPGA solutions. The tight schedule was a major concern. While ECL and ASIC solutions could achieve the needed speed, they presented a serious design risk: the commitments made would have to be right, since there would not be time to start over again. While size might need to be increased with an FPGA approach and timing would not be optimized, this solution would adjust to changing requirements or design miscalculations. Results of the analysis were not conclusive, but RDG opted for the FPGA approach to minimize program risks.

Opportunistic Tool And Process Selection

The engineers were well aware of the need for critical point solution tools to achieve system goals. Figure 5 shows a subset of the tools that were available on our Sun platforms. Although the tools were not all tightly-coupled (i.e., within a unified framework), file-level transfers of information were easily accomplished. RDG had considerable experience with all the tools and an awareness of the challenges associated with mixing manual and semi-automatic efforts to push through a design and implementation within the remaining six months.

First, RDG generated a work package justification. MacProject, an automated project scheduler, was used to set up the program schedule. Figure 6 presents this initial schedule (the white boxes) and a snapshot of program completeness (the percentages complete illustrated

Work package justification	- MacProject
Algorithm Development	- Matlab
Analysis	- XACT
Translation	- XNF2WIR
Simulations, netlist	- Viewlogic
Word & graphics processing	- Framemaker

Figure 5: A partial system development environment

with the black boxes). In order to put the schedule together, our engineers interacted by phone with component and tool vendors. RDG needed to be sure that FPGA simulations would give reliable results at the 50MHz rate.

Next in an architectural analysis step, RDG investigated the possibility of a multi-board solution. This approach would provide fault-tolerance and required throughput, since a multi-board system could route input to parallel boards running at less than the 50MHz rate. The architectural analysis effort was performed with paper and pencil, white board and marker. Since the overall program was in demonstration/validation phase, the sponsor agreed that adding the additional boards and trading size for performance was a valid option. Clearly, this is not always the case. But a lesson to be learned is that every job has such opportunities that can be exploited - if design environments and methodologies are flexible.

Following the architectural analysis, RDG initiated two efforts in parallel. In the first effort, they reverse engineered the prototype schematic to capture functionality in Matlab, an algorithm development tool. By running Matlab scripts on the real data, RDG discovered that some threat situations were not properly characterized by the original data sets. By going back to the sponsor and demonstrating algorithm functionality, RDG was able to converge on a new specification which more accurately reflected real world environments.

At the same time, RDG began the process of allocating functionality to the multi-board configuration. RDG used the simple box and arrow drawing capabilities of a word processor

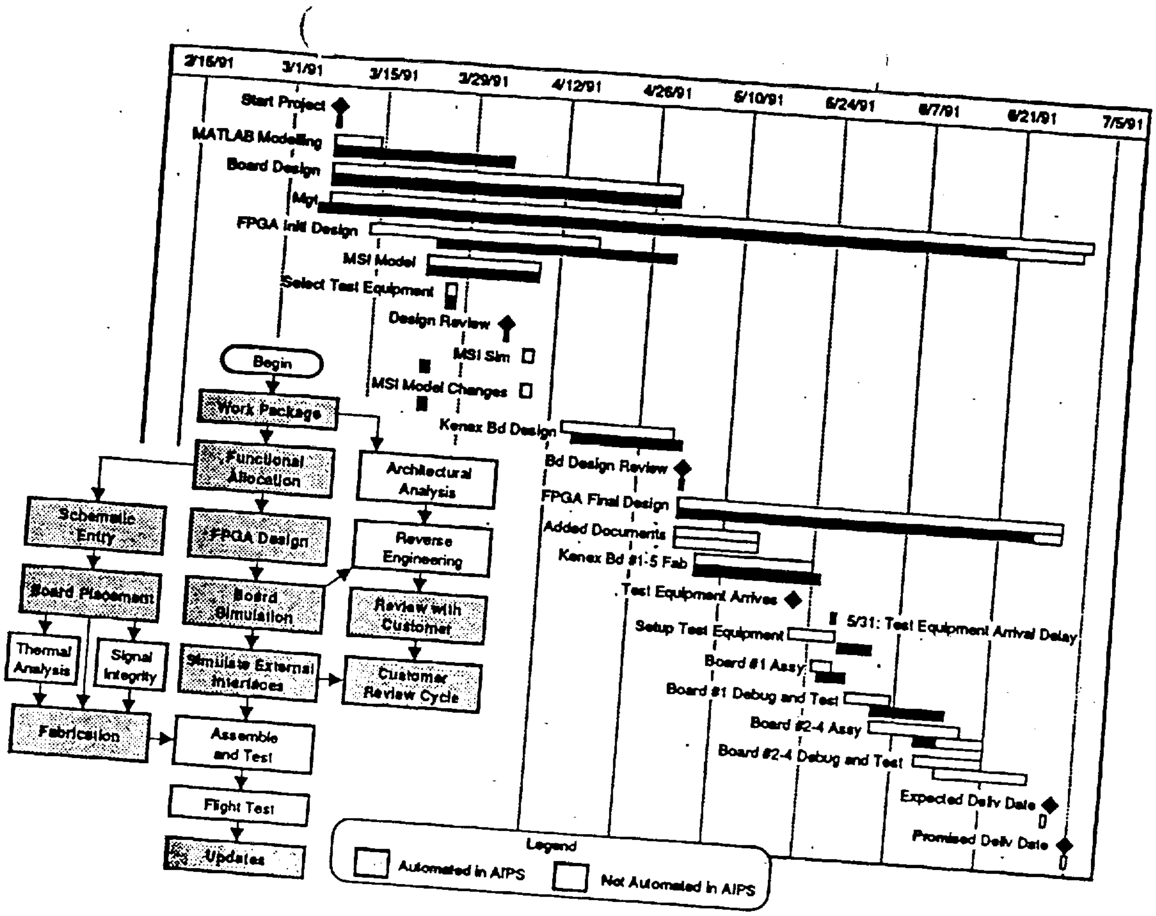

Figure 6: Project Schedule Example

to capture design choices.

Virtual Prototyping

Having chosen a baseline, RDG started down two independent paths to speed up overall design time. In one, engineers used XACT tools to describe and analyze the FPGAs, and in the other, engineers used Viewlogic tools to generate simulations for the boards. While there was no on-line traceability between the functional allocation, the Matlab scripts, and the schematic, RDG bootstrapped construction of the board schematic by purchasing and integrating vendor models. The two independent design efforts were automatically linked through Xilinx's XNF2WIR which translates XACT FPGA descriptions to Viewlogic format. The resulting Viewlogic description is an example of a *virtual prototype*, an executable model made up of a mixture of hardware or software fragments.

By using the virtual prototype, RDG identified errors in the external interface specification. The specification incorrectly set the number of clock cycles for the handshaking protocol between the platform control system and the signal processing subsystem. RDG used the virtual prototype to demonstrate the problem to the sponsor and this helped convergence on an improved interface specification.

Progress continued as RDG used Viewlogic tools to generate board layout placement. This placement needed to be checked for thermal required data rates. While analysis tools were available and might have been helpful at this point, RDG weighed the cost and schedule impact of tool acquisition and training against the value-added to the program. The engineers could not justify utilizing these tools. Rather, RDG relied on manual inspections. Clearly, more automated verification would have been desirable, but this was not a justifiable option given other development constraints.

When the analysis was completed, RDG electronically sent Viewlogic-produced netlists to a board fabrication vendor. When the completed boards were received at Lockheed Sanders, our operations department manually assembled them using RDG's schematic. Each board was individually tested first at 33MHz (a sufficient rate to meet performance requirements using four boards) and then at 50MHz (the desired target rate for a single board). Finally, the sponsor placed the boards in the fielded system. While our system had met its acceptance test criteria, the sponsor discovered that they had a problem: the AIPS system did not correctly identify the features for an unanticipated class of pulse train types.

The Payoff for Virtual Prototypes

RDG needed to find a way to identify and fix the problem. Fortunately, the control system captured data at the entry and exit points of the AIPS subsystem and RDG was able to run this data through the virtual prototype. This identified the problem as an inappropriate threshold setting and RDG used the virtual prototype to isolate the problem. This step by itself justified our choice of FPGAs. Engineers found a *modification entry point* only slightly upstream from the point at which the error was discovered. Using XACT, RDG created new PROMS which reprogrammed the FPGAs and sent these PROMS to the sponsor for a successful upgrade of the fielded system.

In summary, the key points to the AIPS initiative included:

- The use of an integrated suite of development tools
- A very flexible approach to requirements acquisition
- The development of a virtual prototype

2.2 Reaction Jet Driver

The Reaction Jet Driver (RJD) is an analog and digital hardware system that is currently under development. The RJD will interpret commands from a flight control system and energize two solenoids to independently open and close valves which control reaction control system fuel and oxidizer flow within a thruster. The RJD features health management capability and will incorporate flight safety design features. RJD is being developed in two phases. Phase I is a prototype system. This single jet system is intended as a precursor to a

Phase II, fully engineered system for initiating and monitoring jet firings for a wide variet of platforms. The multi-phase effort illustrates several points for process improvement in th area of architectures and process flow.

Requirements Acquisition

The RJD prototype requirements evolved over several months in the spring of 1993. In addition to defining performance requirements and physical constraints, the analysis centered on several market-oriented questions. For example, we needed to identify potential installations/platforms for the work. While we recognized the need to develop health maintain capability, it was important to identify the potential insertion points in plans for existing or new platforms. It was also important to layout a strategy for promotion of our technology - a form of virtual prototyping that extends across multiple programs.

With all these concerns in mind, we came up with the following list of essential requirements for the RJD.

- *Recovery From Failure*: The prototype will recover from a single device failure on oxidizer or fuel paths. The system shall be able to continue to control the solenoids in the event of this single localized failure.
- *Independent Control*: The prototype will be capable of independently controlling the oxidizer and fuel solenoids.
- *Support Diagnostics*: The RJD electronics will support diagnostics to test the capabilities of the RJD subsystem to perform specific firing sequences.
- *Perform Initialization Built-In-Test (perform-IBIT)*: Perform-Initialization-Built-In-Test is a process which operates in the Startup Mode of the system and tests for RJD readiness.
- *Perform Continuous Built-In-Test (perform-CBIT)*: Perform-Continuous-Built-In-Test is a process which operates in the Normal Mode of the system to continuously test the RJD operation and manages any error signals from RJD components.
- *Determine Shorted Load During Jet Firing* Determination of a shorted load is also a Vehicle Health Management concern. A shorted load may indicate a lack of a jet firing. Reporting of this condition will be required for an understanding of jet integrity and possible jet selection work-arounds.
- *Pressure Indication Processing*: The RJD will process pressure indications coming from a pressure transducer in the jet chamber. Processed transducer signals will be digitized and can either be stored in the digital circuitry or transmitted directly via the 1553 bus.

- *Solenoid End Of Useful Life (Solenoid-EOUL)*: This is a definite goal for the final system, but will not be a capability of the prototype effort. The capability to predict solenoid-EOUL will be explored during prototype development. Degraded solenoid performance has been observed immediately prior to solenoid failure. The degraded performance has been identified in the solenoid turn-on current waveform. The means of determining when a solenoid has reached the end of its useful life prior to its failure is being researched. Specific phenomenon seen in the solenoid turn-on current have not been conclusively documented. Hence, the contractor shall consider Solenoid-EOUL to be an investigation issue but not a capability of the prototype. In support of post flight evaluations and potential on-board vehicle health management needs, the prototype shall effectively record as a minimum solenoid activation signatures for every 50th firing of vernier jets and every 3rd firing of primary jets. It is estimated that in the course of a typical mission the vernier jets will fire up to 5000 times while the primary jets will fire approximately 300 times.

In phase II, we will re-open many requirements issues as we move toward a fully engineered system. Note that many typical hardware nonfunctional requirements (e.g., power, volume, weight, environmental stress) are *NOT* in the above list. This is an intentional omission. The prototype addresses health maintenance and a re-engineering of analog-based control to a digital logic approach. This does not mean that other nonfunctional issues are totally ignored. Part of our approach has been to select high performance components whenever possible and to specifically record design decisions which will be subject to change when upgrades are necessary.

Conceptual Design

Since, the digital circuitry appeared to be straight forward, we initiated the design with a heavy concentration on the analog portion of the system.

To a first approximation, RJD is a pair of "smart" switches. This concept is shown in Figure 7. Note that a single switch can be modelled as two series switches in parallel with two more series switches. We choose this configuration due to the major design concern of preventing a failed switch device from powering a solenoid. This concern is heightened since the most common failure mode for semiconductor power-switching devices is a main current path low impedance connection. To avoid this sort of failure mode, we selected a series connection of two power-switching devices.

At a conceptual level, the architectural components of this design consist of primary energizing paths, cross-over networks, and the controlling digital circuitry.

- *The Primary Energizing Path*

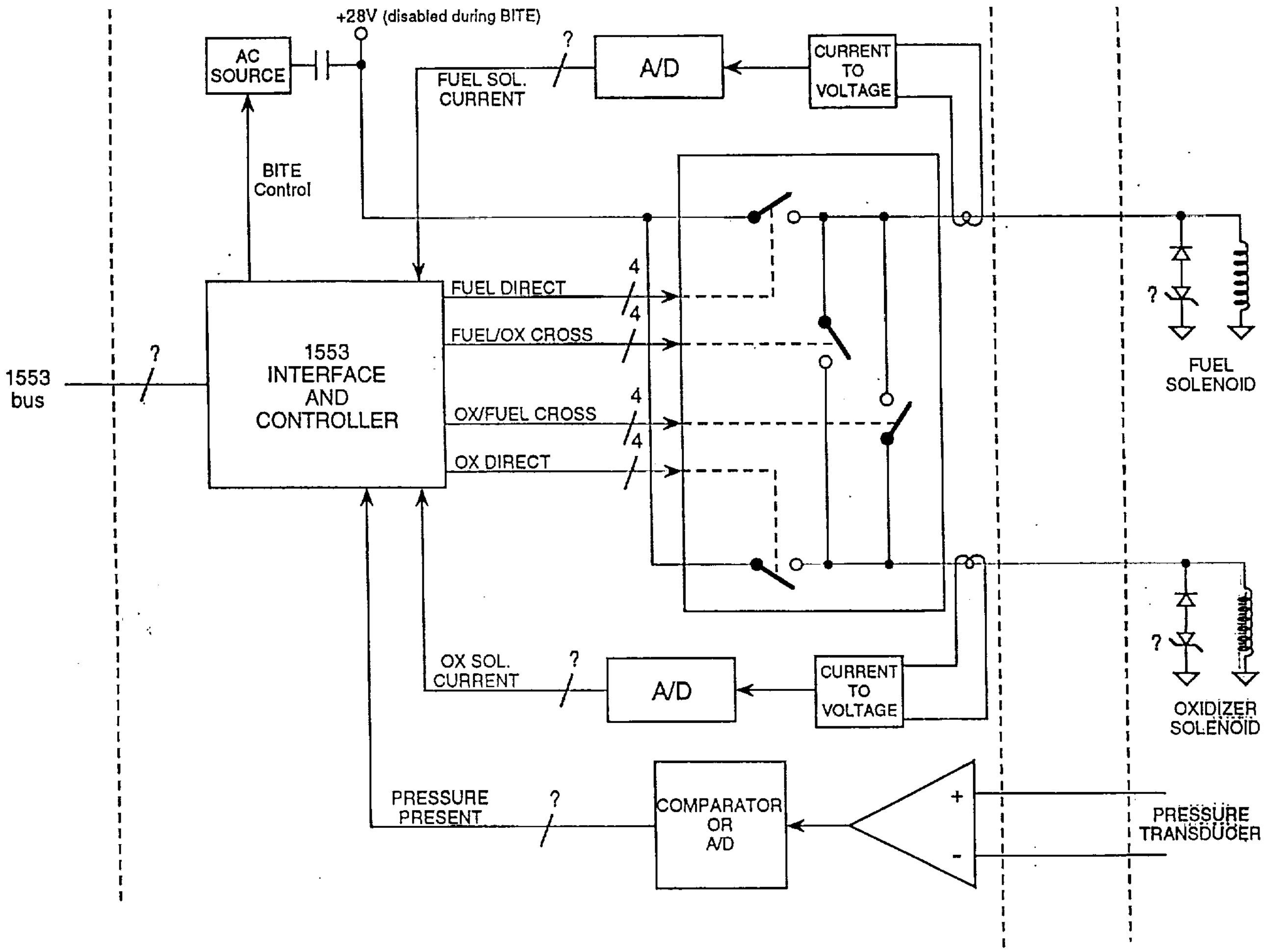

Figure 7: Top level view of RJD

The primary energizing path is a collection of four power-switching devices. Each solenoid pair will have its own primary energizing path to be used the majority of the time for energizing the solenoid. This satisfies the a specification for independent control of solenoid circuits.

- *Cross-over Networks*

  Failure recovery is provided in the event of either a power-switching device failure on both device paths or a failure resulting in an inability to command the four main devices in one of the other primary energizing path. The design contains an two additional sets of two pass elements in series connecting the oxidizer and fuel main energizing paths. These series connections form a network which allows the energizing of either solenoid from a redundant energizing path.

- *Digital Circuitry*

  Conceptually the digital circuitry divides into three parts: a controller, digital storage, and 1553 bus interface. The controller will provide control signals for sampling and addressing needed by the analog and power-switching circuitry. The storage will retain all pertinent mission information for later processing. The 1553 bus interface will provide for communications with the flight control system.

Each of the above three modules can be instantiated through many choices. For example, we have selected Mosfets for the primary energing paths and cross-over network modules and have selected Xilinx FPGA's for the digital side.

RJD upgrades

The prototype should be seen as an initial entry for a family of RJD realizations which play well with system and subsystem-level vehicle health management strategies.

For example, a multi-jet RJD configuration may consist of several analog driver printed circuit boards used in conjunction with a single controller board. This option will save space and reduce digital software overhead since all communications are handled by a single controller. In addition, the analog driver and controller boards can be contained within a single card cage simplifying the construction. Note that the modular approach we have taken in our design has successfully segmented the digital versus analog issues requisite for such implementations.

It is envisioned that subsequent development will benefit from good predictive capabilities for Solenoid-EOUL. This will enable local or flight control system management of jet firings to maximize overall system effectiveness and reduce maintenance costs. By recording solenoid signatures and other life-cycle histories on-board, we have taken a first step toward providing sufficient diagnostic information flow to centralized or distributed situation assessment and risk management processors.

Another capability to be explored is the introduction of-redundant fault tolerant 1553 interface configurations. Specifically, we need to understand the throughput impact of using quad-redundant interfaces in concert with RJD's.

When implementation is finished this fall, the RJD will serve as a starting point for a number of critical investigations.

In summary, the key points to the RJD initiative included:

- A highly iterative approach to requirements acquisition
- An exploration of analog engineering methodologies
- The tension between state of the shelf engineering and market-driven high performance forces

2.3 A Vision for the Future

Figure 8 illustrates several key features of the typical flow of design information in a future scenario. Much of the process flow mirrors that of the AIPS and RJD efforts, but the design environment has dramatically shifted the operating point toward more effective machine-mediation. Engineers work from statements of need, mission descriptions, conditions in the environment of the proposed system, requirements for new systems or perhaps descriptions of existing systems which are targeted for upgrade.

Design Environment

As a first step, the engineer identifies an appropriate tool set for handling the design and development. The expectation is that there will be multiple entries for each type of required tool and that tool selection will be driven by the needs of the specific application as these needs flow down to methodological commitments. The resulting tool set will probably contain one or more system engineering requirements capture and traceability tools, software modeling tools, analog simulation environments, and hardware schematic capture tools.

Since the design environment is tool inter-operability-centered rather than centered on specific CAD tools or frameworks, the engineer will mix and match tools to optimize engineering performance. Many of the selected tools will be available on a "fee per use" basis. That is to say, rather than making outright purchases of tools, companies will pay vendors for time spent in utilizing the tool. Importantly, this technology lowers the entry cost for both developers and tool vendors, and with more players in the field we envision a dramatic increase in the rate of innovation.

Substantial Machine Initiative

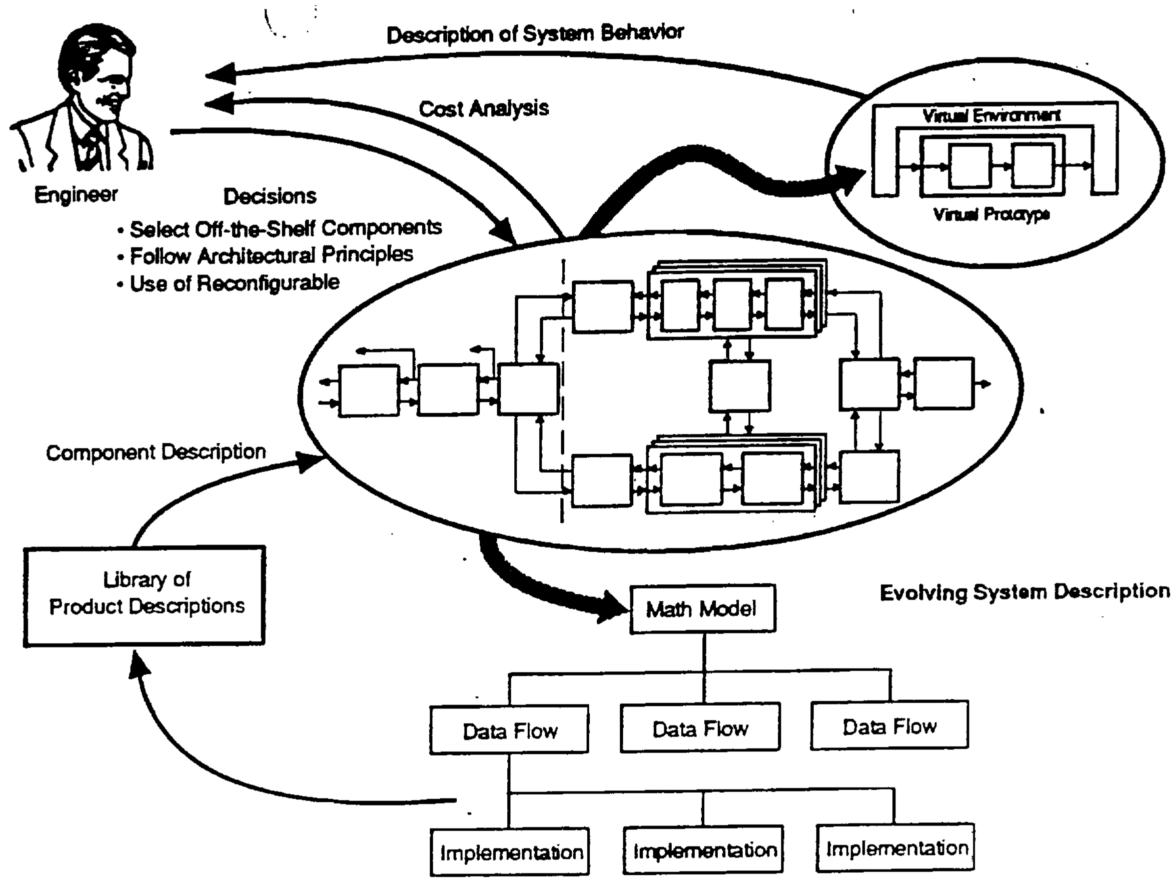

Figure 8: Achieving substantial process improvement through a free flow of design information

As a first design step under machine-mediation, a system engineer and an applications expert check plausibility of the requirements. This analysis is based on on-line access to application-specific design rules and extensive databases of related reusable designs. In most cases, the engineers find systems with very similar requirements descriptions and they quickly assemble pre-existing module descriptions to bootstrap early simulations and basic algorithmic flow. The engineering staff creates a virtual prototype which they present (either on-site or over the network) to a sponsor. The sponsor will be able to run simulations and record observations and concerns in the active project database. For many application, engineers or sponsors will insert such simulations in distributed (i.e., with players located around the country) simulations. This cycle will be repeated over and over again as initial virtual prototypes crystallize into high fidelity simulations and then to mixes of real hardware-in-the-loop combined with some simulated pieces.

Life Cycle Cost Impact Analysis

As the design proceeds, the design environment provides immediate feedback to engineers on the life cycle ramifications of their decisions. Specific warnings are provided when a decision dramatically impacts a life cycle cost. For example, the use of a non-standard interface will adversely effect reuse and upgrade potential. Similarly, the overloading of a module may result in matched-pair packaging (i.e., coordinating the production of two or more boards which are intended to be fielded in tandem) which drives up production and field supportability costs. Hence, engineers will be able to perform on-line trade studies on implementation technologies. The trade-off between performance, throughput power, cost-centered development schedule, development time, development cost, and life cycle cost will result in early realization of near optimal designs.

The use of detailed design rules will ensure a smooth transition to producible and easily fieldable systems. Engineers will express system descriptions in an architectural format which is tightly coupled (i.e., maximizes the potential for automatic synthesis and traceability) to implementations and is "reuse-oriented". Note that this implies a specific commitment to select "state-of-the-shelf" components which may not immediately achieve full performance gains. Through this process, engineers will employ a specific reusability methodology to place new designs into the databases, thereby bootstrapping the next effort where higher performance may be achieved.

3 Process Improvement

We are working to bring this vision into common practice and as an initial accessment of our efforts we can point to architecture, process, and design environment innovations we are exploring.

3.1 System Architecture

Our architectural principles are driven by the need for reusability and upgradeability in our systems.

3.1.1 Reuse

Engineers can reduce development time by using existing requirements, design and implementation fragments. We have approached this important component of rapid development in two ways:

- *Ad hoc Reuse*

  RDG has had good success with ad hoc reuse such as accessing appropriate hardware or software descriptions and tools over the internet. The available software, including compilers, graphics packages, and editors is often of high quality due to the large number of users. These ad hoc approaches rely heavily on "word of mouth" among expert developers for success. We are finding that retrieval issues are not significant despite a lack of formalized trappings around each fragment. This approach is particularly successful for large relatively self contained software packages with well-defined functionality (e.g., an object-oriented graphics package).

- *Scalable modular architectures for reuse*

  In addition to the above abstract work to providing "reusability order" to system requirements, we have worked on defining scalable modular hardware and software architectures which specifically trade performance for reuse and upgrade potential. For example, in the signal processing arena it is possible to exploit essential pipeline architectures and to maintain modular integrity albiet at the expense of maximum performance. Once a processing approach is validated for a particular application, in subsequent design iterations it can be scaled up (if greater functional performance is required from newly available technology) or down (if size, weight, or power reductions are called for). At the same time, we conduct field demonstrations with a system design which is functionally identical but, perhaps, not form and/or fit replaceable with the final product.

- *Workspaces and Folders*

  We focused our own technology investigations on requirements reuse. The primary units of organization are *workspaces* and *folders*. Each engineer has one or more private workspaces — collections of system descriptions that are to be interpreted in a common context. Whenever an engineer is working on a problem, it is in the context of a particular workspace. Each workspace consists of a set of folders, each of which contains formal and/or informal definitions of interrelated system terminology or behavior. Engineers can use folders to organize their work in such a way that they share some work and keep some work separate.

  The folders can be used to maintain alternative models of concepts, which engineers may choose from when constructing a system description. Each model is suitable for different purposes. An engineer selects folders by building a new folder that *uses* the folders containing terminology he or she is interested in. Capabilities are provided for locating concepts in related folders, and linking them to the current folder.

  As illustration, within the ARIES project, we created a library of domain and requirements knowledge is subdivided into folders. The ARIES knowledge base currently contains 122 folders comprising over 1500 concepts. These concepts include precise definitions of concepts, as well as excerpts from published informal documents describing requirements for particular domains, e.g., air traffic control manuals.

In summary, we have developed technology which can improve the coordination of multiple engineers (perhaps representing multiple disciplines) and we have demonstrated the effectiveness of rapid prototyping methodologies which overcome some of the common pitfalls of conventional large team engineering.

3.1.2 Making upgrade explicit: Working with Families of Systems

One aspect of rapid development goals is the use of up front requirements for entire families of systems. In this view, requirements are not developed from scratch and thrown away. Rather, engineers continually look for opportunities to reuse requirements from other systems or classes of systems, and to organize their requirements in such a way that they might be usable for system upgrades and reusable on future projects. These requirements provide a baseline for subsequent development and upgrades independent of specific hardware/software solutions. That is to say, we recognize and plan on solutions that will change considerably with time as new technology becomes available and/or the operating point for person/machine interaction shifts toward higher degrees of automation.

For example, on the RJD work we have continually resisted design choices that would restrict interoperability with sibling subsystems, or would require total redesign for insertion in adverse environmental conditions (e.g., severe radiation) and/or limited volume allocation. When such choices have been unavoidable, we have explicitly documented the status of our intended system and have planned out specific upgrade paths which can be followed for future systems with the same functionality.

3.2 Process

Process innovations include iterative requirements refinement, virtual prototyping, and managing isolated design activity.

3.2.1 Iterative Requirements Refinement

This aspect of our commitments supports substantial sponsor/contractor interaction. Iterative requirements refinement involves managing system decomposition, incremental attack on requirements issues, and the use of flexible technologies with explicit upgrade paths. The RJD program has exhibited this approach in two ways. First, requirements acquisition was conducted over a period of time and involved NASA, LESC, and Sanders engineering in a joint effort to generate a set of requirements that would enable substantial investigation of digital solutions and vehicle health management capabilities. Second, the selected implementation technologies assume that requirements may need to be modified far into the implementation phase. For example, RJD engineers have employed an FPGA solution initially with an intention of building the final system as an ASIC module.

To use iterative requirements refinement, only a portion of the system goes through the iteration at a time. That is to say, engineers make explicit choices about how they will iteratively add more and more capability. For example, the RJD prototype currently under develop addresses vehicle health management issues, but makes no commitment on meeting the radiation hardness, power, or volume restrictions associated with a fielded system.

For each iteration, more functionality is added to the existing system. In our experience, there generally are three to six such iterations which last two to four months each. Design activities are performed to constrained subsets of the eventual system requirements. The scope of each iteration gradually widens as the program matures, and various design fragments are tied together.

3.2.2 Virtual prototyping and/or executable requirements

Rapid development technology enables the end-users to exercise system behavior and flesh out a good set of requirements. The methodology of allowing for a series of validation steps during the development process, progressing from a skeletal implementation to finished product in highly observable steps is essential for validation. A byproduct of such validation steps is that the need for expensive "paper" control is lessened.

The RJD prototype is of interest since it is forcing us to concurrently deal with analog and digital simulation capabilities.

3.2.3 Managing isolated design

Complex systems are extremely detailed and work is typically divided among many engineers. This division may correspond to physical system components or it may correspond to different process phases. In either case, we need to find a balance between coordinated and independent engineering since there is a significant overlap among the work partitioned out to individuals.

3.2.3.1 Capturing Design Rationale A first approach to this problem is to simply provide a method for the easy capture of design rationales so that fellow workers (or an engineer reviewing his or her own previous work) can understand the context for decisions. On the RJD program, we employed database mechanisms which support engineers in expressing engineering decisions and which enable various members of the team to browse through and understand these decisions. For example, the items below show the nature of requirements information expressed for two RJD requirements.

```
(defrequire independent-control
    :print-name "Independent Control"
```

```
  :ako flexibility
  :text "The prototype will independently control the oxidizer and fuel
solenoids."
  :implemented-by separate-circuits-for-each-solenoid
  :why-choice "This option was selected to provide for independent
 control of
the oxidizer and fuel channels.")

(defrequire recovery-from-failure
  :ako fault-tolerance
  :print-name "Recovery from Failure"
  :text "The prototype will recover from a single device failure on
oxidizer or fuel paths. The system shall be able to continue to
control the solenoids in the event of this single localized failure."
  :implementation-alternatives cross-over-network-control
  :implemented-by cross-over-network-control
  :why-choice "This option was selected to provide for fault
tolerance without
excessive duplication of components and circuits.")
```

3.2.3.2 Design Assistants: For some aspects of the task it is possible to amplify a single user's capability to handle large portions of an effort. Design Assistants take the view that it is possible to automate some design decisions or at least offer on-line advice on design decisions. The manufacturing or testing expert is now replaced with a program. The ARPA Initiative on Concurrent Engineering (DICE) contains several examples of this approach. DICE's goal is to create a concurrent engineering environment that will result in reduced time to market, improved quality, and lower cost. For example, the DICE Design for Testability (DFT) Advisor contains three components. A test specification generator helps engineers select a test strategy consistent with sponsor requirements and project constraints; a test planner finds alternative ways to test the components in a hierarchical design early in the design process; a test plan assessor uses quantitative metrics for evaluating the test plans. The DICE Design for Manufacture/Assembly system is a rule-based expert system with several components for printed wire board design. It advises board engineers on manufacturability based on specific board geometric and functional requirements and on assembly based on guidelines and cost estimation. The design assistant approach requires a substantial implementation investment. In addition, significant maintenance is required when the application domain is not stationary.

3.2.3.2.1 Design Thermometers: The goal of thermometers is to dramatically increase engineer's awareness of unit cost and life cycle cost. Thermometers display cost, schedule, producibility, reliability, and supportability estimates for a given partial design. Thermometers address an important ingredient of the solution; they help to mitigate the downstream cost associated with uninformed design commitments. Today's engineers have difficulty in giving adequate consideration to the manufacturing, verification, and support impact of their decisions. The technology is available for providing engineers with immediate feedback on this impact. What is required for this approach is to connect easily optainable metrics from evolving artifacts and to feed them into multi-faceted cost models.

3.2.3.2.2 Credit-Blame Assignment Assistants: This approach aims at improving designs by finding specific flaws and tracing them back to originating decisions which can be retracted and/or avoided in subsequent design sessions. Domain independent and domain dependent approaches have been considered.

A domain independent approach is the use of *constraint propagation* [cite Steele]. Dependency networks keep track of the assertions which lead to some conclusion. If a conflict occurs, original assertions can be revisited and modified without having to redo computations having no bearing on the conflict.

The ARIES system contains a constraint propagation system that is used for enforcing nonfunctional requirements and for managing mathematical, logical, or domain-dependent engineering interrelationships. Types of nonfunctional requirements include storage (e.g., memory), performance (e.g.,mtbf, response-time, processing-time, accuracy), physical (e.g, power, size, weight), and operational-conditions (e.g., operational-temperatures, corrosivity, anticipated wind-speeds). These properties are higly interrelated and severe requirement errors occur from overlooking these relationships. A constraint propagation system addresses this problem by performing local propagation of values and various forms of consistency checking. Propagation occurs bi-directionally through propagation rules connected to nodes in constraint networks. An underlying truth maintenance system is responsible for contradiction detection, retraction, and tracing facts back to originating assertions.

This works in the following way. Engineers enter values for various nonfunctional characteristics. The constraint processor uses a constraint network to compute additional characteristics based on the values supplied by the engineer. The constraint processor detects contradictions between requirements (e.g., 10mhz resolution can not achieved in 40 usec processing time) and indicates what additional information is required in order to enable the constraint processor to compute the value of a given characteristic (e.g., "in order to compute positional error, you need to establish sensor error, sampling rate, and acceleration characteristics of the aircraft").

It is instructive to contrast this approach to the the thermometers approach. Thermometers assume uni-directional propagation (e.g., from design parameters to cost). Constraint propagation makes no assumptions about the order or direction of computation, but does require the availability of underlying formulas or logical dependencies which may not be available (e.g., while it may be possible to deduce signal resolution from processing-time, one can not deduce board components from unit cost specification). Our view it that an appropriate mix of these two notions can provide substantial feedback to engineers on the ramifications of their decisions.

Domain dependent initiatives have addressed this issue as well. FAD [7] uses information from earlier VLSI designs to determine resource interactions, perform credit-blame assignments, and determine how early decisions in the previous iteration affect later design decisions and ultimately the resource usage of the solution. These approaches require explicit knowledge of the connections among parameters.

3.2.3.3 Tolerance for Inconsistency It is also important to look at the nature of the information that is to be shared. We have investigated mechanisms that alleviate communication problems during requirements development by making it possible to mix agreed upon components with points of view that are held locally in conflict with a team's perspective. To this end we have developed machine-mediated ways to support separation and subsequent merging of work products, rather than to force engineers to constantly coordinate whenever an area of potential common concern is identified. Although consistency is an important goal for the process to achieve, it cannot be guaranteed and maintained throughout without forcing engineers to constantly compare their descriptions against each other. One cause of inconsistency is the employment of multiple models. For example, the RJD behavior can be described as a smart switch which responds to continuous on-off commands or it can be more carefully modelled as a device which responds to discrete changes in state which need to be updated periodically to achieve continuous behavior. Consistency must be achieved gradually, at an appropriate point in the development process. Nevertheless, it may not be possible to recognize all inconsistencies within a system description.

3.3 Design Environment Issues

The potential impact of tools on process, suggests that we consider any recommendations in two waves:

- *Policies and procedures for today* - given a specific design environment maturity, what are the best methodologies for system development today? For example, we may choose to continue with some control-oriented practices because the requisite groupware technology is not available for enabling observation-oriented improvements.

- *Future directions* - how do we transition to more automated processes - more expressive power in modeling and simulation capabilities, effective reuse, improved synthesis methods, automatic design? For example, our ARIES work demonstrates that with emerging technology in place, significant change occurs in the following four areas:
  - Engineers work with on-line multiple visualizations of complex system descriptions, greatly increasing their ability to understand and manipulate system artifacts (e.g., requirements, simulations results, software and hardware implementations).
  - Engineers effectively reuse requirements fragments within entire families of developments.
  - Synthesis and validation based on hybrid combinations of reasoning mechanisms greatly improve productivity and catch requirements errors. Rapid prototyping and virtual prototyping based on initial partial descriptions helps reduce the errors and brings down the cost of subsequent development. Additional consistency checking, propagation of the ramifications of decisions, and requirements critiquing all play a role in assisting in the development of reliable systems.
  - Engineers evolve descriptions in a controlled fashion. Change is inevitable, but engineers are able to rapidly respond to changing requirements and replay previous requirements evolutions.

  This work is currently experimental, but is well beyond the "toy" phase. It reflects capabilities that should find there way into commercially available products within the next few years.

3.3.1 Requirements for Environments

A description of capabilities of existing environments and tools is far beyond the scope of this report. What we can do however is take a systems perspective and outline essential requirements for computerized environments. Key components are support for heterogeneous tools, local and remote electronic access to engineering data, dynamic cost and schedule models to support program management, libraries of reusable hardware and software components, and flexible access to standard hardware and commercial software integrated via standards.

3.3.1.1 Open and Heterogeneous It is essential for the design environment to be both open and heterogeneous. By open, we mean that the environment permits the integration of any commercially available tools suited for use in a phase of the development. By heterogeneous, we mean that multiple hardware and software development tools (e.g., hardware

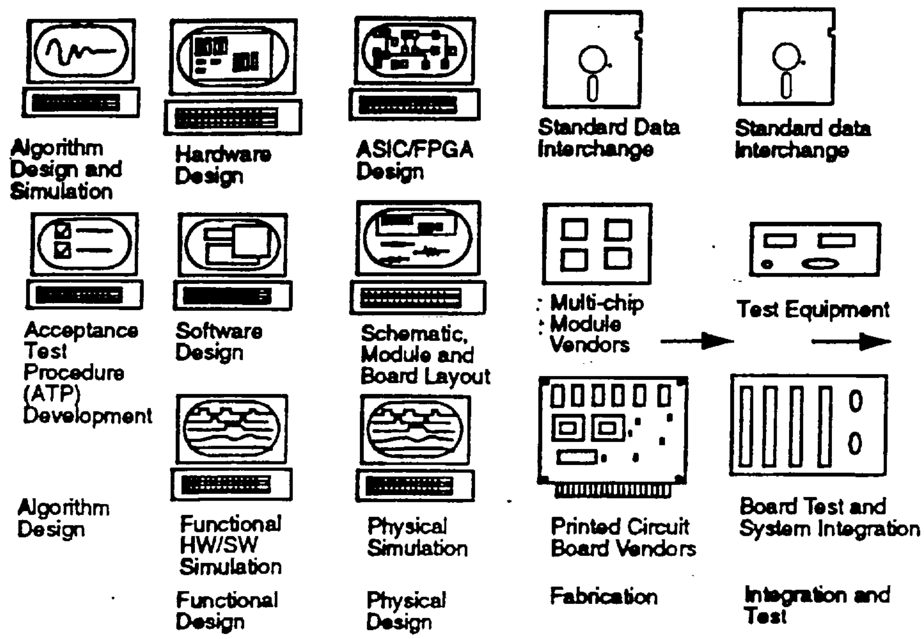

Figure 9: A typical integrated development environment

synthesis, compilers, document production, spread sheets, project management support, requirements traceability) are concurrently supported by the environment, and that there are display terminals which can access any software application running on any of the host hardware platforms from a single location. As illustration, Figure 9 shows the Lockheed Sanders integrated development environment that is based on these principles.

This requirement grows out of the recognition that while the development (or re-implementation) of a tightly integrated (homogeneous) solution is sometimes feasible, from practical considerations we seldom have the luxury to rebuild and tightly couple existing tools. It is important to recognize that the collection of commercially available tools for supporting engineering processes is growing rapidly and what we work with today is only the "tip of the iceberg" for what is possible. As new tools are introduced we need to consider how they will be used within existing informal or computer-realized development environments.

Product standards such as PDES will help with tool inter-operability. However, no single description can be expected to handle the intricacies of multiple domains. Individual problem solvers may make use of idiosyncratic knowledge that need not be shared with other problem solvers. This position is consistent with recent work on knowledge-sharing (e.g., [8]). We need sharable vocabularies which convey enough information without requiring it to be the union of all the internal vocabularies of the individual tools.

3.3.1.2 Easy Access to Information Substantial on-line data for making design and development decisions is readily accessible today, but it is can not always be cheaply and quickly obtained, nor can it be applied at the right places. The entire system development process needs to be much more open than is the case today. The benefits are obvious. For example, sponsors should be empowered to interact with and control the development because they will have access to substantial amounts of data on how a system will perform and on what options are available for development. In like manner, engineers should have access to manufacturing and vendor products and models. Links need to exist to proprietary and legacy design files so that engineers can economically integrate data into their own work space. This easy interchange of design information within and across families of systems is the key to effective reuse.

Concurrent engineering goals can be met through interactive computer models for production and support costs (and other life-cycle dominant concerns). These models need to be coupled closely to the engineers' design database. Reflecting life-cycle-cost, power, weight and other inputs back to algorithm engineers, and system implementors is essential for high quality design activity.

3.3.1.3 Explicit Notion of Process When a design environment contains some notion of the process and standard practices, it can provide direction and guidance as work proceeds. The next paragraphs briefly examine some innovative technologies that are making significant contributions to our notion of development environments.

Semistructured Messages:

This technology falls between simple infrastructures which route uninterpreted messages and deeper computation which dispatches on analysis of formalized descriptions of artifacts and problems. One illustration should be sufficient to make the point. Often engineers recognize that they are moving into "uncharted territory". They are uncomfortable about making a design commitment because they know it could lead to problems downstream. For example, a engineer would know that a non-standard chip size might create downstream problems. When a design calls for an oversized chip, the chip might easily popping off a board. Similarly, an undersized chip might be difficult to test. If experts are easily identified within an organization, the area of semistructured messages [10] can be very beneficial. For example, the engineer could enter a semistructured message such as "need ADVICE on impact of CHIP SIZE in MANUFACTURING and TEST" and be guaranteed that the message would be routed to someone knowledgeable about the impact of chip size. This would perhaps initiate a dialog and would lead to a solution in a timely fashion. Note that the message does not identify the message recipient (or recipients). It is the responsibility of the machine to determine this information from keywords in the message. The technical challenge lies in developing a specific vocabulary that can be used for the semistructured messages. The strength of this approach is that it is an small incremental step beyond current communications protocols (e.g., distribution lists in email, news subscriptions) and hence is easily achievable. The weakness of the approach is that it relies totally on the ability of engineers to be sensitive to potentially costly design decisions.

Concurrent Engineering Support:

At RPI, an emphasis has been placed on using object-oriented database technology to control concurrent editing of evolving designs. They are working on the problems of partitioning design data into coherent units to which changes can be applied and for which versions can be associated with different versions of the total design. The Palo Alto Collaborative Testbed (PACT) [2] integrates four extant concurrent engineering systems into a common framework. Experiments have explored engineering knowledge exchange in the context of a distributed simulation and redesign scenario. The strength of these approaches is that they address coordination aspects of multi-user problem solving. This focus is important for managing interactions in large organizations.

Process Modeling

Another approach builds symbolic models of some aspect of an enterprise or process. These models serve as the glue which holds a suite of tools together. For example, enterprise integration has largely focused on symbol models of the manufacturing environment. Individual nodes in these models, might serve as personal assistants for people in-the-loop or might carry out some tasks (e.g., a task on the manufacturing floor) themselves. One example of this work is MKS [9], a framework for modeling a manufacturing environment. Their emphasis has been on creating computerized assistants (i.e., small modular expert systems) which can interact directly through a dedicated message bus or through shared databases. At MCC, a CAD Framework initiative [1] provides tool encapsulation (i.e., creating a layer of abstraction between tool and user), task abstractions, design tracing, and process placement and control in a distributed, heterogeneous computing environment. It has been used for compiling and linking a large CAD tool composed of approximately 300 modules. A number of systems use a planning metaphor for modeling a process. For example, ADAM [6] unifies a number of design automation programs into a single framework associated with custom layout of integrated circuits. ADAM handles design decisions at a very course grain level. It plans activities and resources to be used and determines the parameters for each tool invocation. It then relies on the tools acting intelligently in concert even though little information is passed between them. Recent USC work has focused on synthesis from VHDL behavior level down to netlists for input to place and route tools.

In the software development arena, plan recognition techniques [4] have been used to plan and execute sequences of commands using knowledge of process actions such as build and release of software artifacts.

Distributed Problem Solving

Several issues must be addressed for achieving remote problem solving. In addition to basic infrastructure there are issues of centralization of both data and control.

By centralizing data, we ensure that tools have a consistent view of the information shared by all. In a concurrent engineering application, this repository holds the evolving agreed-upon description of the system under design.

The existence of a centralized repository does not imply centralization of all or even most of the data. Each engineer may have a private workspace containing information which may or may not be shared with others in the course of a development.

Centralized control can lead to bottlenecks [11]. Concurrent engineering problems require decentralized solutions. Computerized tools must run on separate processors co-located with the engineering staffs they support - perhaps at geographically distributed sites. These tools must communicate results over computer networks; hence questions about controlling the extent of communication and ensuring current applicability of information are very important.

Some tools may uniquely take on moderator-like responsibilities such as archiving information and nudging a development group to make progress.

3.3.2 Requirements for Tools

Tools can address either direct productivity-oriented (e.g., synthesis which transitions between levels of abstraction - specification to design, data flow to ASIC, high level language code to machine code) or evolution-oriented (i.e., manipulation of information without changing abstraction level) needs.

While computer-aided software engineering (CASE) promises substantial improvements and while considerable activity goes on in the research community, substantial portions of engineering has not yet benefited in significant ways. Tools have limited notations for expressing complex system concerns. For the most part tools have their origins in methodologies for software design and do not adequately cover full life-cycle considerations. Moreover point solutions for specific tasks (e.g., reliability analysis, maintainability analysis, availability analysis, behavioral simulation, life cycle cost models) may not be well-integrated.

To achieve computer-aided improvements covering all of the above concerns, we need tools that are *additive*, have an *open architecture*, are *formally-based*, and are designed for *evolution support*. Tools that are additive allow users to gracefully fall into lowest common denominator (e.g., simple text editing) environments. Tools that have an open architecture can be tailored to special processes, empowered with domain-specific solutions, and can be easily extended as technology moves forward. Formally-based solutions allow for adequate expression of engineering constructs - terminology, behavior restrictions, interactions with the environment. In addition, formal approaches support requirements reuse, and can effectively produce secondary artifacts (e.g., simulations, trade-off analysis, test plans, documents) derivable from primary engineering constructs.

4 Conclusions and recommendations

While much of this work remains an art rather than a science, we are confident in making several specific recommendations.

1. Iterative Requirements Refinement In order to reduce risks, we recommend committing to iterative requirement refinement. The complexity of the systems we build today makes it almost impossible to acquire requirements in a vacuum. Ill-composed or misunderstood requirements may lead to specifications and implementations which do not match end-user needs. By encouraging iterative user/engineer interaction and by demonstrating system capability even during requirements analysis, we will develop systems that reflect real end-user needs. It is critical that we balance three things for successful iterations: using available design fragments (the "State-of-the-Shelf"), rapidly fabricating interfaces for gluing fragments together, and careful crafting of the requirements subset that is tackled in a given cycle.

2. Life Cycle Cost Awareness

   We recommend elevating engineering awareness of the impact on cost, schedule, and risk. The current practice often ignores these parameters as a form of simplifying assumption - get a functional description, then retrofit it to meet cost, schedule, risk constraints. This is the wrong way to simplify. Rather, byimposing these life cycle cost constraints early, we will reduce the design search space and avoids subsequent errors.

3. State-of-the-Shelf Approaches

   During the three to four years required to execute the conventional development processes, the end-users and engineers get locked into system paradigms that are continually based on emergent technology trends. When engineers respond to stringent system functionality, performance, and cost restrictions by targeting next generation state-of-the-art technology, they introduce tremendous uncertainty - inadequate tool and model support, unknown system capabilities, and poorly understood requirement interactions. In the past this may have been the only alternative. However, in today's environment, engineers need to seriously investigate the availability of on-the-shelf solutions. By considering cost/performance tradeoffs engineers will be opting for the on-the-shelf solutions.

4. Engineer-centered Tool Selection

   The way that many design organizations function also contributes to the cost and schedule risks of the conventional design process. Organizations may not maintain information on the current market trends of development support tools. Consequently, either the organization spends considerable time and effort up front selecting tools and technology, or it uses tools and technology that have less capability than required. We recommend empowering developers by making tool selection and investigation an intrinsic part of in-cycle developments.

5. Open Heterogeneous Environments

   A point solution provided today will not necessarily address the design issues faced in five years. A single initial vision will not likely accommodate a wide range of future entrants in the development technology arena. Design environments will need to be truly open and will need to support a rich mixture of heterogeneous tools.

6. Team Composition

   We recommend using narrow but deep problem decomposition to eliminate unnecessary communication burdens. RDG has experimented with this approach and found that functions can be handled by small independent groups who manage the evolving system description from requirements to implementation. Good interfaces on the developed pieces are critical so that the group can work independently of other teams. This approach avoids the error-prone "throw it over the wall" mentality that we often see in top down approaches.

7. Unified Test Procedures

   We recommend unified test procedures for all process phases (e.g., hardware development, software development, integration, production). Identification of real-world data, validation and verification questions and scenarios are critical system engineering products and should be created and linked to system engineering algorithm and architecture commitments.

8. Targeting Early Technology Transfer

   It very important to begin technology transfer early. Over the years, RDG has worked extensively with product line efforts within Lockheed Sanders to transition the lessons we have learned into company-wide process improvement strategies. As one illustration, we worked with a Lockheed Sanders component of the Patriot program to introduce rapid validation cycles into their methodology. Success was demonstrated later in the program. When hardware designers developed the delivered hardware, integrators were able to couple production-quality software with the real hardware in just a few days. Technology transfer occurs when technology receivers are motivated (e.g., they can not build a product with out the process) and they have confidence in the new processes (e.g., key people in an organization understand, or better, are engineers of new processes)

4.1 Summary

The road to achieving increased productivity and well-managed efforts follows evolutionary steps in which careful measurements on all of the above recommendations determine what works and what does not work. This view is important because without it, we miss the key ingredient: technologies (architectures, methodologies, environments) that grow with the changing technology base both now and into the future.

5 Acknowledgments

This article has grown out of the hard work of many technology developers and hardware/software implementors. We wish to acknowledge the contributions of Dr. Webster Dove and Dr. Cory Myers who have been instrumental in developing rapid development approaches. Many of the key aspects that we present follow from their comments and recommendations. Tony Martuscello provided the information on the AIPS example. Mark Lanciault is the chief designer for the RJD work. David Campagna, Ken Streeter, and Rob Costantino have made significant RDG contributions to numerous programs and have helped formulate many of the ideas which we present. The work on the ARIES project was co-developed with Information Sciences Institute under contracts F30602-85-C-0221 (Rome Laboratory sponsor) and F30602-89-C-0103 (Defense Advanced Research Projects sponsor). Specifically, we wish to acknowledge the contributions of Dr. W. Lewis Johnson (ARIES principal investigator), Dr. Martin Feather, and Kevin Benner. Other members of the ARIES team included Jay Runkel and Paul Werkowski at Lockheed Sanders. Additional technical direction was provided by Dr. Charles Rich who has made a significant impact on all of our knowledge-based initiatives.

References

[1] W. Allen, D. Rosenthal, and K Fiduk. The mcc cad framework methodology management system. *28th ACM/IEEE Design Automation Conference*, pages 694–698, 1991.

[2] M.R. Cutkosy, T.R. Gruber, R. S. Englemore, W.S. Mark, R.E. Fikes, J. M. Tenenbaum, M.R. Genesereth, and J. C. Weber. Pact: An experiment in integrating concurrent engineering systems. *Enterprise Integration Technology TR 92-02*, 1992.

[3] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. Report on a knowledge-based software assistant. In *Readings in Artificial Intelligence and Software Engineering.* Morgan Kaufmann, Los Altos, CA, 1986.

[4] K.E. Huff and V.R. Lesser. The GRAPPLE plan formalism. Technical Report 87-08, U. Mass. Department of Computer and Information Science, April 1987.

[5] W.L. Johnson, M.S. Feather, and D.R. Harris. Representation and presentation of requirements knowledge. *IEEE Transactions on Software Engineering*, 18(10):853–869, 1992.

[6] D.W. Knapp and A.C. Parker. Representation and presentation of requirements knowledge. *IEEE Transactions on Computer-Aided Design*, 10(7):829–846, 1991.

[7] C.W. Liew, L.I. Steinberg, and C.H. Tong. Use of feedback to control redesign. *Proceedings of the IFIP TC5/WG5.2 Working Conference on Intelligent Computer Aided Design*, 1991.

[8] R. Neches, R.E. Fikes, T. Finin, R. Gruber, R. Patil, T. Senator, and W.R. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3), 1991.

[9] J.Y.C. Pan and J.M. Tenenbaum. An intelligent agent framework for enterprise integration. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1391–1407, 1991.

[10] Malone T. W., Grant K. R., Lai K., Rao R., and Rosenblitt D. Semistructured messages are surprisingly useful for computer-supported coordination. In Irene Greif, editor, *Computer-Supported Cooperative Work: A Book of Readings*, pages 311–334. Morgan Kaufmann, Los Altos, CA, 1988.

[11] R. Wesson, F. Hayes-Roth, J.W. Burge, C. Stasz, and C.A. Sunshine. Network structures for distributed situation assessment. In *Readings in Distributed Artificial Intelligence*, pages 71–89. Morgan Kaufmann, 1988.