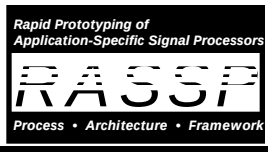


Rapid Prototyping of Application-Specific Signal Processors (RASSP)

RASSP Methodology

Version 2.0

Volume 1

October 1995

ADVANCED TECHNOLOGY LABORATORIES

LOCKHEED MARTIN

LOCKHEED MARTIN

Rapid Prototyping of Application-Specific Signal Processors (RASSP)

RASSP Methodology

Version 2.0

Volume 1

Submitted By:

Lockheed Martin RASSP Team
Lockheed Martin Advanced Technology Laboratories
Bldg. A&E 2W
1 Federal Street
Camden, New Jersey 08102

Contract Number:

DAAL01-93-C-3380

Date:

October 1995

Table of Contents

List of Illustrations

List of Tables

Section One
Overview

The RASSP Methodology is organized into five basic sections. This section summarizes the overall methodology from a perspective that spans individual processes. The individual process areas on RASSP are program planning, the design process, manufacturing and test, and specialty engineering. Section 2 describes the project planning phase of the program. Section 3 is the focal point of the document; it describes the overall design process for RASSP. Sections 4 and 5 describe the subsequent manufacture and test, and specialty engineering ('ilities, etc.) disciplines, respectively, and how they interface to the design process.

1.1 Objective and Scope

The goal of the Rapid Prototyping of Application-Specific Signal Processors (RASSP) program is to improve by (at least) a factor of four the time required to conceptualize, design/upgrade, and field signal processor designs, with similar improvements in design quality and life-cycle cost. As shown in Figure 1-1, to achieve the RASSP goals requires a "technology triad" of enhancements in signal processor architecture, design methodology, and an integrated enterprise-wide development environment. This paper describes the RASSP design methodology, which is the cornerstone for defining the requirements that will drive the Model Year Architecture and design automation portions of the program.

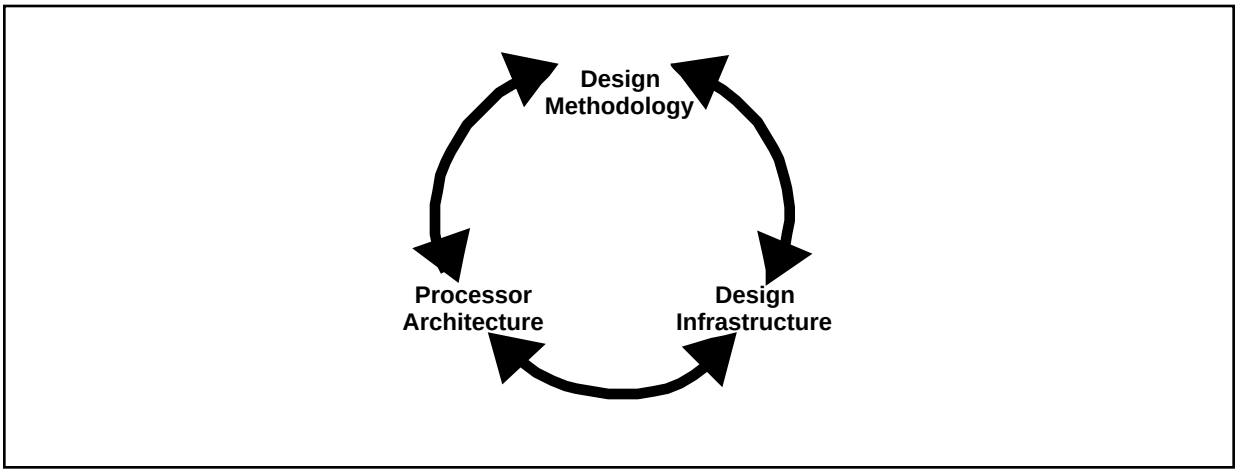


Figure 1-1. RASSP technology triad.

This section overviews the RASSP objectives and scope of the methodology; Section 1.2 describes the RASSP methodology as both a series of functional processes and a set of unifying processes that span all functional areas; and Section 3 summarizes the application and implementation of the methodology.

The overall RASSP methodology is based upon two major beliefs: 1) that concurrent design practices using an integrated approach to hierarchical design verification are required to improve design quality and performance; and 2) that dramatic (>4X) decreases in cycle time can only be achieved by maximizing reuse of both hardware and software elements. These two beliefs lead to the notion of design using modular hardware and software functions that are represented in a reuse library. To significantly improve the cycle time and cost of developing signal processing systems, major changes are required in the traditional process—particularly in the areas of processor architecture and software design, which both hold promise for the largest improvements.

In addition to speeding up the individual processes, RASSP will reduce a portion of the cycle time by eliminating time-consuming and costly design rework cycles. The RASSP process must provide a method

to support high-quality designs that lead to first-pass success of all hardware and software elements.

Scope of RASSP

The scope of the RASSP methodology effort focuses on the design process and its *interfaces* to the other processes in the production cycle to support seamless interaction with these areas. It is not within the scope of RASSP to redefine new methods of manufacturing, test, and specialty engineering. This has enabled the methodology efforts to focus on signal processor design and implementation using an enterprise-wide framework that enables concurrent engineering by integrating designers to the other product disciplines (manufacturing, test, etc.).

The RASSP program also focuses on the development of signal processors. In the large (platform-level) system view of the world, this effort focuses on the signal processing subsystem level. While many of the steps in the RASSP system process are hierarchical and are applicable to larger systems as well, the definition of "system" in the RASSP context refers to the signal processor subsystem, as shown in Figure 1-2.

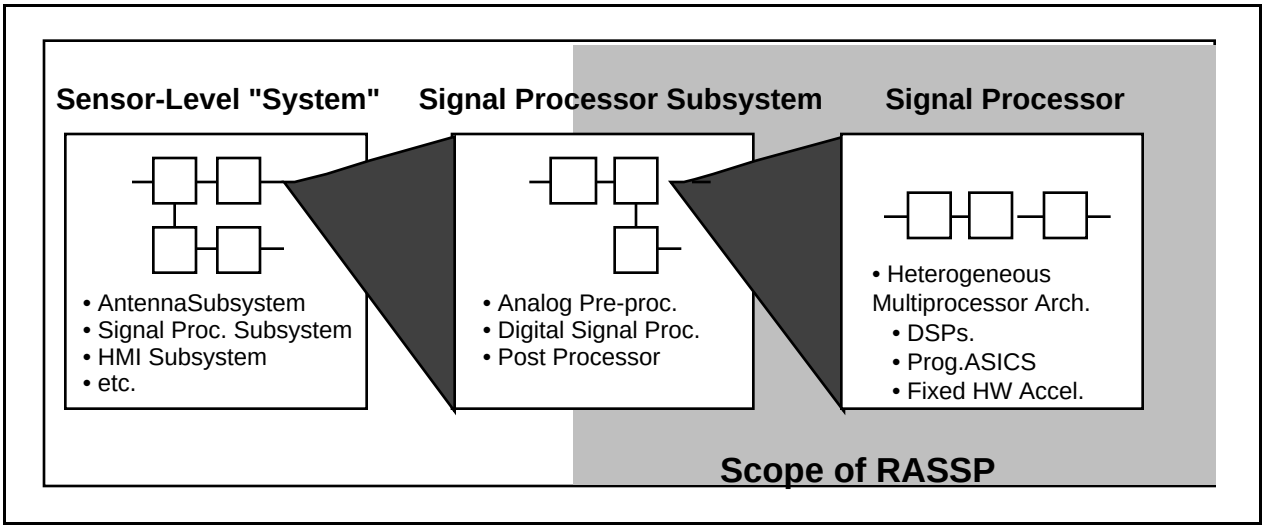


Figure 1-2. RASSP system definition.

The RASSP design methodology provides for iteration of signal processor requirements in the context of the overall system to support trade-offs between signal processor design alternatives and parent system requirements. The functional requirements may include both procedural requirements (operations to be performed) and functional constraints (driven by performance requirements). Requirements in the form of signal processor subsystem constraints, such as form factor, power consumption, thermal characteristics, and cost, are assumed to be an input to the RASSP process. Inputs to the RASSP embedded digital signal processor (DSP) are digital data streams, typically from a sensor or another special-purpose digital processor, and digital

control signals from other digital subsystems. Outputs from the RASSP DSP are digital data streams and control signals to the other subsystems.

1.2 RASSP Development Process

We leveraged the methodology developed on the Martin Marietta Engineering Process Improvement (EPI) program as a starting point on RASSP. The EPI program represents an investment of >$150M to develop and instantiate engineering Best Practices within Martin Marietta. The RASSP effort is enhancing these developments to support rapid prototyping, as shown in Figure 1-3. The program includes:

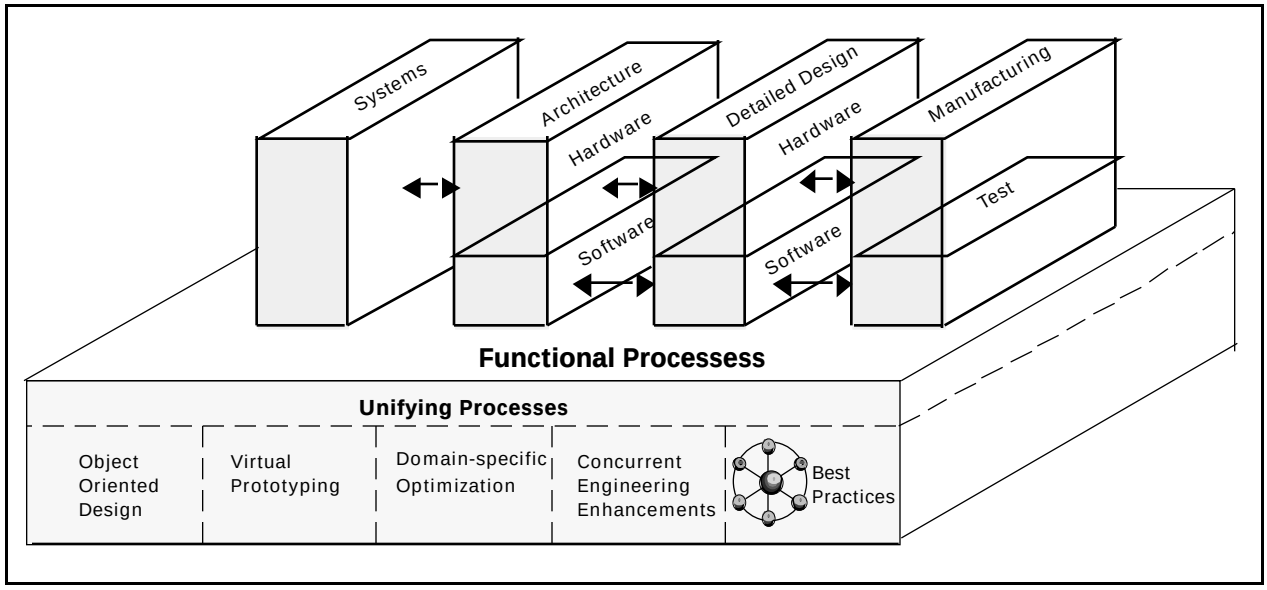


Figure 1-3. RASSP methodology composed of functional and unifying processes.

- A new set of functional processes that are composed of functions optimized to support rapid signal processor design and deployment. This includes creating a new architecture process that supports hardware/software codesign and concurrency between the systems and detailed design processes.

- Additional unifying processes applied across *all* functions to support design efficiency and reuse (best practices). These include:

  - Signal processor-specific optimization to reduce development time
  - Object-oriented design approaches to enhance reuse
  - Concurrent engineering enhancements (within tasks and across tasks), implemented using Integrated Product Development Teams (IPDT) via the RASSP enterprise system
  - Virtual prototyping through the application of a spiral methodology to provide iterative, risk driven designs
  - Maximum use of data exchange and interface standards to support efficient design upgrades of both the RASSP Model Year Architecture and the enterprise system.

A number of methodology elements apply across the entire product life cycle, such as application of a spiral methodology, the concurrent engineering elements of the program, and the Model Year Architecture approach. Section 1.2.1 overviews these elements. The functional processes shown in Figure 1-3 are described in Section 1.2.2.

1.2.1 Unifying Processes

Iterative Virtual Prototype *(spiral model)*

The goals of the RASSP program, which include rapid prototyping combined with continual process and product improvements, are more applicable to an iterative, spiral model than the traditional waterfall model. The spiral model was originally directed at supporting the software development process. Since the RASSP scope includes more than just software, adaptation of the model is appropriate. A view of the spiral model, as adapted for RASSP, is shown in
Figure 1-4.

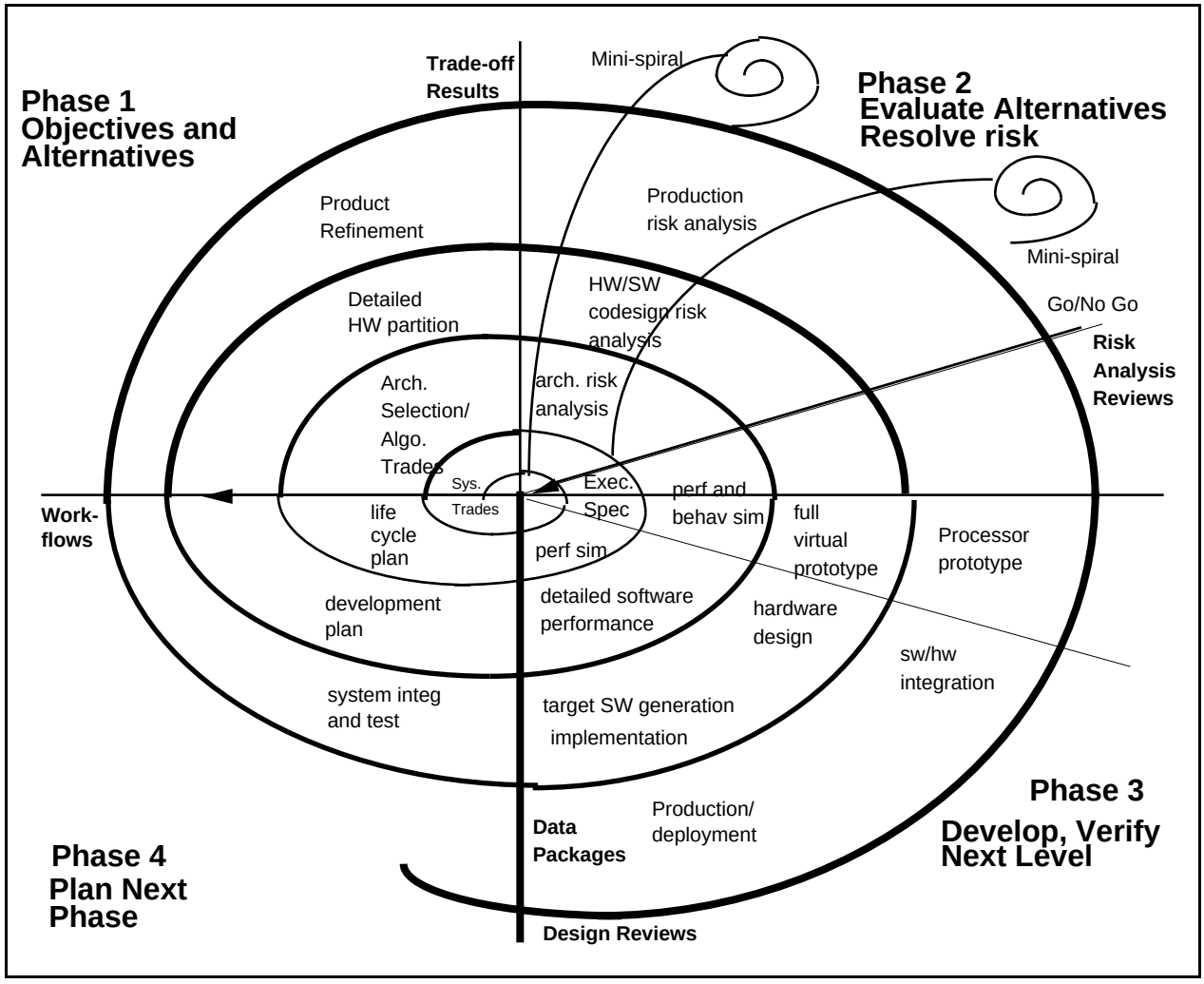


Figure 1-4. RASSP spiral model.

Four phases are associated with each major cycle of the spiral. In the first phase, the baseline approach and appropriate alternatives are developed to meet program objectives. During the second phase, the approaches are evaluated against the objectives and alternatives, and the risk associated with these approaches is evaluated. During the third phase, the prototype is evaluated and the next level of the product is developed. This phase results in a prototype of the design. In the fourth phase, the product is reviewed and plans for the next development stage are established. This entire process is repeated to the next level of detail by repeating each of the four phases; hierarchical application of the process results in an interactive, risk-driven approach to rapid prototyping.

In the traditional spiral model for software, each cycle might be a full set of software at some level of functionality and/or maturity, or might have critical portions of functionality prototyped (with others

"stubbed" out for implementation in subsequent cycles). RASSP applies the concept of *virtual prototypes*, or simulated versions of the signal processor (encompassing hardware/software functionality and performance) at each cycle of the spiral. During each spiral iteration, a new virtual prototype emerges—from executable specification to high-level behavioral prototype, and then to full functional and timing versions of the design. The design then transitions into a full hardware prototype and matures into a design for full-scale production. Risk analysis and design reviews provide decision points within the cycle and enable backtracking to evaluate alternatives. New spirals are spawned to reduce high-risk elements or continue the design as planned. This model represents a more temporal overlay to the RASSP functional processes and supports concurrency (overlap) between functional disciplines.

In the RASSP methodology we hierarchically apply the spiral process. Each data package may be the result of a series of mini-spirals, while also being part of the primary set of spirals within the design. As we identify high-risk elements in the design process, "mini-spirals" can be spawned to develop critical items that are "long poles" in the design schedule. Note that this approach allows development across processes as a function of time [e.g., we can perform architecture or detailed design during the early stages (systems process) for high-risk portions of the design]. The logical consequence of the process is that each successive virtual prototype has more complete functionality than its predecessor.

An expanding information view of the spiral process is shown in Figure 1-5. The major spiral cycles correspond to the iterations of a virtual prototype associated with the overall signal processor. The mini-spiral cycles correspond to the system, architecture, and detailed design processes associated with portions of the design and/or models (e.g. custom processor, MCM, new hardware model, new software primitive, etc.). This view corresponds to the idea that, based upon risk, pieces of the overall design may be at differing levels of maturity. At the same time, there must be a minimum maturity level achieved and data package produced for the overall signal processor to proceed from one major spiral cycle to the next. The data packages form the basis for the design reviews associated with each major spiral; these are used to drive the next iteration of the design. The data packages are inputs to (and become a subset of) the next stage of the process.

The concept of the virtual prototype as an expanding information model of the processor maps well into this paradigm. At all steps of the design, the model is characterized by four major elements—workflows, requirements, executable models, and test benches. The four elements are summarized as follows:

1) Workflows — The steps used to develop the current/next phase of the development. This is the result of the fourth phase of the spiral model.

2) Requirements — An appropriate set of parameters specifying the performance and implementation goals for the processor (size, weight, power, cost, schedule, etc.).

3) Executable Models — The functional and structural definition of the processor under development. VHDL is being heavily emphasized to specify functionality throughout the design process on RASSP. Once hardware/software is allocated, we will specify software functionality using High-Order Language (HOL) programming languages (C and/or ADA).

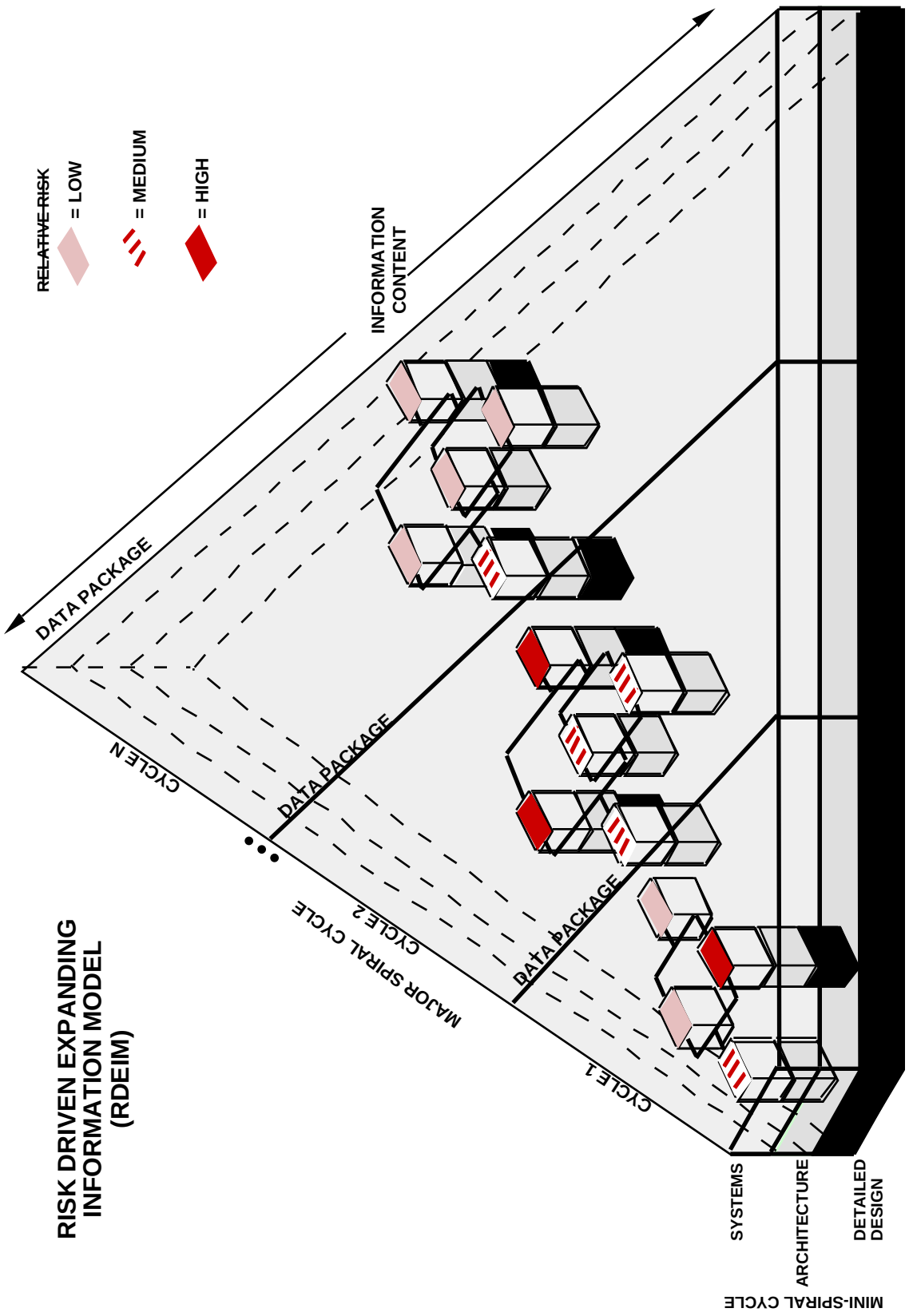


Figure 1-5. Risk-driven expanding information view of RASSP design process.

4) Test Benches — An appropriate set of test vectors, data sets, etc. that fully verify the functionality and performance of the model. We intend to use VHDL (and the corresponding WAVES standard) to the greatest extent possible for RASSP test benches.

Concurrent Engineering

Concurrent engineering includes elements applied across all tasks throughout the product cycle; these include the roles and use of IPDTs and the Best Practices used by the team. In addition, we use the program planning and review processes to ensure that the proper cross-fertilization occurs on a formal basis, to assess program risk, and to identify appropriate retirement actions. Concurrent engineering occurs both within and between tasks, resulting in a set of overlapping collaborative design efforts optimized to support rapid prototyping.

Integrated Product Development Team (IPDT)

We are implementing concurrent engineering on the RASSP program using IPDTs. In the large systems context, an IPDT oversees development of an entire product, such as a radar system, with the signal processor being one of the products that are integrated by the IPDT. The Signal Processor Subsystem (SPS) Product Development Team would thus oversee RASSP's scope, including interaction with both the other PDTs and the IPDT.

The IPDT, under the leadership of the program manager, has overall responsibility for design, analysis, development, fabrication, and evaluation of the product. It coordinates tasks assigned to the PDTs, and its members are responsible for specific overall systems engineering and program management tasks not delegated to PDTs. The roles of the IPDT members are focused in three areas: program management, technical design integration, and functional specialty integration.

The IPDT is composed of a set of PDTs. Each PDT is composed of a set of skills, or core competencies, for the specific product. These skills usually include engineering, manufacturing, test, and management and they can be located across companies or sites. The RASSP concept of the enterprise system supports distributed IPDTs/PDTs by providing seamless electronic access to data and enabling realtime design interaction.

Product Development Teams (PDT)

PDTs partition the design and development task into manageable efforts that can be accomplished by multifunctional teams of reasonable size. The task partitioning is based on the derived system-level architecture and the subsystem partitioning, and it is optimized on a per program basis. Each team is responsible for accomplishing assigned design/development, fabrication, and validation tasks, and each is provided design-to-cost budgets to guide their development. The key responsibilities and tasks of the PDT members are summarized in
Table 1-1.

Each member of the PDT is responsible for implementing a set of Best Practices that is refined throughout the program. The practices identified to date are listed below; all are oriented to support quality improvements and design/cost efficiency:

- Design reviews
- Design to cost
- Design for test
- Requirements management
- Central engineering services
- Standard methodology and tools
- Standard parts
- Product standardization
- Product release packages/manufacturing interface

Table 1-1. Integrated Product Development Team member roles.

Member	Role in Signal Processing Subsystem PDT
Team Leader	Provide PDT leadership and integration of the multifunctional discipline activities to produce the signal processing system that meets performance requirements at the lowest life cycle cost
Systems Engineer	Monitor and interpret specification requirements; lead system tradeoff studies and system design verification
Architecture Engineer	Interpret subsystem requirements; lead architecture trade studies, HW/SW codesign, and architecture verification
Digital Engineer	Digital requirements analysis; support architecture trade studies; participate in architecture verification; design, build and test hardware
Software Engineer	Software requirements analysis; participate in HW/SW codesign; design, code, and test library elements; participate in integration and test
Mechanical Engineer	Support architecture trade studies; develop chassis; frame design; perform thermal analysis, packaging trade studies
Manufacturing	Identify special processes that may be required for production build
Test and Maintenance	Develop test philosophies to optimize testability and maintainability and conduct development testing for prototype
Producibility	Support trade studies for part selection/approval and maintain parts library; assess candidate designs and verify that the final design meets requirements for RAM, safety, EMI, Tempest, and life cycle cost
Sourcing	Support trade studies and design by identifying vendors that are able to provide high quality, reasonably priced materials in a timely manner
ILS	Assess the consequence of design decisions on ILS and total system costs
Cost Analyst	Support trade studies by providing cost impact of implementation technology; support life cycle cost analysis throughout design cycle

In addition, the Martin Marietta team is internally evaluating new concepts, such as object-oriented analysis and design techniques applied to the entire signal processor design. These results will be made available and incorporated into the RASSP methodology, if appropriate.

In contrast to today's highly sequential processes, Figure 1-6 shows the concurrent RASSP process over the same product development phases. Note that *all* process disciplines participate throughout the product development, as shown in the pie charts. The new architecture process provides for true hardware/software codesign of the signal processor. The overlap of disciplines in the RASSP systems, architecture, and detailed design processes implies that as the signal processor design progresses, there is free interchange among disciplines. This includes the opportunity for modification and iteration of adjacent processes (e.g., the systems process can be modified by results of the architecture process, etc.). There is no attempt in this figure to represent elapsed time throughout the design cycle.

Role of Model Year Architecture

To enable design reuse and minimize upgrade costs, a set of common hardware and software architecture elements is required. A key element to implement this methodology is the selection of a Model Year Architectural approach (both hardware and software) that adheres to the following principles:

- The architectures must be open, promoting hardware/software upgradability and reusability in other applications
- The architectures must use emerging, state-of-the-art commercial technology whenever possible
- The architectures must support a range of applications to maintain low non-recurring engineering costs
- The architectures must facilitate continuous product improvement

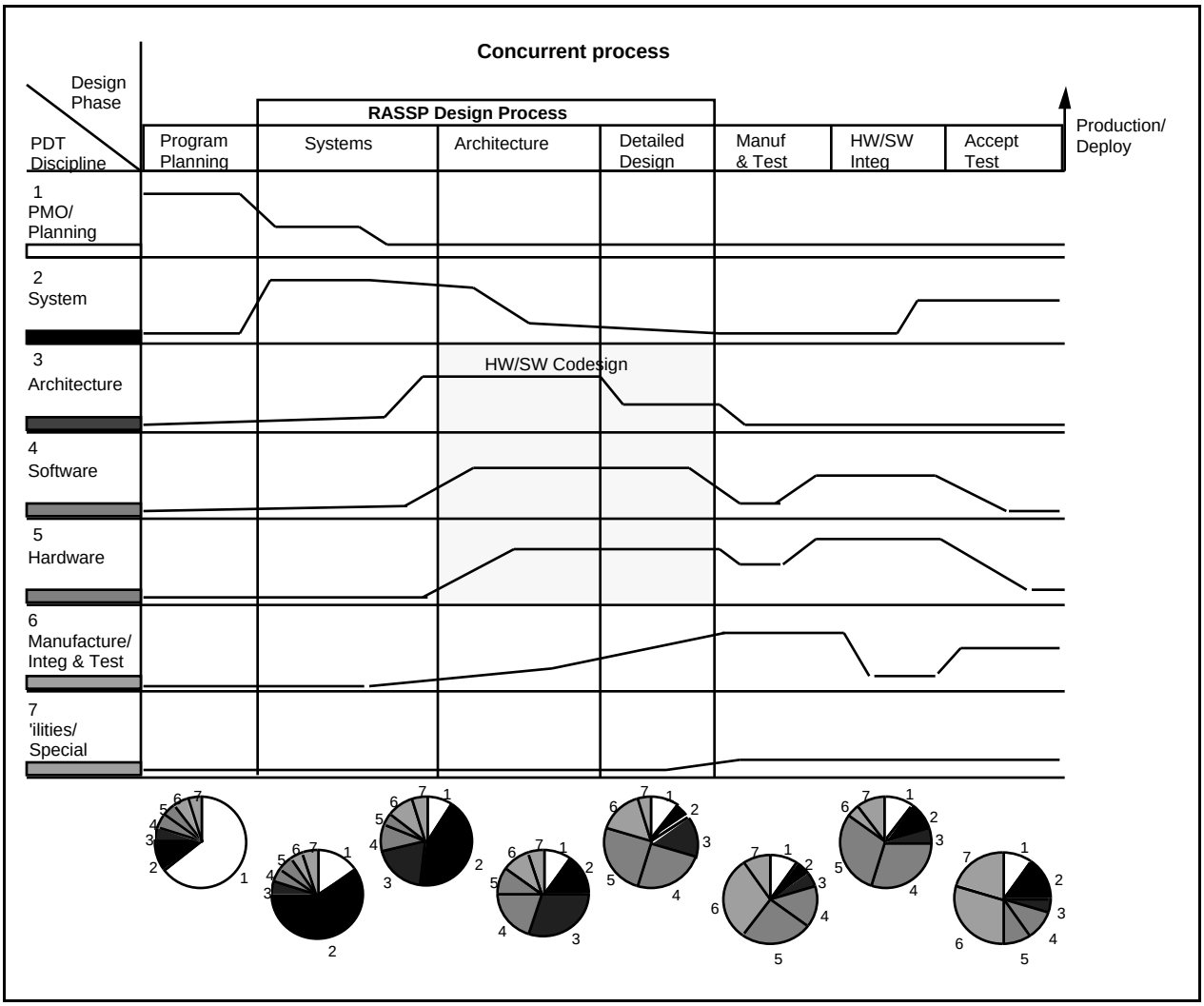


Figure 1-6. Roles of PDT in concurrent engineering process.

The Martin Marietta approach to implementing the Model Year Architecture is based on modular, scalable architectures that use *functional* standard interfaces. This approach strives to move beyond today's approach of standardizing on physical interfaces; this approach is good, but it does not go far enough to ensure technology independence. Physical interfaces encompass the lower levels of the ISO/OSI protocol hierarchy and are specific to the media and operating conditions (voltage, timing, etc.) specified by these layers. Functional interfaces provide the data-transaction-level capabilities to support communications, independent of the underlying physical technology. By standardizing on functional interfaces, we can maximize independence from technology (electrical versus optical) and specific hardware versus software (processor-based versus dedicated hardware) approaches. We will apply this approach to a number of signal processing interfaces required for RASSP, as shown in Figure 1-7. The RASSP team has developed a full description of this approach, which is described in the RASSP Model Year Architecture Working Document Version 1.0, October 28, 1994.

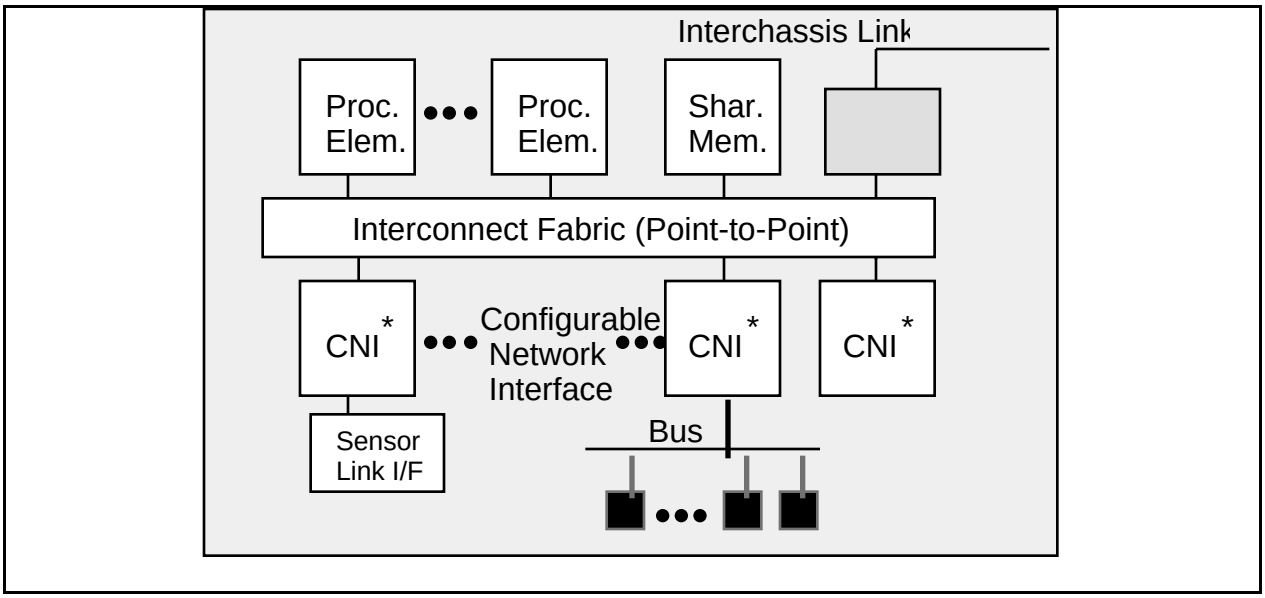


Figure 1-7. RASSP Model Year Architecture.

The RASSP Model Year Architecture(s) must be supported by the necessary library models to facilitate trade-offs and optimizations for specific applications. Reusable hardware and software libraries facilitate growth and enhancement in direct support of the RASSP Model Year concept. The hardware and software components within the library are encapsulated by the functional wrappers required to enable efficient integration and use. We used the Model Year Architecture methodology to develop all elements for inclusion and use within the reuse libraries. This is key to attaining the 4X cycle-time improvements on RASSP. As technology advances, new architectural elements may be included in the library. Rapid insertion of a new element into an existing RASSP-generated design is the goal of the Model Year concept.

Role of Standards

Standards play an important part in all aspects of the RASSP technology triad as shown in Table 1-2. We are implementing the methodology as workflows using the International Data Exchange Format (IDEF). As part of the Model Year Architecture effort, we are defining functional standard interfaces to facilitate technology independence. We are using VHDL to specify architecture/hardware functionality, and HOLs such as C and/or ADA to specify software functionality.

In addition, we are using data exchange-based standards, such as PDES/STEP [Product Data Exchange using STEP(Standard for Exchange of Product Model Data)] and Express, the information modeling language, to support product modeling and automated electronic transfer of product data to manufacturing.

Table 1-2. Use of standards-based models on RASSP.

RASSP Technology Triad	Approach	Standard Information Models	Payoff
Methodology	• Concurrent/collaborative product development • Full product verification prior to manufacturing • Hardware/software codesign • Re-use-based design	Workflows — IDEF	• Reduced time to market • First-pass success • Optimized solutions
Model Year Architecture	• Architecture standardizes functional interfaces • Encapsulate elements into "plug 'n play" re-use library • Exploit COTS processing technology	Architecture/ Hardware – VHDL, Software – ADA/C	• Increased design re-use • Reduced development cost • Low-cost upgrades – minimum hardware/ software breakage
Automation Infrastructure (Enterprise)	• Use standards-based product data modeling/ interchange • Exploit electronic highway • Distributed data management	Product data flow – PDES/STEP, Express	• Seamless data transfer • Higher efficiency • Concurrent product development

1.2.2 RASSP Product Development Process

The overall RASSP development process, shown in Figure 1-8, has four major elements:

- Project Planning/Management — Project planning/management provides the first step of the process: develop the technical plan, and form the PDT and the overall management approach for the specific signal processor development.

- Design Process — The design process represents development of the signal processor design from requirements capture through release to manufacturing. It has three major processes: systems design, architecture design, and detailed design. This process takes customer requirements and results in a fully verified (functionality and performance) virtual prototype of the signal processor.

- Manufacturing/I&T Processes — The manufacturing, integration and test processes define for RASSP the interface of design engineering to the disciplines required to build and test signal processor prototypes. This means the interaction of manufacturing and test personnel throughout the design cycle to support concurrent engineering through the release of the virtual prototype to manufacturing. This process also supports subsequent testing of the manufactured design and hardware/software integration of the signal processor.

- Specialty Engineering ('ilities, etc.) Process — Specialty engineering covers all the non-design engineering tasks required to successfully develop and field RASSP products. These include reliability, maintainability, parts, and EMI engineering (where applicable), as well as integrated logistic support engineering and services.

The RASSP team is focusing the majority of its efforts on improving the design portion of the process, which is more fully discussed in the following paragraphs.

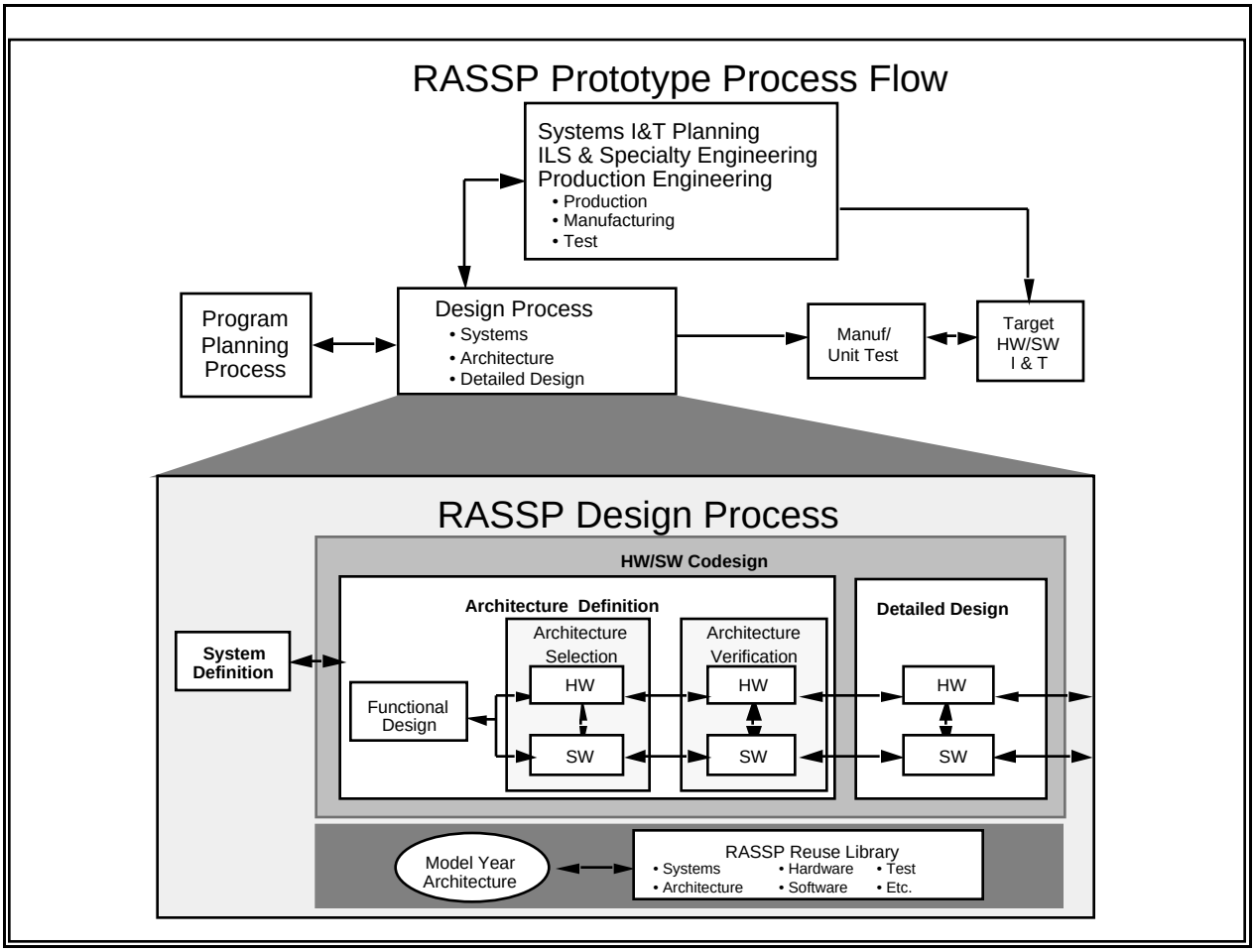


Figure 1-8. RASSP functional design process.

1.2.2.1 RASSP Design Process

The RASSP design process has three major functional processes—the systems, architecture, and detailed design processes; it is shown in expanded form in Figure 1-8. We partition the process as a function of the abstraction level of the evolving design, not as a function of the discipline. This is a result of merging hardware and software into a true codesign process; any distinctions between hardware and software are made within the specific process. Hardware/software codesign is implemented from the initial partitioning of functions to hardware and software elements all the way to manufacturing release. At each step in the hierarchy, interactive simulation using hardware and software models is performed at equivalent levels of abstraction to verify both functionality and performance. This means that each process area is closely tied to the RASSP vision of an iterative (spiral-like) development, resulting in a series of virtual prototypes and data packages.

Each major cycle of the spiral process represents an iteration of a virtual prototype. Within each prototype iteration, pieces of the design can and most likely will be at different levels of maturity, as shown in Figure 1-5. Each piece of the design may be represented by a mini-spiral where the spiral cycles correspond to virtual prototypes of the piece. Consequently, for each major spiral cycle, there may be activity in the system, architecture, and detailed design processes.

Systems Process

The system process captures customer requirements and converts these system-level needs into processing requirements (functional and performance). We perform functional and performance analyses to properly decompose the system-level description. The system process has no notion of either hardware versus software functionality or processor implementation.

The first major cycle of the spiral shown in Figure 1-4 results in a requirements specification that has been captured in an appropriate tool; it is the first instantiation of a virtual prototype (VP0). This information is then translated into simulatable functions, which we refer to on RASSP as an *executable specification*. This cycle represents the first level at which requirements are specified so that we can readily match to simulators to verify performance and functionality automatically. In this phase, processing time is allocated to functions and functional behavior is defined in the form of algorithms that are executable. At this point, all functions are implementation-independent. High-risk items can spawn prototype analysis and development efforts in a mini-spiral. The executable specification represents a major portion of the systems design information model. In addition, diagnostic concepts and test requirements are developed along with subsystem test plans and test benches..

A major portion of the systems behavioral definition will be in terms of algorithmic functionality, which can be decomposed into functional and behavioral stages. Functional algorithm development deals with the logic and mathematics of the desired functionality and is not within the scope of RASSP. Behavioral algorithm development is concerned with mapping functional algorithms onto implementation-specific architectures and is within the scope of RASSP. This stage of development does not change the fundamental logic and mathematics, though it may perturb the specific selection and sequence of operations. Additionally, issues such as finite word length effects and scaling strategies are addressed. A Systems Design Review (SDR) is conducted when the executable specification is completed and all other requirements have been defined.

Architecture Process

The architecture process transforms processing requirements into a candidate architecture of hardware and software elements. Architecture selection, which corresponds to the second virtual prototype iteration (VP1) of the signal processor system, initiates the trade-offs between the different processor architecture alternatives. During this process, the system-level processing requirements are allocated to hardware and/or software functions. The hardware and software functions are verified with each other via "co-verification" at all steps. The architecture verification process, corresponding to the next virtual prototype (VP2), results in a detailed behavioral description of the processor hardware and the definition of the software required for each processor in the system. The intent is to verify all the code during this portion of the design process, ensuring hardware/software interoperability early in the design process.

Processor behavioral and performance simulations support trade-offs. Mixed levels of simulation (algorithm, abstract behavioral, performance, ISA, RTL, etc.) are supported to verify interaction of the hardware and software. These models are largely hierarchical VHDL models of the architecture. We choose the models, to the maximum extent possible, from the Model Year Architecture elements in the RASSP reuse library. The RASSP team develops and inserts new, required library elements into the reuse library to support this design phase. The executable specification has now evolved into a more detailed set of functional and performance models that are architecture-specific. Software algorithm implementations are now specific to the candidate architecture(s). We further define test and diagnostic strategies and develop a built-in test (BIT) architecture along with prototype test program sets. An Architecture Design Review (ADR) is conducted when the architectural trades are completed and the design has been verified to a high degree of confidence.

The software portion of the architecture process deviates significantly from traditional (functional decomposition 2167A) approaches by formalizing reuse and emphasizing DFG-driven autocode generation. The partitioned software functionality can be broken into three major areas: 1) algorithm, as specified in a flow graph; 2) scheduling, communications, and execution, as specified by mapping the

graph to a specific architecture; and 3) general command/control software and other non-DFG based software. The intent of RASSP is to automate the first two to the maximum extent possible. This will be accomplished using a graph-based programming approach(es) that supports *correct-by-construction* software development based on algorithm and architecture-specific support library elements.

We will perform traditional general command/control and other non-DFG based coding using emerging, CASE-based code development, documentation, and verification tools. This includes generating new library elements, which are first entered into the system as *prototype* elements and later promoted to *verified* elements subsequent to prototype verification. The library generation process supports generating, testing, and documenting all new software, which then becomes part of the RASSP component library.

Detailed Design

The next virtual prototype iteration (VP3) involves the detailed design of software and hardware elements. As with the prior processes, we carry out and verify design for both hardware and software via a set of detailed functional and performance simulations. When this process is complete, the design is established, resulting in a fully verified virtual prototype of the system.

During the hardware portion of the detailed design process, we transform behavioral specifications of the processor into detailed designs (RTL and/or logic-level) through a combination of hardware partitioning, parts selection, and synthesis. Detailed designs are functionally verified using integrated simulators, and performance/timing is also verified to ensure proper performance. The process results in detailed hardware layouts and artwork, net lists, and test vectors that can then be seamlessly transitioned to manufacturing and test via format conversion of the data. The entire design package required for release to manufacturing is generated for review at the Detailed Design Review (DDR), which corresponds closely to today's Critical Design Reviews (CDRs).

Since we verify most of the software developments during the architecture phase, they are limited at this point to generation of those elements that are target-specific. This includes configuration files, bootstrap and download code, target-specific test code, etc. All the software is compiled and verified (to the extent possible) on the final virtual prototype prior to the detailed design review. Design release to manufacturing marks the end of the RASSP design process.

Prototype Manufacturing, Integration, and Test

After DDR, data developed as part of the DDR drives the software and hardware builds. Hardware designs have been released to manufacturing and have been built and unit tested. The last part of this phase is the functional and performance testing of the overall prototype using test suites that have already been developed and tested on the behavioral simulations.

1.2.2.2 Hardware/Software Codesign in RASSP Methodology

Hardware/software codesign is the joint development and verification of hardware and software through the use of simulation and/or emulation; it begins with initial partitioning and proceeds through to design release. The principal benefits of hardware/software codesign are:

- Mutual Influence of Both Hardware and Software Early in the Design Cycle — Software performance becomes one of the criteria for selecting an architecture, rather than the more traditional approach of selecting the architecture and forcing the software to fit.

- Continual Verification Throughout the Design Cycle — As the design progresses through subsequent levels of detail, both hardware and software are continually verified to improve design quality. This minimizes iterations after the design is released to manufacturing.

- Enables Evaluation of Larger Design Trade Space — Interoperability of tools and automation of codesign early in the architecture process significantly improves the ability to consider designs which otherwise may be ignored.

- Reduces Integration and Test — Since hardware and software have been co-verified throughout the design process, integration and test will be greatly simplified. Test and diagnostics are developed up-front, while there is still the opportunity to impact the design for testability.

The RASSP design process is based on true hardware/software codesign and is no longer partitioned by discipline (e.g. hardware and software), but rather by the levels of abstraction represented in the system, architecture, and detailed design processes. Figure 1-9 shows the RASSP methodology as a library-based process that transitions from architecture independence to architecture dependence after the systems process.

In the systems process, processing requirements are modeled in an architecture-independent manner. Processing flows are developed for each operational mode and performance timelines are allocated based upon system requirements. Since this level of design abstraction is totally architecture-independent, hardware/software codesign is not an issue.

In the architecture process, the processing flows are translated to a standard data and control flow graph description for subsequent processes. The processing described by the nodes in the data flow graph are allocated to either hardware or software as part of the definition of candidate architectures. This becomes the transition to architecture dependence. Requirements and tasks for non-DFG based software and the corresponding software are developed to the maximum extent possible.

We analyze the hardware/software allocation via modeling of the software performance on a candidate architecture through both software and hardware performance models. For architectures selected for further consideration, hierarchical verification is performed using finer grain modeling at the ISA level and below.

During the detailed design process downloadable, executable application and test code is verified to the maximum extent possible.

Reuse library support is an important part of the overall process. The generation of both hardware and software models is supported in the overall methodology. Software models are validated using the appropriate test data. Hardware models are validated using existing validated software models. Both hardware and software models are iterated jointly throughout the design process.

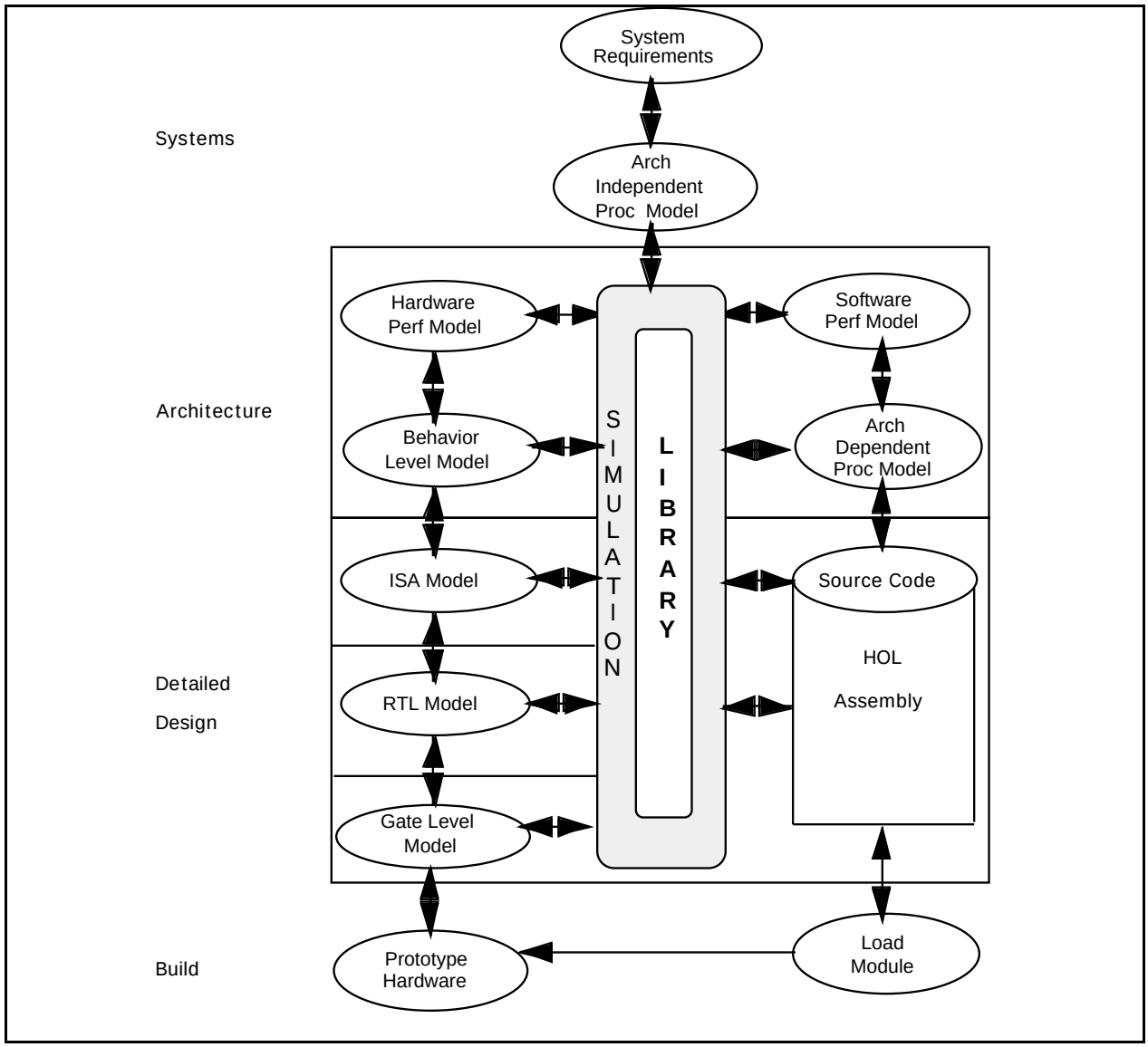


Figure 1-9. Hardware/software codesign in RASSP design process.

Simulation is an integral part of hardware/software codesign. Figure 1-10 shows a top-level view of the overall simulation philosophy in the RASSP methodology. During the systems process, functional simulation is performed to establish a functional baseline for the signal processor.

During the architecture process, we perform various simulations at differing levels of detail as the design progresses. Early in the process, performance simulations are executed using high-level models of both hardware and software from the reuse library. Software is modeled as execution time equations for the various processors in the architecture. Models at the behavioral level, for both processing elements and communication elements, are used to describe the architecture. This level of performance simulation enables rapid analysis of a broad range of architectural candidates composed of various combinations of COTS processors, custom processors, and special-purpose ASICs. In addition, we can evaluate many approaches to partitioning the software for execution of the architecture.

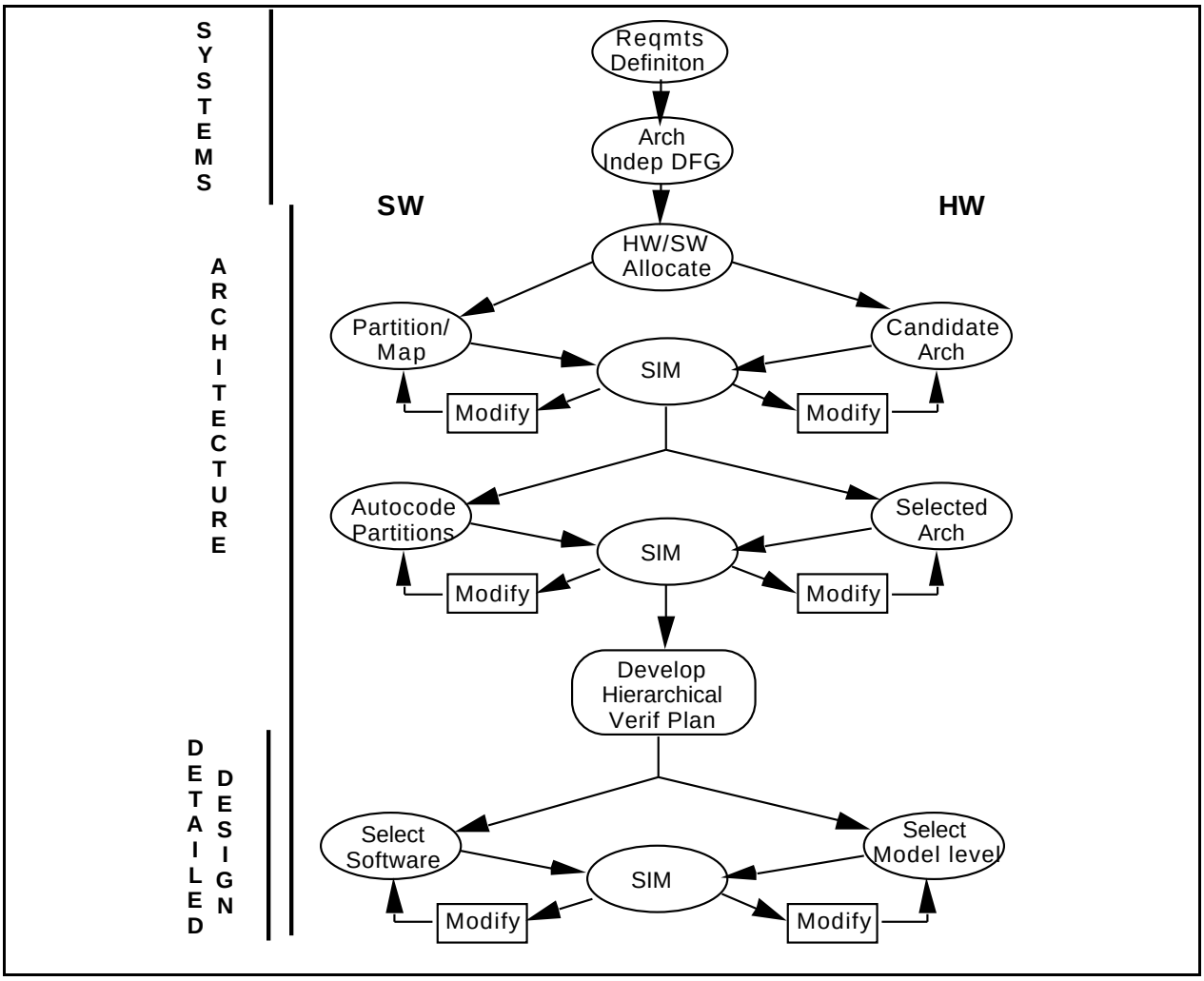


Figure 1-10. RASSP simulation philosophy by design process.

As the architecture process progresses, each graph partition is translated into a software module for execution on a specific processor in the architecture. Functional simulation is used to verify that the generated code is consistent with the functional baseline. Performance simulation using lower-level models, which include the operating system, scheduling, and support software characteristics, provide the next level of assurance that all throughput requirements are met. Source code for non-DFG based software is developed and tested to the maximum extent possible. Finally, hierarchical architecture verification of the architecture is established using selective performance and functional simulation at the ISA and/or RTL level. The goal is to ensure that all architectural interfaces have been verified.

In the detailed design process, we again perform selective performance and full functional simulation. At this point, however, the design has progressed to where simulation at the RTL and logic-levels is most appropriate. Verification of the designs at this level is necessary before release to manufacturing. It is important to note that pieces of the design may be in different stages of the overall process based upon the risk analysis performed in each development cycle. For example, if it is obvious to the designers during systems analysis that a new custom hardware processor will be required to meet the requirements, the design of the custom processor may be accelerated while the overall signal processor design is still in the architecture process. This approach corresponds to the mini-spirals described in Section 1.2.1.

1.2.2.3 Design For Testability in the RASSP Methodology

The Design For Testability (DFT) steps within the overall methodology enables designers to create systems that can be cost-effectively tested throughout their life cycle. Designs that adhere to the

methodology are made testable on the basis of various design for testability (DFT) and built-in-self-test (BIST) techniques. The methodology covers various aspects of test and diagnosis at the chip, MCM, board and system levels, including test requirements capture; test strategy development; DFT and BIST architecture development; DFT and BIST design and insertion; test pattern generation; test pattern evaluation; and test application and control. The methodology provides the designer with a process for introducing testability requirements and constraints early in the design cycle and for addressing DFT and BIST issues hierarchically at the chip, multichip module (MCM), board, and system levels. The payback for early testability emphasis includes lower test cost throughout the life cycle of the product, reduced design cycle time, improved system quality, and enhanced system availability and maintainability.

Elaboration of the DFT steps within the overall methodology can be found in the RASSP Design For Testability Methodology document. There is a close relationship between the DFT Methodology document and the overall RASSP Methodology Document. DFT and test activities are incorporated into the overall methodology document at a high level. The DFT Methodology document describes these activities in more detail and how they interface with other RASSP design activities.

High Level Description of the DFT Methodology

Using Figure 1-11, a high level description of the DFT methodology is presented. The methodology begins with a tangible management commitment. This tangible commitment provides the budget and resources to proceed with the methodology.

System Definition involves test requirements specification. The test requirements come from an integration of customer and derived requirements for the three phases of testing—design, manufacturing, and field support. Specification involves a preliminary, life-cycle cost of test analysis (if such an economics model is available), a test technology assessment, and a design impact analysis to determine the realizability, consistency, and validity of the requirements. Subsequently, during the same step, the three sets of requirements are consolidated into a consolidated requirements specification. The purpose of consolidating the life cycle requirements is to establish a single source of test requirements and, more importantly, to encourage all three test organizations (design, manufacturing, and field) to explore the possibility of a singular test philosophy that can be used throughout the entire life cycle of the system.

The Architecture Definition phase consists of three steps: functional design, architecture selection, and architecture verification. In the functional design step, the consolidated test requirements are used to develop the top level test strategy for the design, manufacturing, and field test phases.

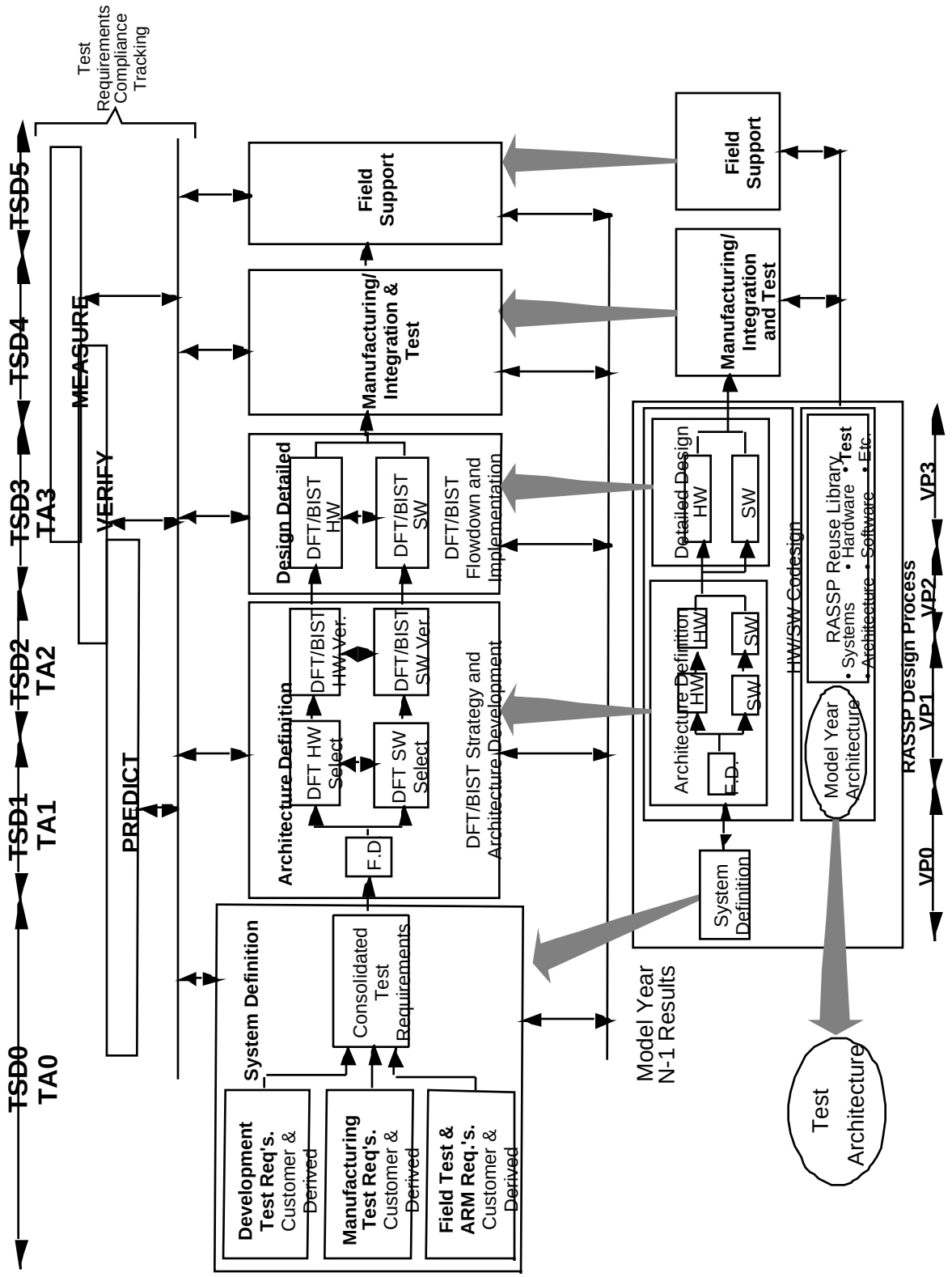


Figure 1-11. Overview of the DFT Methodology.

In the architecture selection step, the top level test strategy is used to develop and evaluate various candidate, top level test architectures, by determining which architecture(s) best supports the top level test strategy with the least impact on the candidate functional architectures. Thus, the impact of each test architecture on the candidate functional architectures is assessed and incorporated into the tradeoff and selection process for them. The test requirements, test strategy, and test architecture are then allocated to BIST/DFT hardware and software for one or more of the selected architectures. In addition, candidate DFT and BIST techniques are identified for later implementation, based on the specified requirements. Also in this step, any top level BIST supervisory software development will begin, as will on-line BIST code, since it may have an impact on performance and throughput. Prediction and verification processes begin in this stage as appropriate for compliance tracking.

In the Architecture Verification step, the next level of detail of the selected test architecture(s) are generated and additional details are provided regarding the test architecture impact on the selected functional architecture(s). For example, behavioral and performance simulations will include effects of DFT/BIST techniques, such as the estimated performance degradation due to hardware concurrent fault detection circuits or due to periodic execution of on-line BIST software diagnostics. Prediction and verification processes continue during this stage for compliance tracking.

In the Detailed Design phase, test strategies, architecture, and requirements are flowed down to the detailed design of BIST/DFT hardware and software. The detailed design of the BIST/DFT hardware is performed concurrently and interactively with functional design using automatic or manual insertion and then is reflected into behavioral and structural simulation models, whenever possible. The BIST/DFT detailed design is performed interactively with the functional design, since each impacts the other. Any remaining BIST software (e.g., for power-up or other off-line BIST functions) is implemented. Test vector sets are developed and verified for each packaging level for physical prototype test, production test, and field test. All test vector sets are documented using WAVES. Prediction, verification, and measurement processes are used in this stage for requirements compliance tracking. As the detailed design is verified and debugged, design flaw models are updated to provide more accurate models for the TSDs. Prediction, verification, and measurement processes are used in this stage for requirements compliance tracking.

In the Manufacturing phase, functional and performance testing of the overall prototype is performed to verify compliance to functional and performance requirements, with DFT and BIST hardware and software included. Ongoing production test is performed. Verification and measurement processes are used in this stage for compliance tracking to test requirements. Measurement data is acquired through such techniques as automatic system fault history logging and ATE-based data collection. BIST and tester-based test cost and performance data are captured and encapsulated for the reuse library. Manufacturing defect analysis profiles and distributions are used to update the manufacturing fault model for use in the TSDs.

In the field phase, BIST and DFT capabilities are in use. BIST and DFT functions are also used for lower level (e.g., organizational and depot level) testing. Verification and measurement processes are used in this stage for compliance tracking. As in the manufacturing phase, measurement data is acquired through such techniques as automatic system fault history logging and ATE-based data collection. BIST and tester-based test cost and performance data are captured and encapsulated for the reuse library. Field defect analysis profiles and distributions are used to update the field fault model for use in the TSDs.

Throughout the entire DFT methodology, interfacing is done to the RASSP reuse library to access existing candidates and to add to the library when appropriate. In addition, feedback is being provided continuously from the compliance tracking process back to the responsible persons to assure corrective action is taken. Finally, it is recognized that iterations may be necessary because of the inherent nature of the RASSP methodology.

One of the key ways in which the DFT Methodology contributes to the achievement of the RASSP goals is through the enforcement of reuse of DFT in four different dimensions. The four dimensions of reuse are as follows:

a. Across the life cycle phases within a given model year.
b. Across the packaging hierarchy within a given model year.
c. Across a single packaging level within a given model year.
d. Across new model years.

Test related reuse items include, for example, test requirements; test strategies; DFT/BIST techniques for certain logic structures; testable chips, MCMs, etc.; BIST software modules; and test vector sets for certain library elements.

Libraries are needed for reuse across model years and product lines. DFT reuse library elements for these needs are managed within the enterprise system by RRDM. Current component reuse library software applications including RRDM do not support the reuse proposed by the RASSP DFT methodology for:

a. Reuse across the life cycle phases within a given model year.
b. Reuse across the packaging hierarchy within a given model year.
c. Reuse across a single packaging level within a given model year.

Control and verification is required for these reuse elements just as it is required for component library elements. Control and verification mechanisms are implemented by Test Strategy Diagrams described in the DFT Methodology document.

The test strategy diagram (TSD) is a key construct used in the DFT Methodology to bridge from requirements to implementation, manufacturing, and field; to "knit" all processes together; to provide a means of carrying information between and within steps of the process; and to manage the requirements specification and compliance tracking. The Test Strategy Diagram is used to allocate a given "fault population" onto "test means" for detection, isolation and correction. The TSDs and Test Architectures (TA’s) are developed hierarchically, in a ‘top-down’ methodology in-step with the development of the virtual prototypes. The TSDs are then used in a ‘bottom-up’ verification path as equipment is built and fielded to verify compliance to the original test requirements.

There is a close relationship between the RASSP DFT Methodology and the RASSP Testability Architecture Description. While the DFT Methodology discusses DFT processes, the Testability Architecture Description discusses a prescribed testability architecture that is based on current DFT techniques used in custom design and COTS design environments. The prescribed architecture is both compatible with and derivable from the DFT Methodology. Since the Testability Architecture reflects the details of the test related features of the RASSP Model Year Architecture, all three documents have a strong relationship.

Overview of the Recommended Testability Architecture

The prescribed RASSP testability architecture (see Figure 1-12) relies heavily on the use of BIST and IEEE 1149.1 boundary-scan incorporated at the IC level. It also relies on the reuse of BIST and boundary-scan tests at all packaging levels from the MCM-level to the system-level. COTS components and/or designs are accommodated by use of the “lead, follow, or get out of the way” philosophy. The “lead” concept suggests insertion or use of a COTS item or re-use item that incorporates BIST or DFT features that can form the basis for a test. The “follow” concept applies to COTS items that may not incorporate BIST or DFT features, but which at least do not interfere with the test (e.g., simply pass the test vectors through to the next stage). Finally, the “get out of the way” concept applies to COTS items which lack testability and BIST, and which must be bypassed during the primary test process and dealt with separately for their own test.

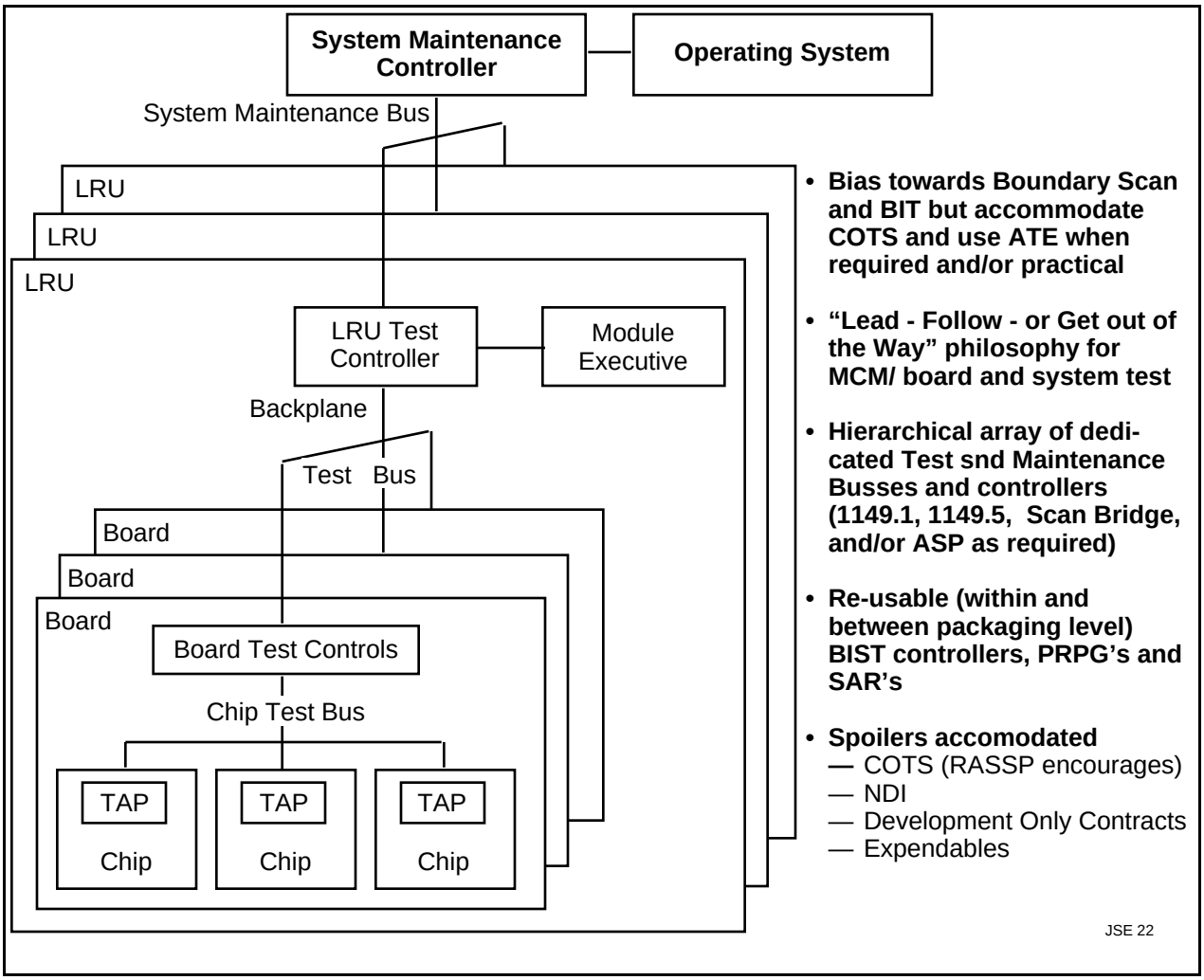


Figure 1-12. Recommended testability architecture.

Communication of test data across the packaging hierarchy is accomplished through a system of hierarchical test and maintenance controllers and busses such as IEEE 1149.5.

Further details of the testability architecture can be found in the "RASSP Testability Architecture Description."

Contribution of DFT to Meeting the RASSP Goals

The DFT Methodology contributes to achievement of the overall RASSP goals in two ways:

First, adoption of DFT practices such as being developed and practiced within industry results in reduced cycle time, reduced cost, improved quality, predictable schedules (including integration and test) or in other words improved time to market (or more importantly time to profit). For example, companies have seen 4-5X reduction in board test time by using boundary scan based testing. These benefits are realized because such practices facilitate debugging (helping both design and test engineers), hardware/software integration, transition of design data to test, test generation, and other test related activities.

Secondly, the structured DFT methodology provides improvement of the DFT process itself compared to current industry practice. This is achieved by the introduction of proven system engineering practices such as the consolidation of test requirements and by leveraging the top down development of the overall RASSP methodology to flow-down the test strategies and architecture from the system to chip packaging levels and across life cycle phases of the product.

Specific contributions of the DFT Methodology to meeting the RASSP goals include:

a. Promotes concurrent engineering by providing a specific methodology for integrating test with design activities. The methodology integrates tightly with the RASSP methodology and leverages top down design, virtual prototyping and hardware/software codesign.

b. Enhances the probability of first pass success by providing specific steps to ensure requirements are consistent, valid and reasonable and by providing a structured process for flowing requirements, strategies and architecture down to lower levels. The impact of errors on schedule and cost are minimized by incorporation of DFT/BIST which detect, isolate and correct as appropriate and/or required.

c. Promotes a singular test strategy to reduce test development time and cost across the product life cycle (example, PC based boundary-scan test for design and manufacturing and then reuse of boundary scan test in the field via embedded boundary scan controllers).

d. Leverages reuse of any output of any DFT methodology step. Reuse for subsequent model years and other products is supported via the RASSP Reuse Data Manager within the RASSP System. In addition, a structured process and mechanism is provided for ensuring reuse of test resources within a model year:
   - Across the life cycle phases
   - Across the packaging hierarchy
   - Across a single packaging level

   Control and verification is required for these reuse elements just as it is required for library elements. This control and verification is implemented by the Test Strategy Diagrams described in this document. The test strategy diagram method enforces reuse analysis and knits all of the DFT Methodology steps together.

e. Provides a framework for codesign of DFT/BIST with functional hardware/software and integrates tightly with the RASSP methodology. The framework facilitates automation and high level synthesis.

Further details of the DFT Methodology can be found in the RASSP Design For Testability Methodology document.

1.3 Implementation/Application of Methodology

1.3.1 Application of the Methodology

The RASSP methodology provides a process users can follow to more rapidly define and implement DSPs. This methodology is applicable to many areas of the more general digital processing domain as well. Applying the process to a particular domain, company, or program will require changes/modifications to support three factors that must be considered to ensure wide-range acceptance and optimize performance.

1) The process must be sensitive to existing and current procedures within a specific organization. While RASSP strives to change the way we do design, it must also recognize that *success will be determined by its ability to be integrated into an existing culture*. Just as the process must support integration of new designs into legacy systems, the methodology must be adaptable to legacy processes that are in place, are being performed by trained personnel, and *are currently working within that organization*. Existing investments in tools and training must be preserved to the greatest extent possible. The RASSP process should be adopted, over time, *to form the best of the two processes* for the organization.

2) The process must be adaptive to support program-specific constraints. Each program has a specific set of performance, cost, and schedule constraints that will dictate which elements of the methodology will be emphasized, de-emphasized, or skipped entirely. The program planning phase at the beginning of the program determines the best use of the methodology. Within the RASSP enterprise context, these parameters can drive the program and methodology management tools that will control the process flow for a specific program.

3) The process must support a range of implementation approaches, covering a continuum from completely new designs to minor upgrades of older legacy designs. Benefits gained will be design-specific, as maximum benefit will be derived from maximizing design reuse, which requires no new architectural element models or software primitives. The methodology provides paths to support new designs (from scratch) and integration into existing (legacy) systems with improved cycle time, cost, and quality. This is accomplished by supporting hierarchical codesign/co-verification of designs that are heterogeneous combinations of new components, existing hardware, and virtual prototypes (verified models).

The process must be adaptable to support either modifications and/or use of subsets of its defined steps. The implementation of the RASSP methodology throughout the design process is controlled using the design methodology manager, which uses a codification of the process flow diagrams in a process definition language. Modifying the process for a particular program/ company/design is accomplished by changes to this methodology management program. Users can easily optimize the process to support a particular set of needs to maximize improvements in design cycle time, cost, and quality.

1.3.2 Enterprise System Implementation

Along with the methodology and the Model Year Architecture, an integrated, enterprise-level development environment is one of the critical elements in the RASSP technology triad. The enterprise implements the methodology and it was conceived to implement an integrated product development environment by providing tools and an infrastructure to support the IPDT and PDTs. The RASSP enterprise framework is shown in Figure 1-13. The enterprise encompasses the RASSP design environment, and the electronic links from the design environment to the virtual prototyping, manufacturing, and test capabilities. The enterprise framework is wholly composed of commercial tools integrated seamlessly in a window-based system that operates on commercial UNIX workstations.

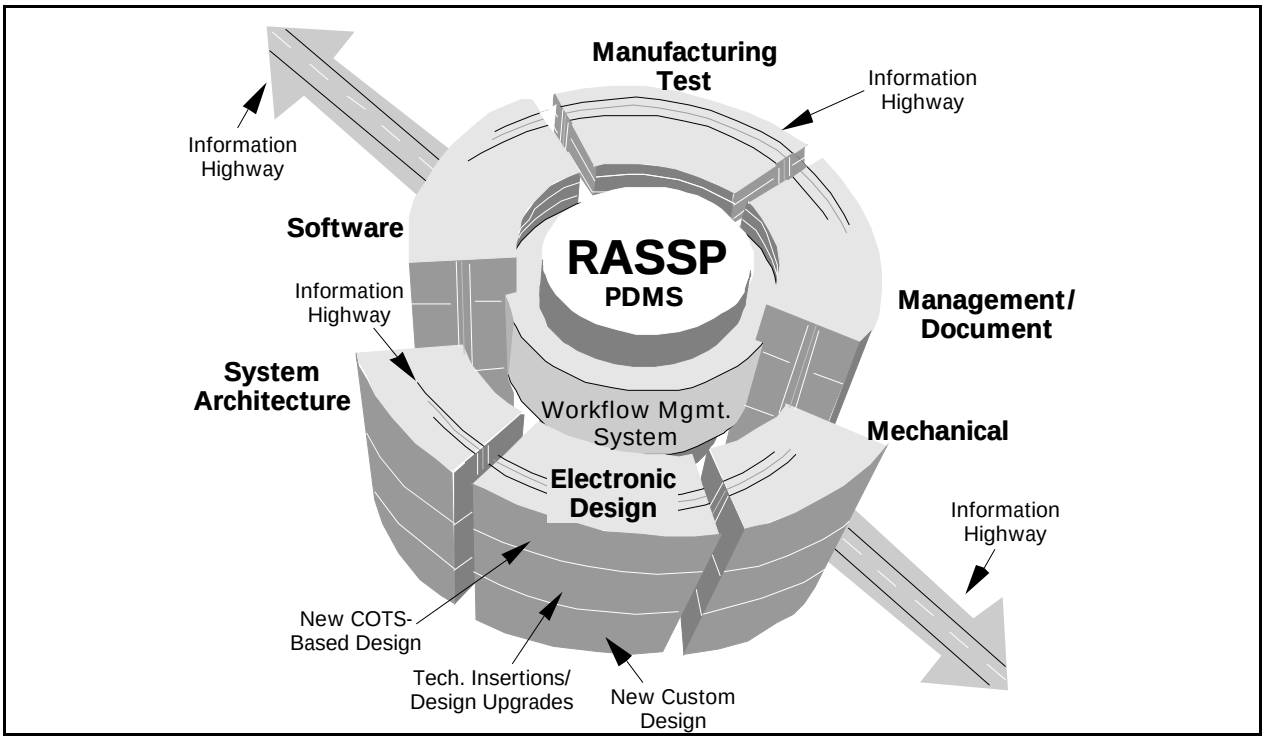


Figure 1-13. RASSP enterprise framework.

Figure 1-14 illustrates the enterprise system in the context of how users utilize the environment. We implemented the RASSP methodology in the enterprise system by codifying the process steps into workflows. The workflows are then implemented using the Design Methodology Manager (DMM). The DMM guides users through the process, providing access to the appropriate tools at the proper times, ensuring that complete data packages are generated and that all critical steps in the process are followed. Seamless tool access throughout the design process is provided via the desktop manager, which also provides a common user interface across the system. DMM also triggers the appropriate elements within the product data management system to ensure that data automatically transitions between workflow steps in the proper format.

Data management in the system is supported by the network file manager, which provides a complete file tracking capability as the development progresses for documentation of the subsystem/system configuration. This capability provides a means for archiving data/analyses generated during the development, with appropriate levels of configuration control. The library management system provides for data management and data searching at the object level, while also supporting hardware and software objects that comprise the RASSP reuse library.

The RASSP network is critical to integrated product development. The network supports concurrent design activities within each PDT and between PDTs. The enterprise supports the concept of a virtually collocated PDT, enabling remote interaction with tools and data at all stages of the design process, and enabling electronic transfer of design information, process data, procurement and scheduling information to developers, RASSP team members, and third-party vendors. Access to other communication media is also important, such as email, on-line access to documentation and CAD files, and video teleconferencing.

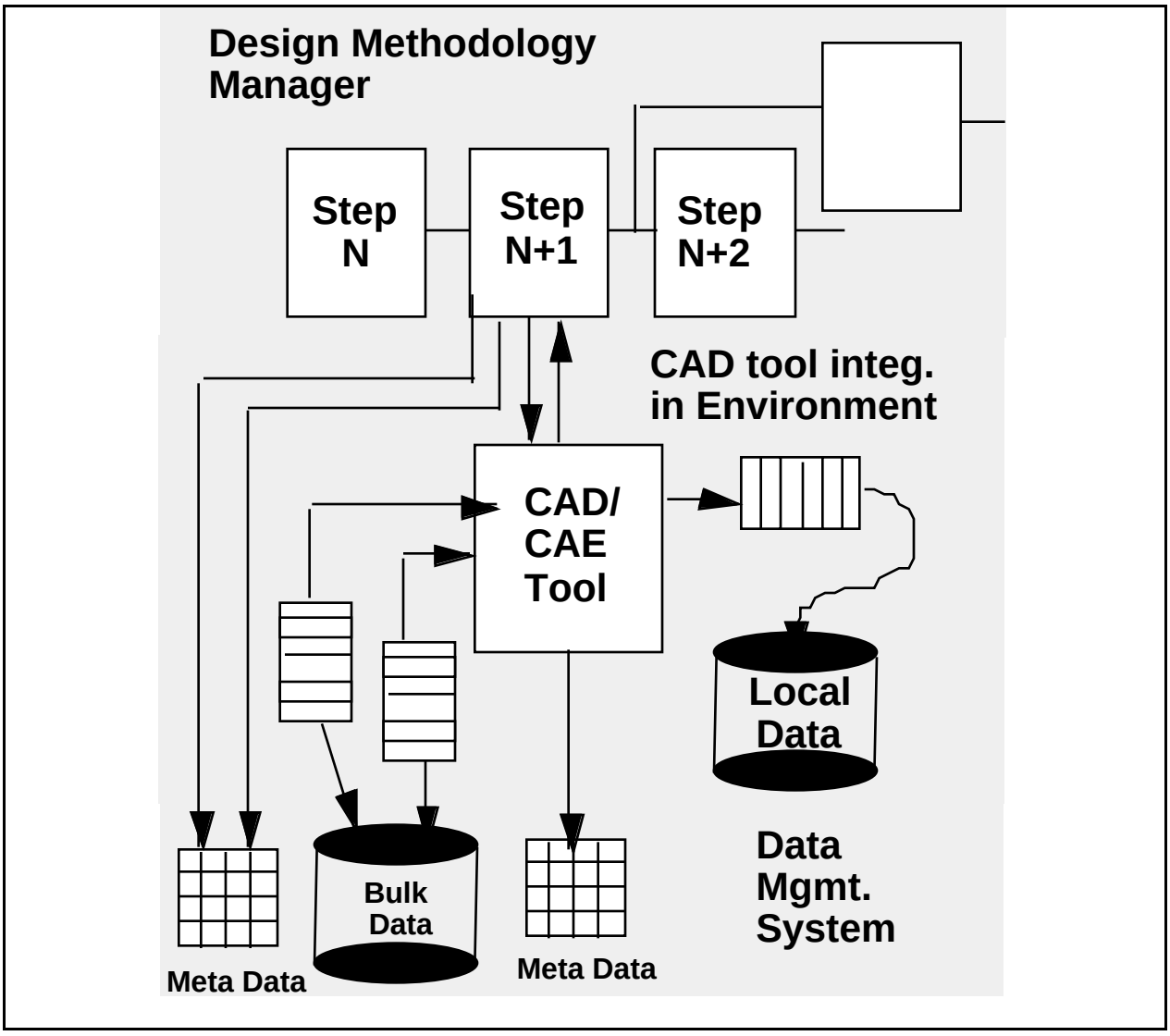


Figure 1-14. User view of enterprise system.

Section Two
Program Planning and Management

As shown in Figure 2-1, the first step in the RASSP product process flow is to develop a technical plan for the program. The project management function then executes this plan throughout the program and interfaces with all other process elements.

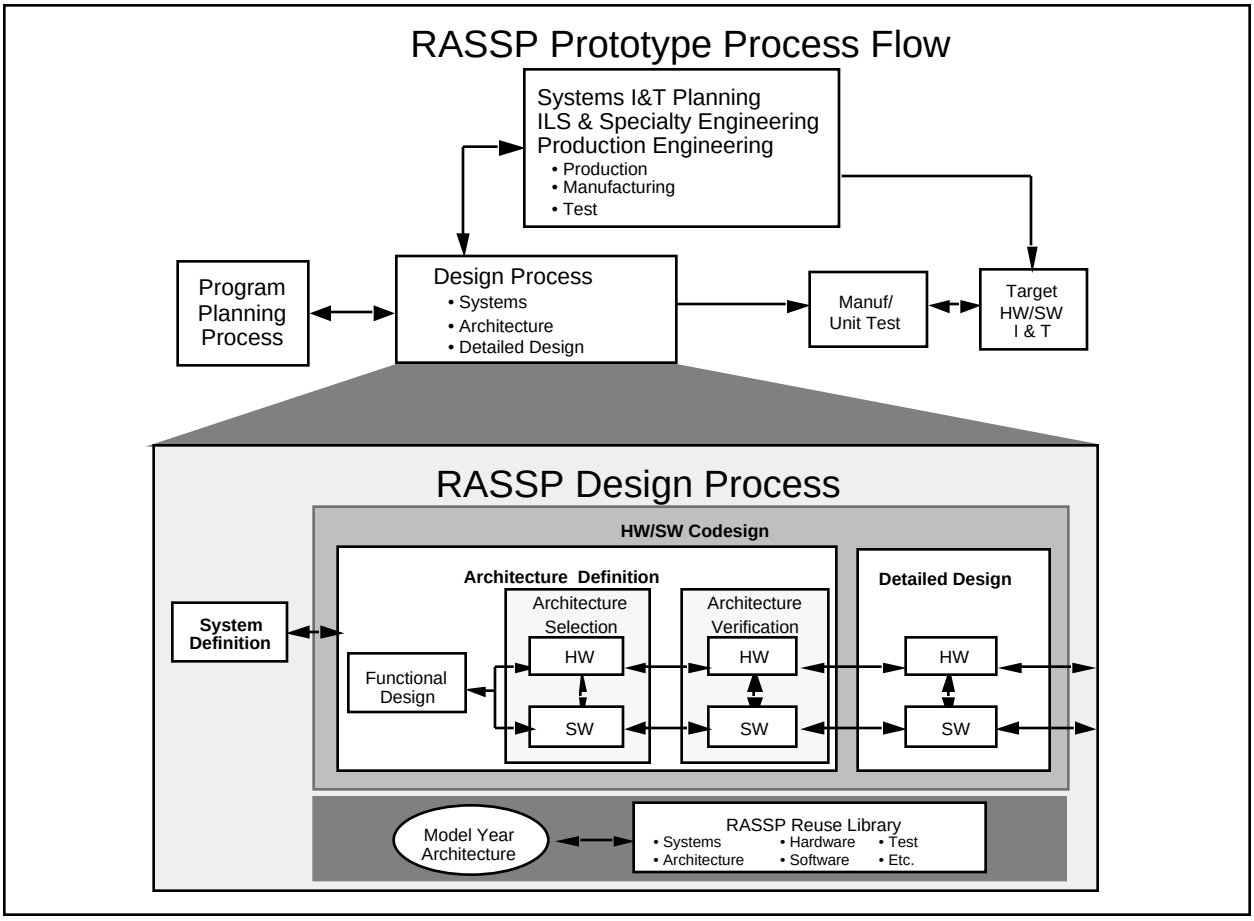


Figure 2-1. RASSP product process flow.

The planning process provides control of the technical development process. It establishes the environment, tasks, organization, and control features that permit the subsequent engineering processes to meet schedule, cost, and design goals. The plan provides visibility into the status of the development process and is a basis for resource adjustments to meet commitments. For RASSP, this planning process is a living technical management device that is applied more dynamically than in traditional methodologies. The prototyping nature of RASSP requires the overall program plan to be modified iteratively as the processor development progresses. These iterations correspond to specific levels within the RASSP spiral model (described in Section 1.2), providing both planning for successive cycles in the design process and developing risk reduction plans at critical decision (design review) points within the program.

2.1 Program Planning Process

The program planning process takes the materials available at the beginning of a program (procurement package, proposal, RASSP methodology handbook, departmental procedures, and existing technical data) and produces a technical database, the System Engineering Management Plan (SEMP), and

develops an approach for concurrent engineering. These items are described in the following sections.

2.1.1 Technical Data Base

The program technical database is a single repository for all current and historical technical information and direction. It is a technical interface between all engineering disciplines and process steps. It includes, but is not limited to:

- Administrative Files — This includes all supporting material, such as referenced methods and standards, briefing material, directive memos, technical interchange memos, and justification memos. This file also includes all published or publication-ready documents.

- Design Data Files — This data is the technical material that expresses the design of each entity being designed (software and hardware). This includes, but is not limited to, drawings, code listings, and parts lists.

- Requirements Files — These are the requirements "shall" statements and their associated verification statements, with the pointers to the requirement sources, equipment allocations, decomposition statements, trade-off studies, publication paragraphs, and publication titles.

2.1.2 System Engineering Management Plan (SEMP)

The SEMP formally publishes the plans for the development process to be undertaken by engineering on a program. The document is the basis for agreement between PMO and Engineering for the engineering tasks on a program, for cost estimating, and for monitoring progress. The SEMP is usually applied to an overall system (platform-level) description, but in the RASSP context will be applied to the development of the signal processing subsystem for use by the Signal Processing Subsystem Product Development Team.

The SEMP is composed of three parts: the approach to concurrent engineering, the description of engineering process, and a description of the program planning and control.

2.1.3 Concurrent Engineering

The RASSP concurrent engineering methodology is based on four major elements:

- Multifunctional, virtually collocated product development teams (PDTs)
- A functionally integrated program schedule
- Integrated tools
- Performance metrics

2.1.3.1 Multifunctional Teams

The multifunctional PDT designing the Signal Processor Subsystem (SPS) works closely with the Integrated Product Development Team (IPDT) to ensure that this subsystem operates properly within the context of the overall system. These organizations are briefly described in the following paragraphs(repeating some information from Section 1 for completeness). The interaction between members is described, followed by their roles within the RASSP process.

Integrated Product Development Team (IPDT)

The IPDT, under the leadership of the program manager, has overall responsibility for design, analysis, development, fabrication, and evaluation of the product. It coordinates tasks assigned to the PDTs, and its members are responsible for specific overall systems engineering and program management tasks not delegated to PDTs. The roles of the IPDT members are focused in three areas: program management, technical design integration, and functional specialty integration.

The Program Management role includes the IPDT members responsible for programmatic integration and management. This group leads the program and is the formal interface with the customer. The group coordinates programmatic schedules, manages the program cost performance report process, and manages subcontracts. They conduct the life-cycle cost (LCC) program, the risk management program, and the configuration control program. The group plans and initiates team training for concurrent engineering and group dynamics, facilitates creation of the program concurrent engineering handbook, and advocates continuous process improvements.

The Design Integration role is responsible for the technical integrity of the product. They perform the system-level design and technical integration between PDTs. They are also responsible for all systems requirements analysis, functional decomposition, system-level performance analysis, trade studies, and support to the IPDT. This group assures that the PDTs develop an integrated, synergistic product and they coordinate both external and internal technical interfaces.

The Functional Integration role is responsible for assuring that design engineering, engineering specialties, manufacturing, and ILS disciplines work together to produce a product that meets the customer needs with the lowest possible LCC. The group:

- Coordinates all specialty disciplines across all PDTs
- Manages the reliability and maintainability program
- Establishes the ILS plan
- Establishes and maintain the NBC and MANPRINT programs
- Establishes an integrated manufacturing plan
- Provides general support across all PDTs

The day-to-day, cross-team participation of these IPDT members is vital to achieving design consistency and avoiding design sub-optimization (i.e., one PDT making decisions that have negative LCC impacts in other area).

Product Development Teams (PDTs)

PDTs partition the design and development task into manageable efforts that can be accomplished by multifunctional teams of reasonable size. The task partitioning is based on the derived architecture and the subsystem partitioning. Each team is responsible for accomplishing assigned design/development, fabrication, and validation tasks, and is provided design-to-cost (DTC) budgets to guide their development activities for their respective SPS. The key responsibilities and tasks are:

- Maintain an SPS PDT handbook and design process that will ensure that the SPS meets its performance, design, producibility, reliability, testability, maintainability, and LCC requirements within its assigned DTC budgets and the constraints of the program master schedule
- Perform trade studies in support of LCC reduction
- Electrical design
- Software design
- Mechanical design
- Fabrication, integration and test
- Software coding and unit test
- Software integration and test
- Generate CDRL data.

The design team approach is the key to implementing concurrent engineering on RASSP developments. The multifunctional teams are assigned to work together throughout all phases of the design. Each team is responsible for ensuring that producibility/production processes, as well as design, reliability, maintainability, safety, quality assurance, and supportability needs are met cost-effectively. This is accomplished mainly by considering them from the very beginning of the design process. This up–front emphasis on cost and requirements for the entire life cycle of the product achieves the most effective design and shortens the design cycle by minimizing design changes/updates. The general roles of the team members are described in Table 2-1 for the SPS PDT.

Table 2-1. Role of team members in PDT.

Member	Role in Signal Processing Subsystem PDT
Team Leader	Provide PDT leadership and integration of the multifunctional discipline activities to produce the signal processing system that meets performance requirements at the lowest life cycle cost
Systems Engineer	Monitor and interpret specification requirements; lead system tradeoff studies and system design verification
Architecture Engineer	Interpret subsystem requirements; lead architecture trade studies, HW/SW codesign, and architecture verification
Digital Engineer	Digital requirements analysis; support architecture trade studies; participate in architecture verification; design, build and test hardware
Software Engineer	Software requirements analysis; participate in HW/SW codesign; design, code, and test library elements; participate in integration and test
Mechanical Engineer	Support architecture trade studies; develop chassis; frame design; perform thermal analysis, packaging trade studies
Manufacturing	Identify special processes that may be required for production build
Test & Maintenance	Develop test philosophies to optimize testability and maintainability and conduct development testing for prototype
Producibility	Support trade studies for part selection/approval and maintain parts library; assess candidate designs and verify that the final design meets requirements for RAM, safety, EMI, Tempest, and life cycle cost
Sourcing	Support trade studies and design by identifying vendors that are able to provide high quality, reasonably priced materials in a timely manner
ILS	Assess the consequence of design decisions on ILS and total system costs
Cost Analyst	Support trade studies by providing cost impact of implementation technology; support life cycle cost analysis throughout design cycle

Separation of the design process into phases provides checkpoints for reviews to provide feedback, so that necessary corrective actions can be taken in a timely manner. The RASSP design phases are delineated as Systems Design, Architecture Design, Detailed Design, and Validation (fabrication, integration and test). Critical decision points within each process corresponds to each cycle of the spiral model. At the conclusion of each spiral, a design review is held to evaluate the current virtual prototype and identify the risks that will drive the next iteration of the design. These reviews correspond to the decision points within the RASSP spiral model (shown in Figure 1-4). All PDT functions are assigned tasks

for each phase, which are summarized in Table 2-2. Design issue checklists are established, tailored for the specific product, and issued early within the design phase. These checklists define the issues to be resolved before the design review that concludes each iteration of the virtual prototype. It is the responsibility of the team to work together toward a solution that addresses all items in the checklists.

Design guidelines are developed by the functional specialists and applied to all PDTs across the program. This ensures consistent design practices and standards for all teams (functional integration). Vendors/suppliers are included in the design teams, and they must address the same life-cycle issues as the internal design team members. The principal benefits of concurrent engineering are well understood. The implementation strategy for RASSP is the charter of the small, multifunctional design teams that must address all product life cycle issues at each phase. The team leader is responsible for ensuring participation of all functions through out each phase. The team is responsible for identifying high-risk items, and developing risk reduction plans and appropriate activities to reduce risk early in the program. The team also develops the plans for the next program phase in the RASSP iterative process model. The team is further chartered to make LCC trades to develop a design that meets all performance requirements at the lowest LCC. Each team is provided with an LCC DTC target, and the LCC specialist supports the design team by developing the relationships between product characteristics and total LCC. The SPS PDT will perform the electrical, software, and mechanical design, fabrication, coding and unit test, integration, and test of the RASSP engineering prototypes.

2.1.3.2 Master Schedule

The SPS schedule identifies the systems, mechanical, electrical, and software members' functions that must occur as a function of time. The other members of the PDT monitor these design and implementation activities. The SPS PDT closely monitors the activities of other PDTs to ensure that scheduled inputs are maintained or work–around plans developed.

2.1.3.3 RASSP Integrated Tool Environment

Along with the methodology and the Model Year Architecture, an integrated enterprise-level development environment is one of the critical elements in the RASSP technology triad. The objective of this document is not to describe the RASSP enterprise and design environments; however, a discussion of how the enterprise system is used within the context of the PDTs is warranted.

The RASSP enterprise system facilitates IPDTs by providing tools and an infrastructure to support concurrent engineering in a highly automated and distributed environment. Figure 2-2 illustrates the enterprise system in the context of a concurrent engineering environment. Users are supported by a desktop environment that provides services to support interaction with either individual tools or groups of related tools coupled with specific frameworks. The enterprise system provides support services that enable interaction of users with other users on common internal networks. In addition, the enterprise system supports interaction with external system users, such as team member organizations, manufacturing centers, and external suppliers who are members of the concurrent engineering development team.

Table 2-2. Role of PDT members throughout RASSP design process.

	Roles of Product Development Team Members in RASSP Processes			
	Systems Process	**Architecture Process**	**Software Process**	**Hardware Process**
Team Leader	Overall technical responsibility and coordination with IPDT.	Overall technical responsibility and coordination with IPDT.	Overall technical responsibility and coordination with IPDT.	Overall technical responsibility and coordination with IPDT.
Engineering • Systems • Architecture • Digital • Software • Mechanical	Systems - lead role; interpret customer rqmts, perform system level design, manage test rqmts, coordinate development of subsystem specifications and system level trade studies. Architecture - participate in system level tradeoffs to allocate the requirements to subsystems. Digital - minimum role, analyze the system level hardware requirements to determine impact on system design. Software - minimum role, analyze the system level software rqmts to determine the impact on system design. Mechanical - analyze operational system rqmts and perform tradeoffs to determine the structural and packaging rqmts for each subsystem.	Systems - ensure that all candidate architectures comply with system rqmts. Interact with systems process to refine system definition. Architecture - lead role; perform all tradeoff studies; ensure incorporation of testability into trades; responsible for the arch selection and verification of final architecture. Digital - support trade-offs and participate in the arch verification efforts prior to design release. Software - intimate interaction with arch engineers via HW/SW codesign. Define specs for and implement prototype library elements. Mechanical - provide support to arch trade studies, initial mechanical/thermal estimates, develop conceptual mechanical approach.	Systems - ensure that all timelines dictated by system requirements are being satisfied. Architecture - intimate interaction with software engineers in the HW/SW codesign process. Digital - minimum support. Software - lead role; responsible for the data integrity and real time performance of the final system; responsible for all in-process and out-of-process software development, software testing, and documentation. Mechanical - no support.	System - ensure that all designs comply with overall system and subsystem requirements. Architecture - close interaction to translate behavioral designs to detailed hardware components. Digital - lead role; partitioning, design capture and simulation of detailed module, ASIC, etc. designs. Development/ verification of full virtual prototype. Mechanical - lead role; packaging trade studies, development of detailed design and generation of structure and assembly drawings for chassis and frame designs. Perform thermal analysis.

Table 2-2 (cont.). Role of PDT members throughout RASSP design process.

	Roles of Product Development Team Members in RASSP Processes			
	Systems Process	**Architecture Process**	**Software Process**	**Hardware Process**
Manufacturing	Assess the impact of allocated functional and physical requirements on current manufacturing processes and participate in the make/buy decisions for each subsystem.	Support trade studies with regard to manufacturing difficulty of alternate architectures.	No inputs required.	Close interaction to ensure that designs meet DTC and manufacturability goals.
Test	Determine the system level test requirements needed to determine functionality on the manufacturing floor, in the depot and in the operational environment; decompose requirements and allocate to subsystem; develop test plans and identify test risk areas.	Support trade studies by evaluating each candidate architecture with regard to testability over the life cycle of the product; develop test strategies and partition architecture for fine grain fault ambiguity groups.	Prepare the specific test plans for all the in-process and out-of-process software in accordance with the Software Validation Plan; include plans for library elements, assembly of the library elements into a software system and interconnection, data integrity and real time performance of the system.	Develop Test Program Sets and Automatic Test Sets; develop fault diagnostic techniques and dictionaries; conduct tests; collect and analyze data; feedback in-process test data to IPDT.
Producibility • Prod Assurance • Parts Mgmt • RAM&S • EMI/Tempest • LCC/DTC	Ensure that the system can be safely built and maintained and that the final product meets all 'ilities as well as environmental requirements; ensure that cost impact over the life cycle is properly addressed.	Ensure that candidate architecture(s) meet 'ilities and environmental requirements; Ensure that architecture decisions reflect both the design to cost goals and the long term impact on cost.	LCC personnel will work with software engineers to provide inputs to the architecture tradeoff process.	Support final parts selection and specialty engineering analyses to ensure final design meets all requirements. Work with digital and mechanical to ensure case of maintenance.

Table 2-2 (cont.). Role of PDT members throughout RASSP design process.

	Roles of Product Development Team Members in RASSP Processes			
	Systems Process	**Architecture Process**	**Software Process**	**Hardware Process**
Sourcing	Responsible for developing the overall sourcing plan for the project; participate in the make/buy decisions for each subsystem.	Provide support with respect to component availability versus schedule requirements.	Provide information regarding cost and availability of the necessary compilers, assemblers, debuggers and related required support software.	Support procurement activities. Develop and maintain reliable sources.
ILS	Examine the operational supportability rqmts and perform tradeoffs to determine the subsystem supportability rqmts.	Assess impact of support requirements on arch decisions.	Define the long term software support required for the system.	Work with systems engineer to prepare final ILS plan.

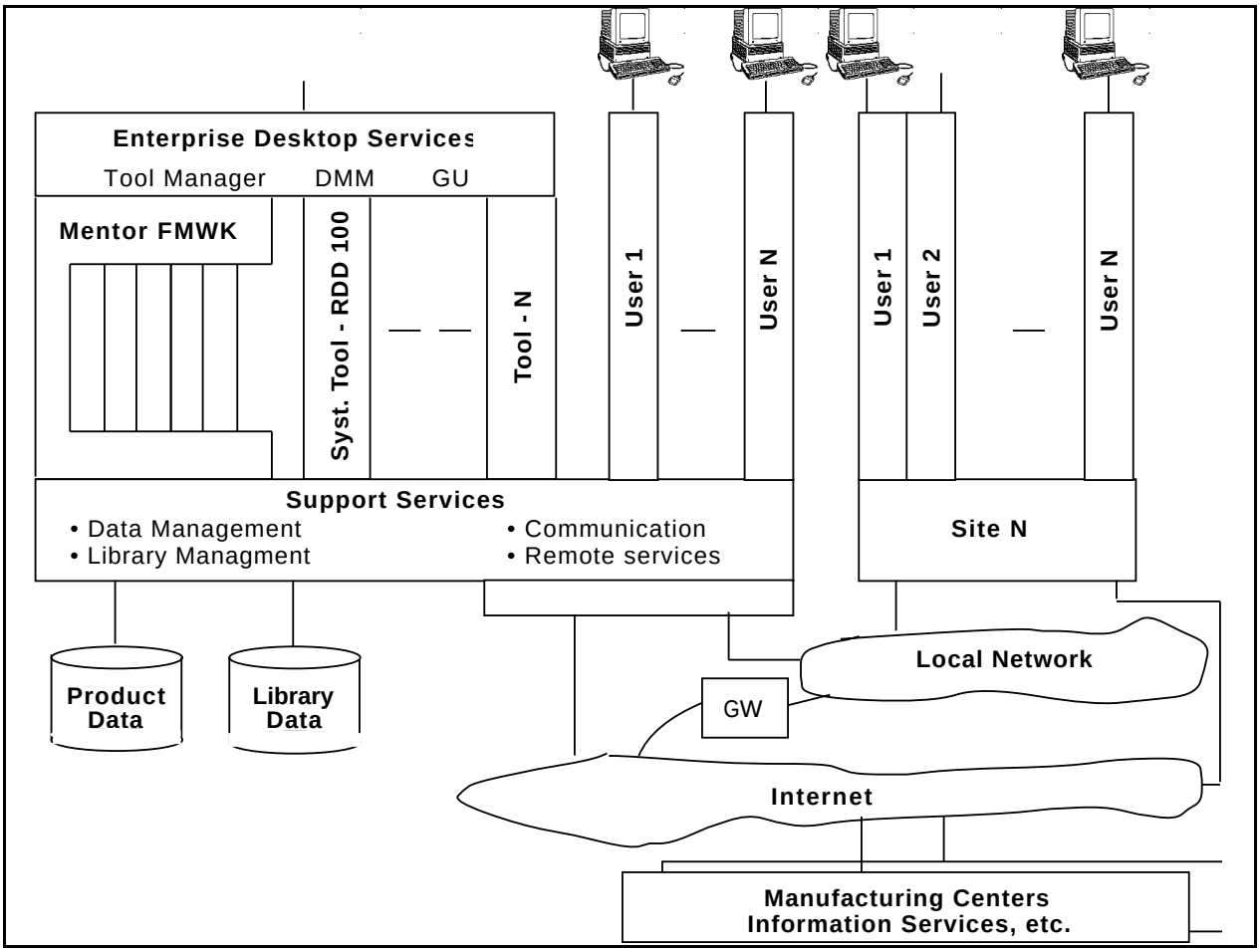


Figure 2-2. RASSP enterprise in concurrent engineering environment.

In the early phases of a signal processor development, the RASSP enterprise provides tools to support system requirements analysis, system definition, and architectural designs and trade-offs. The enterprise desktop provides a common interface to these tools and facilities. The resulting analyses and designs are tracked and managed by the Network File Manager and, as the files are released, are put under a level of configuration control appropriate for the conceptual phase.

As architectural decisions are made, based on system-level analyses and trade-offs, the subsystem definition will proceed and the responsible IPDTs assigned. The RASSP enterprise system will provide the tools for each team, and workflows will be established to implement the RASSP methodology (tailored for each subsystem design process) using the Design Methodology Manager. The resulting processes will be the basis for the PDT handbook developed by each team. Each team will access the tools via the enterprise system.

The RASSP network is critical to integrated product development, as Figure 2-2 illustrates. The network will support concurrent design activities within each PDT as well as between PDTs. Most importantly, it will enable the IPDT to integrate PDT activities and access data that is generated within each PDT to support LCC analyses, supportability and maintainability models, overall parts management and sourcing, etc. In the case of supplier personnel, for example, team members might be collocated during early phases of the program, but during the detailed design phase it might be more appropriate for the supplier personnel to execute their design tasks in their environment. This remote supplier design group would have access to the design database via the enterprise network. Tools would also be accessible on the same basis that other geographically-distributed users would access the RASSP tool suite; the only difference would be access to the particular design database of the PDT with which the supplier personnel are involved. Access to other communication media is also important, such as email, on-line access to documentation and CAD files, video teleconferencing, etc.

Table 2-3 illustrates in more detail the use of the enterprise system by the IPDT and the PDTs. AT the PDT level, the RASSP network/communication system facilitates the concurrent design/review of each subsystem as the design progresses, and supports the key element of the RASSP methodology, namely, hardware/software codesign. Manufacturability and producibility are monitored throughout the course of the design using the RASSP network facilities. At the IPDT level, the network provides access to team schedules, cost data to update the LCC model, and critical issues as they arise. The parts control program is administered centrally using the network, and Sourcing constantly monitors parts selection within the PDTs for long-lead items and high-cost parts.

The Network File Manager provides a complete file tracking capability as the development progresses for documentation of the subsystem/system configuration. This will provide a means for archiving data/analyses generated during the development with appropriate levels of configuration control.

The Design Methodology Manager (DMM) is the key to implementing the RASSP methodology for the integrated product development environment. The overall workflow will be established for the program by the IPDT, including the specific tools and design guidelines to be used, and these workflows will be tailored by each PDT. This establishes the process used by each PDT that is incorporated in each team's PDT handbook.

The Library Management System provides different levels of capability for the IPDT and the PDTs. During conceptual design, at the architectural level, the system will be searched for possible reusable subsystem blocks from prior signal processor designs, both hardware and software.

Table 2-3. Integrated product development supported by the RASSP enterprise.

RASSP Enterprise Capability	Usage	
	Integrated Product Development Team	**Product Design Teams**
Enterprise network and communications	• Track IPD designs in progress - schedules - costs - long lead items - critical issues - parts approval	• Multifunction review of subassembly design as the design progresses - electrical, mechanical reviews - manufacturability • Communication to IPDT - schedule status - critical issues - cost status - parts approval
Network File Manager	• Product Meta Data and Configuration Control as Design evolves - specifications/requirements allocation - analysis/trade studies - architectural designs - product configuration/BOM	• Design documentation/Data - analysis/trade studies - CAD files
Design Methodology Manager (I/NFM)	• Program design guidelines, tools and processes across all PDTs	• Develop work flow specific to each IPD using guidelines/ tools specified by IPDT
Library Management System (CLMS)	• Application of reusable subsystem blocks and software designs	• Reusable components, data and models for hardware design; reusable software designs
Enterprise Desktop Manager	• System level architecture design schedule/cost performance • Management and Reporting • LCC models • Sourcing plan • Integrated Manufacturing Plan • Reliability/Maintainability models • Parts control program • Software development plan • LS program plan • QA program plan	• CAE Tools - Subsystem design - Hardware design - Software design

The Desktop Manager provides several types of tools, all accessed via a common user interface. One set of tools supports the conceptual design and architectural design phases, principally at the IPDT level. This set includes architectural simulation tools, requirements management tools, etc. A second set of tools, largely spreadsheet in nature, supports LCC models and reliability/maintainability models. Scheduling tools are required to support the development of an integrated manufacturing plan, ILS and QA plans, etc., all accessible via the Desktop Manager. The third set of tools supports the design process within each PDT for subsystem design, hardware, and software design.

2.1.3.4 Performance Metrics

Evaluation of PDT performance is a critical element in RASSP because it enables quantitative measurement of the process improvements being generated by the cross-functional interaction on the program. The following measures are candidates to be added to the periodic technical performance measures and risk assessments to measure the multifunctional design process. The appropriate metrics for process improvement will evolve over the RASSP program. Each PDT leader is responsible for and is measured by a set of metrics, including the following:

- DTC/LCC Projection — DTC and LCC projection are reviewed between the PDT leaders and the PMO regularly.

- First-pass Drawing Yields — Is defined as the number of drawings released without red–lines, over the total number of drawings released. The objective is for each PDT leader to facilitate the team so that they can release drawings which receive a multifunctional sign–off without any red lines.

- Avoidable ECN $ — PDTs are measured on the estimated cost of the avoidable ECNs they generate. The process will be to divide any ECNs into two categories: those we could have avoided through the proper application of concurrent engineering principles, and those we could not. An unavoidable ECN might be, for example, due to changes in customer requirements, or recent technology advances in the supplier community. For those avoidable ECNs, a rough-order-of-magnitude estimate of the cost to implement the ECN is generated. It is the objective of each PDT to drive this cost to zero.

- Design and Process Checklists — The PDTs define their detailed schedules, and establish a set of checklists to access conformance to the design process and best practices in design guidelines. These checklists are used at upcoming design reviews to assess conformance to the design process and design guidelines. To use best practices in product design, design guidelines are either directly embedded within the design tools or are recorded in design checklists. At the beginning of the program, the PDTs receive training on design checklists of assembly, supportability, and testability, and the nature of the issues that drive them. At design reviews, the design is assessed to the checklists and discrepancies are discussed. This enables the design teams to become intimately aware of the best practices in manufacturing and supportability areas, and after being attuned to these, they can design to these best practices.

- Team Dynamics Assessment — It is the responsibility of each PDT leader to facilitate their team to assure that the members properly understand the goals, mission, and objective of the team, and to ensure that they communicate well. To facilitate this, each PDT leader is trained on how to facilitate a team. Periodically throughout the program, a team dynamics assessment will be filled out by the team members to measure the performance of the team.

These metrics are not meant to be a complete set, but rather a representative sampling of the type of data that must be collected to evaluate the process improvement. A large set of metrics were requested in the RASSP Benchmark RFP; these are being collected during benchmarks 0 and 1. These metrics will be evaluated by MIT as part of the Benchmark program to determine their utility. We are developing a parametric cost model (PRICE) to support the measurement of savings associated with the use of the RASSP methodology. The methodology will incorporate these metrics and models as they become available. These metrics are a valuable part of the feedback process for evolving/improving the overall process.

2.2 Program Control Process

Overall control of the entire program is the responsibility of the program manager in conjunction with the management programs put in place within the IPDT. It must be remembered that the RASSP methodology is focused at a lower level — the SPS development. As such, it is the responsibility of the PDT leader to ensure that all efforts within the PDT are in concert with the overall development plan and schedule put forth by the IPDT.

2.2.1 Contract Work Breakdown Schedule (CWBS)

The WBS is the basis for defining, planning, budgeting, controlling, and reporting all work performed on the program. The WBS reflects all elements of the Statement of Work (SOW), and defines all tasks to be performed to execute the program. The PDT leader coordinates team development with the CWBS set up by the IPDT. The overall CWBS contains a block at the SPS level. It is the responsibility of the PDT leader to manage all efforts corresponding to work under this block and coordinate that effort with other PDTs through the IPDT. The RASSP enterprise system facilitates coordination between the PDT and IPDT.

2.2.2 Risk Management

Risk can be defined as a potential occurrence that would be detrimental to plans or programs. Risk is measured as the combined effect of the likelihood of the occurrence and a measured or assessed consequence given that occurrence.

The risk management process (see Figure 2-3) includes a systematic and comprehensive approach to detect and identify potential program risks, analysis, handling (mitigation), and monitoring. This process is managed for the overall program by the IPDT, with participation by the various PDTs. Risks are assessed during every cycle of the spiral process. As risks are identified, the risk mitigation plan may spawn a mini-spiral to address that risk by prototyping through simulation, modeling or breadboarding. It may also cause portions of the design to reiterate along previous levels of the spiral.

Risk Identification

The first step in risk management is to identify and assess all potential risk areas. Some degree of risk always exists in the program design, test, logistics, production, and specialty engineering areas. The impact of these risks can be categorized as cost, schedule, or technical, as shown in Figure 2-3. Program risks include funding, schedule, contract relationships, and political risks. Design risks may include the ability to meet a performance requirement (such as maneuverability or survivability), the feasibility of a design concept, or the uncertainty associated with using state-of-the-art equipment or software. Production risks include concerns over packaging, manufacturing, lead times, and material availability. Specialty engineering risks include reliability, maintainability, operability, and trainability concerns. Understanding risks in all areas evolves over time. Consequently, risk analysis must continue through all program phases and is formalized as part of each cycle of the spiral process.

Risk Analysis

Describing and quantifying a specific risk and the impact of that risk on cost, schedule, and technical performance usually requires some modeling. Typical tools for risk analysis to address program impact include:

- Program Evaluation Review Technique (PERT)
- LCC Model

The models are used to examine schedule and program cost risk as various options for procurement strategy, logistics maintenance levels, and contractor support are considered.

A product of risk analysis is a "watch list." This list identifies consequences that are likely to occur and indicators of the start of the problem.

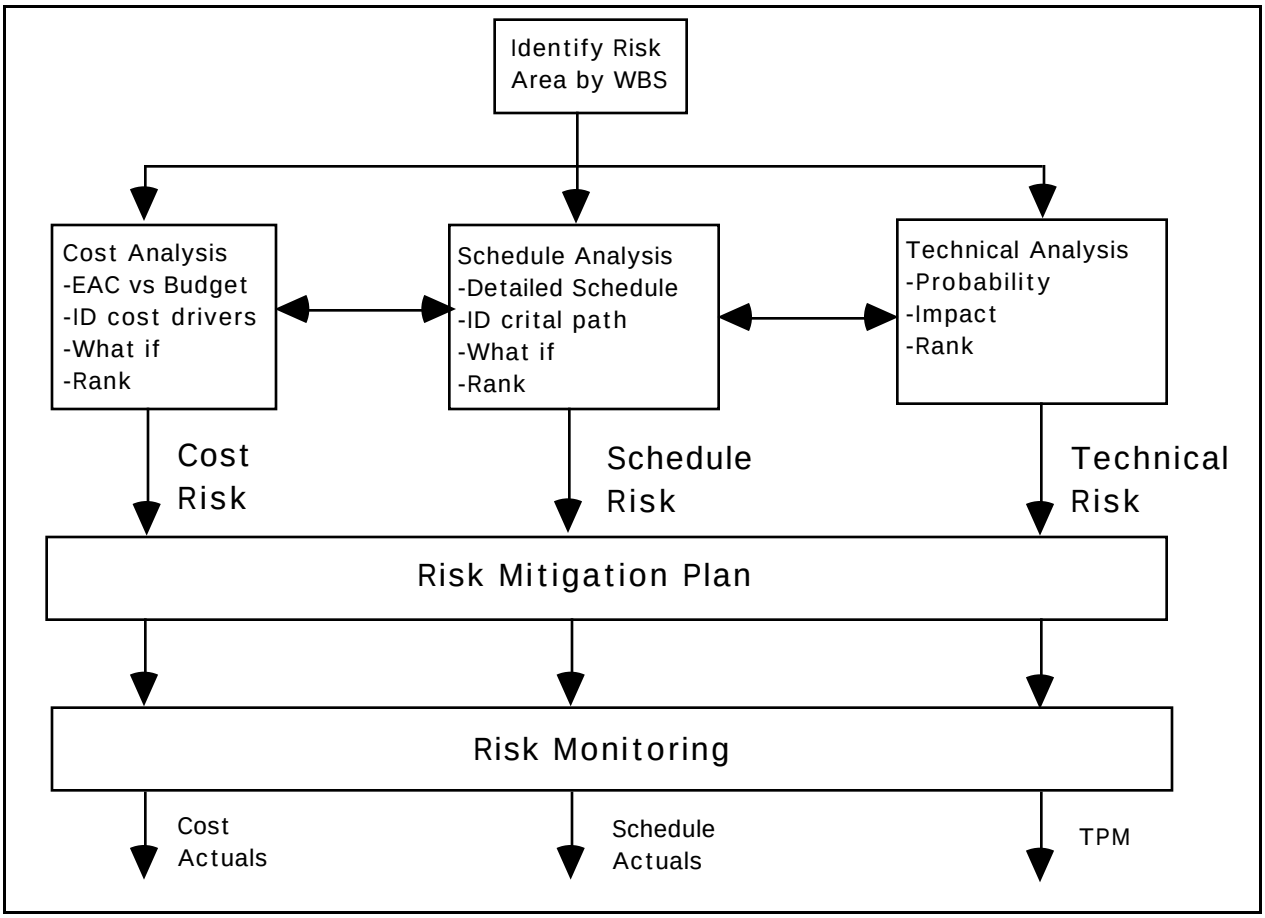


Figure 2-3. Risk management process.

Risk Handling

The techniques to reduce or control risk fall into four categories:

- Avoidance — To avoid risk is to avoid the potential failure consequence and/or its probability. Risk avoidance may be reflected in the system concept selection and contractor source selection.

- Prevention (Control) — Risk prevention is the process of continually sensing the condition of a program and developing options and fall-back positions to permit alternative lower risk solutions.

- Assumption (Retention) — Risk assumption acknowledges the existence of the risk, but a decision to accept the consequences if failure occurs.

- Knowledge and Research — Knowledge and research, as a method for risk handling, is a continuing process that enables participants to perform risk reductions by:

  — Initiating early development activities
  — Implementing extensive development testing
  — Developing simulations to predict performance.

Mini-spirals in the design process may be spawned as part of the risk mitigation process. Such an effort may be the detailed simulation to verify performance or the hardware prototyping of a new processor interface. In this way high-risk portions of the product may proceed through the development process at a faster pace than could be achieved otherwise.

Risk Monitoring

Risk items are monitored continually by the PDT and risk reduction actions are recommended. Inherent in the monitoring of technical performance design risk is evaluation of predicted performance against specified requirements. Appropriate performance parameters for risk monitoring are established at the top level, together with their contributors (or allocations) at lower levels. Properly managed Systems Engineering ensures that the risks associated with each design decision are identified and treated in risk assessment updates: how to recognize the risk if the potential problem should occur (e.g., higher failures when stress is over 60 foot-pounds) and what actions should be taken if the problem is due to the potential risk area (e.g., use substitute component).

2.2.3 Technical Performance Measurement

Technical performance measurement (TPM) is defined as the product design assessment which estimates, through engineering analysis and tests, the values of essential performance parameters of the current design of product elements. It forecasts the values to be achieved through the planned technical program effort, measures differences between the achieved values and those allocated to the product element by the Systems Engineering process, and determines the impact of these differences on system effectiveness. The purpose of this process is to:

- Provide visibility of actual versus planned performance
- Provide early detection of problems
- Assess program impact of proposed change alternatives.

Use of the TPM alerts the program manager to potential performance deficiencies before irrevocable cost or schedule impact occurs. TPM provides data for technical risk planning and assessment. TPM is a continual process in the RASSP methodology. Current design status is always available to the IPDT through access to the enterprise system. Through the continual process of prototyping in ever-increasing detail, along with requirement traceability and iterative interaction between the RASSP processes, product status is always available to assess the impact of the current design performance on system effectiveness.

2.2.4 Technical Reviews and Audits

Engineering technical reviews are held throughout the RASSP process. The intent of the reviews is to ensure that contractual requirements are met cost-effectively and to achieve contractually-required product reliability, maintainability, and producibility. The design reviews assess the technical decisions made or contemplated, evaluate the design practices used, and determine the producibility within the cost, schedule, and performance constraints in the intended production environment.

As each major cycle of the spiral is completed, there is a design review that assesses the current state of the overall design and all its pieces. It is important to note that this does not imply that all pieces of the design are at the same level of maturity. In fact, this is unlikely. The highest risk element of the design, as identified during the previous iteration, may proceed to higher levels of maturity to reduce the overall program risk. New elements of risk may have been identified during the previous iteration, which warrants initiating a new mini-spiral that explicitly addresses the new risk.

The formal design reviews described in Table 2-4 correspond to the decision points in the spiral model defined in Figure 1-4.

Table 2-4. Design reviews corresponding to spiral model decision points.

Review	Tasks	Output
System Requirements Review (SRR)	Quantify system requirements	Requirements Specification
Virtual Prototype 0 (VP0) Review (System Definition)	• Functional allocation of processing times • Definition of functional behavior • Risk identification and assessment	• Executable Specification
Virtual Prototype 1 (VP1) Review (Architecture Selection)	• Signal processor architecture tradeoffs and selection • Mapping of DFG(s) to candidate architectures • Library population as necessary • Non-DFG software task identification	• Candidate architecture (s) • HW/SW allocation • Performance estimates of SW on target architecture • Tradeoff Matrix
Virtual Prototype 2 (VP2) (Architecture Verification)	• SW Autocode generation • CASE based SW development for command program • HW/SW Verification for candidate architecture • Library population as necessary	• Verified design • Full virtual prototype • Hierarchical simulation results
Virtual Prototype 3 (VP3) (Detailed Design)	• Detailed HW design • Detailed non-DFG SW development	• Fully verified virtual prototype
Processor Prototype (VP4)	• Hardware build • I&T	• Signal processor functional prototype with operational software

Section Three
Design Process Description

This section addresses the portion of the overall methodology that relates to the RASSP design process. The design process is detailed in Figure 3-1. The system definition, architecture definition, detailed design processes, and library population are addressed in individual subsections. Software is discussed in the architecture, detailed design, and library population processes as part of hardware/software codesign; it is also addressed separately to present a consolidated picture. The bold arrow between the individual processes and the library population in Figure 3-1 is meant to convey that the overall methodology is an iterative process with feedback from any process to preceding processes. The development of new library elements (software primitives or architectural elements) can be initiated anywhere within the design process, as the need arises.

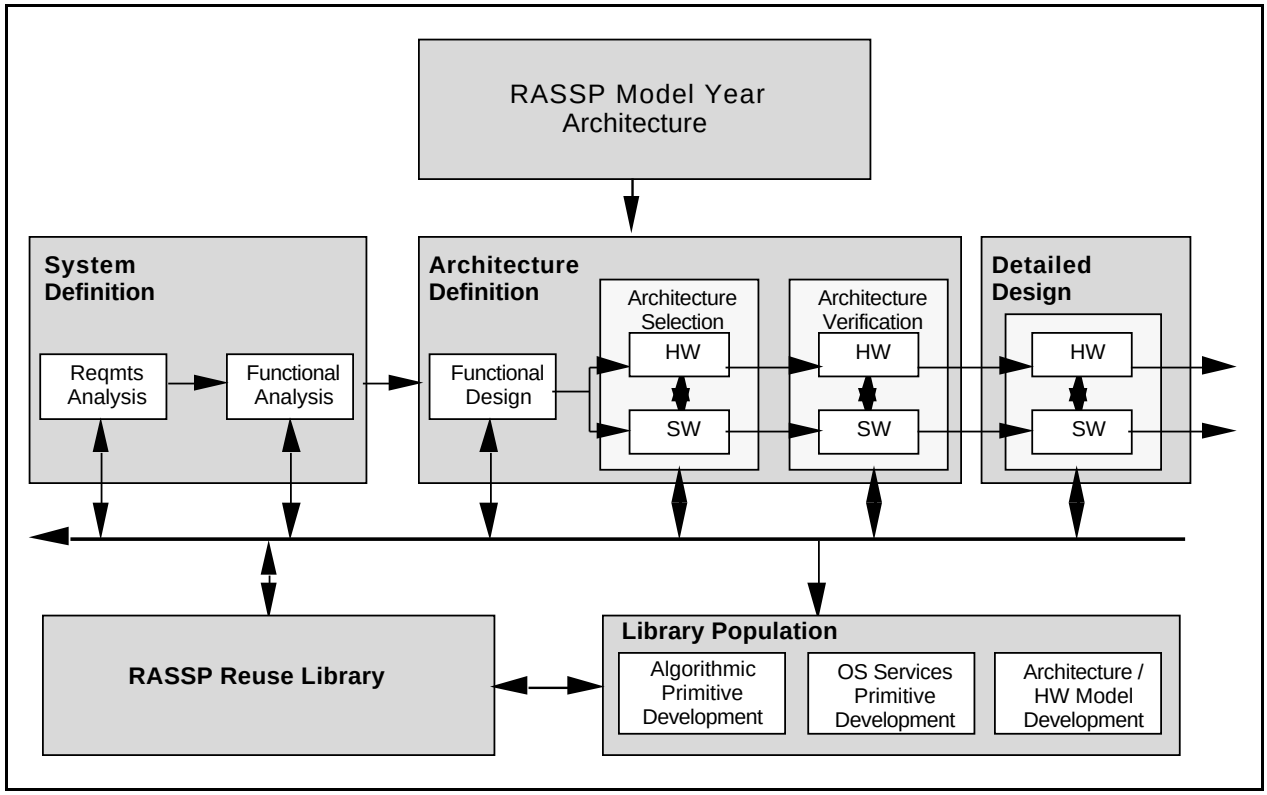


Figure 3-1. RASSP design process.

To dramatically improve the process by which complex digital systems are specified, designed, documented, manufactured, and supported requires a signal processing design methodology that recognizes a number of application domains. Within these domains are many common characteristics that can be served by the same hardware and software architectures. A key element to implement this methodology is a Model Year Architecture approach (both hardware and software) that adheres to a specific set of principles:

- The architectures must be open to promote hardware/software upgradability and reusability in other applications
- The architectures must use emerging, state-of-the-art commercial technology whenever possible
- The architectures support a wide range of applications to maintain low non-recurring-engineering (NRE) costs
- The architectures must facilitate continuous product improvement and substantial life-cycle-cost (LCC) savings in fielded system upgrades

The RASSP Model Year Architecture(s) must be supported by the necessary library models to facilitate trade-offs and optimizations for specific applications. Reusable hardware and software libraries facilitate growth and enhancement in direct support of the RASSP Model Year concept. The notion of Model Year upgrades is embodied in the reuse libraries and the methodology for their use. As technology advances, new architectural elements may be included in the library. Rapid insertion of a new element into an existing, RASSP-generated design is the goal of the Model Year concept. Details of the Model Year Architecture concept are more fully discussed in the RASSP Model Year Architecture Working Document Version 1.0

The RASSP design process is based upon the use of the reuse libraries and it has three major functional processes along with library development.

Systems Process

The system process captures customer requirements and converts these system-level needs into processing requirements (functional and performance). Functional and performance analyses are performed to properly decompose the system-level description. The system process has no notion of either hardware versus software functionality or processor implementation.

The first major cycle of the spiral shown in Figure 1-4 results in a requirements specification that was captured in an appropriate tool; it is the first instantiation of a virtual prototype (VP0). This information is then translated into simulatable functions, which we refer to on RASSP as an *executable specification*. This cycle represents the first level at which requirements are specified so that we can readily match with simulators to verify performance and functionality in an automated manner. In this phase, processing time is allocated to functions and functional behavior is defined in the form of executable algorithms. At this point, all signal processing functions are implementation-independent. High-risk items can spawn prototype analysis and development efforts in a mini-spiral. The executable specification represents a major portion of the systems design information model. This process results in a System Definition Review (SDR) based upon VP0.

A major portion of the systems behavioral definition is in terms of algorithmic functionality. Generating an executable version of the algorithms using systems-level tools is part of the systems effort. In addition, the detailed specification of all messages that must be passed to and from the signal processor must be generated, along with the definition of all required mode transitions.

Architecture Process

The architecture process is broken into functional design, architecture selection, and architecture verification. During functional design, initial performance analysis is conducted based upon the processing flows and requirements flowed down from the systems design process. Processing flows are converted to detailed data-flow graphs based upon reuse library primitives. Non-DFG software requirements and tasks are identified. The architecture selection process transforms processing requirements into a candidate architecture of hardware and software elements. This process, which corresponds to the second virtual prototype iteration (VP1) of the signal processor system, initiates the trade-offs between the different processor architecture alternatives. During this process, the system-level processing requirements are allocated to hardware and/or software functions. A non-DFG software architecture is defined, if necessary, and initial software development is begun. The hardware and software functions are verified with each other via co-verification at all steps. The architecture verification

process, corresponding to the next virtual prototype (VP2), results in a detailed behavioral description of the processor hardware and definition of the software required for each processor in the system. The intent is to verify all the code during this portion of the design to ensure hardware/ software interoperability early in the design process.

Processor behavioral and performance simulations support trade-offs. Mixed levels of simulation (algorithm, abstract behavioral, performance, ISA, RTL, etc.) are used to verify interaction of the hardware and software. These models are largely hierarchical VHDL models of the architecture. We choose the models, to the maximum extent possible, from the Model Year Architecture elements in the RASSP reuse library. The RASSP team develops and inserts new required library elements into the reuse library to support this design phase. The executable specification has now evolved into a more detailed set of functional and performance models that are architecture-specific. Software algorithm implementations are also now specific to the candidate architecture(s). We conduct an Architecture Design Review based on VP1 and VP2 when the architectural trades are completed and the design has been verified to a high degree of confidence.

The software portion of the architecture process deviates significantly from traditional (functional decomposition 2167A) approaches. The partitioned software functionality can be broken into three major areas: 1) algorithm, as specified in a flow graph; 2) scheduling, communications, and execution, as specified by mapping the graph to a specific architecture; and 3) general command/control software and other non-DFG based software. The intent of RASSP is to automate the first two to the maximum extent possible. This will be accomplished using a graph-based programming approach(es) that supports *correct-by-construction* software development based on algorithm and architecture-specific support library elements. The execution control of these graphs is provided by a run-time system that provides reusable data flow control, which extends the notion of reuse beyond signal processing primitives.

The general command/control software and other non-DFG based software development will use emerging, CASE-based code development, documentation, and verification tools. The command program interfaces with the outside world via a messaging system and translates messages into graph and I/O control commands for execution by the run-time system. We design all the non-data flow graph software required for the signal processor during the architecture process. We functionally simulate the joint operation of graph-based and non-graph based software to validate the proper interaction of the command program with the data flow graph execution.

Detailed Design

The next virtual prototype iteration (VP3) involves the detailed design of software and hardware elements. As with the prior processes, we design and verify both hardware and software via a set of detailed functional and performance simulations. When this process is completed, the design is established, resulting in a fully verified virtual prototype of the system.

During the hardware portion of the detailed design process, we transform behavioral specifications of the processor into detailed designs (RTL and/or logic-level) through a combination of hardware partitioning, parts selection, and synthesis. Detailed designs are functionally verified using integrated simulators, and performance/timing is also verified to ensure proper performance. The process results in detailed hardware layouts and artwork, net lists, and test vectors that can then be seamlessly transitioned to manufacturing and test via format conversion of the data. We generate the entire design package required for release to manufacturing for the Detailed Design Review based on VP3, which corresponds closely to today's Critical Design Reviews (CDRs).

Since most of the software developments are verified during the architecture phase, they are limited at this point to generation of those elements that are target-specific. This includes configuration files, bootstrap and download code, target-specific test code, etc. All the software is compiled and verified (to the extent possible) on the final virtual prototype before the detailed design review. Design release to manufacturing marks the end of the RASSP design process.

Software Development

In the RASSP methodology, we defined the development processes to be independent of discipline; therefore, software is not considered a separate process. It is one of the activities that occurs within the systems, architecture, and detailed design processes. It is imperative that software development progress from an isolated discipline to an integrated activity within the functional processes. There are three components to the overall RASSP software: autocode generation from the data flow graph, software library element generation, and non-graph-based software development. Within the RASSP methodology, we emphasize autocode generation, particularly for the signal processing represented by the data flow graph(s). For the non-graph-based software development, some of the emerging tools also support autocode generation. We develop software, in the sense of writing code, as part of the library population and/or non-graph-based support software development. The non-graph-based software development, which includes general command/control software, uses emerging, CASE-based code development, documentation, and verification tools, which may include some automated code generation. The section of this document on software development puts the entire software activity in context.

Library Population

Library population encompasses the development of both hardware and software models. For software, this includes generating new library elements, which are first entered into the system as *prototype* elements and later promoted to *verified* elements after prototype verification. The library generation process supports generation, testing, and documentation of all new software, which then becomes part of the RASSP component library. Software library elements include both application primitives and operating system services. In addition, there is library generation and/or modification activity required to support the addition of both computational and non-computational architectural elements to the reuse library. This may include new I/O drivers and/or generating timing information for existing primitives on new computational elements.

Sections 3.1 - 3.5 provide more detail on each of the three major RASSP design processes and an integrated software view and library population discussion.

3.1 System Definition Process

The RASSP design methodology must support the program goal of improving product cycle time, quality, and LCC by a factor of four. This design methodology is expected to evolve during the four-year program. The system definition process for RASSP is described in this section. We analyze customer requirements and perform trade-offs to determine the functional and performance requirements for each processing subsystem during the system definition process.

The RASSP system definition process starting point is the Martin Marietta system engineering design methodology developed and continuously refined on the Engineering Process Improvement (EPI) program. While we perform the same type of functional decomposition and allocation as in the traditional design process, we are making several significant extensions for RASSP that will lead to shorter design cycles. We emphasize early decision-making in the design process (versus requirement flow-down through specifications) so that prototyping efforts can begin early in the program to reduce high-risk elements. These early prototyping activities are represented by mini-spirals in the RASSP spiral model, as described in Section 1.2.1. The output of the system definition process is a set of executable specifications that have the requirements for each processing subsystem in an executable form. The executable specifications support the RASSP concept of reuse and minimize errors due to human interpretation. In addition, the process is tightly coupled to the other signal processor design activities (architecture and detailed design processes). Traceable system requirements are passed via executable specifications from the system definition process to the architecture design process. As the design progresses, the ability to meet requirements is passed back up to the system-level simulations so that the impact of processor trade-offs are analyzed.

3.1.1 System Definition Process Description

The system definition process is a front-end engineering task in which we develop signal processing concepts to meet customer requirements and we perform top-level tradeoffs to define the signal processing subsystem requirements. Depending upon one's perspective, a system could represent a platform, sensor system, signal processor or processing board. For RASSP, a system is defined at the signal processor level as shown in Figure 1-2. For this document, a “system” represents a signal processing system and a “subsystem” represents a major component of the signal processor. The system definition process for RASSP starts after the requirements have been established for the signal processing system.

The inputs to the system definition process include all the customer documentation detailing the processing system specification. The outputs of this process include the functional, performance, and physical requirements for each signal processing subsystem. Typical signal processing requirements include system mode functional descriptions (search, track, waveforms, algorithms), performance requirements (processing gain, timeline and precision requirements), physical constraints (size, weight, power, cost, reliability, maintainability, testability, etc.), and interface requirements. The system definition process is iterative, requiring constant interaction with the customer and the product development team members as the impact of system-level decisions on LCC are significant.

The system definition process is shown in Figure 3-2. We perform top-level trade-offs to determine how the system will operate and what set of subsystems are required. We develop system-level functional and timeline simulations to characterize system behavior. The output of the system definition process is a set of functional, performance and physical requirements for each subsystem. As the subsystem designs progress, key subsystem parameters are back annotated, and system-level simulations are re-run to ensure that performance is maintained.

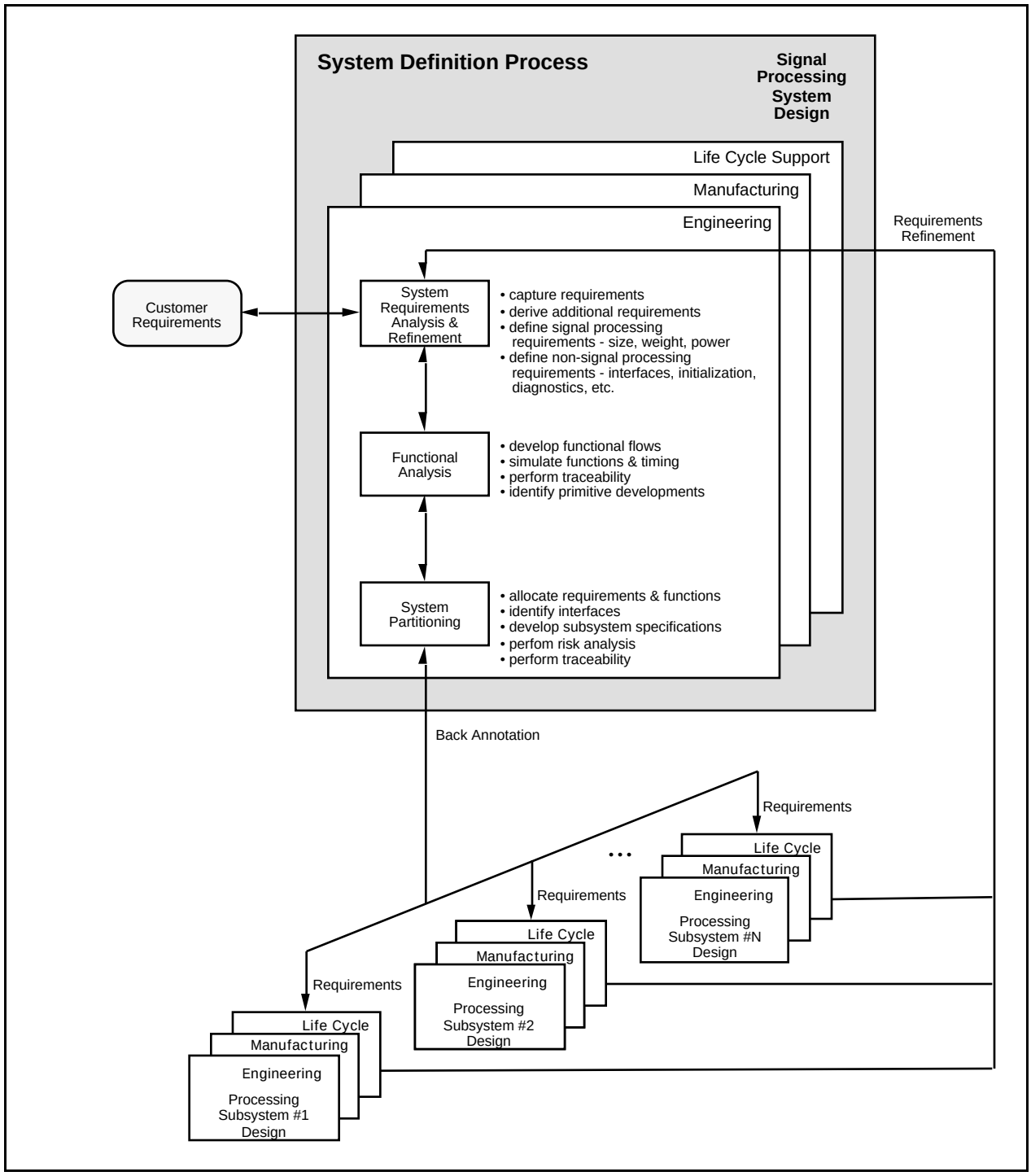


Figure 3-2. System Definition Process.

We constantly monitor subsystem requirements to make sure the development risks are balanced among the subsystems. There is a feedback path back to the system-level from each subsystem design; this path is used whenever cost-effective subsystem designs cannot be obtained. We then reexamine system requirements and perform analyses to determine a refined partitioning of the requirements.

A multidiscipline Product Development Team (PDT) performs the functions specified by the system definition process. The specific roles of the PDT team members are described in Section 3.1.4. The process consists of three main subprocesses: 1) systems requirements analysis and refinement; 2) functional analysis; and 3) system partitioning. Each of these subprocesses, their concurrency, and risk management activities are summarized in the following paragraphs.

System Requirements Analysis and Refinement

We initially examine the operational and procurement requirements to ensure that all requirements are well understood. There is close interaction with the customer to clarify any confusion with the requirements. A traceable path must be established when requirements are allocated to functions and components. Both mission and threat analyses are performed to understand how the signal processing system should behave. The system is defined by describing the system modes, functions and interfaces. We establish measures of effectiveness (cost, performance, risk, testability, etc.) for the system to provide metrics to compare different system configurations. We develop operational scenarios to determine system performance.

Functional Analysis

We decompose the system into its functional elements while establishing the requirements. This functional analysis is performed by determining what functions are required to implement each system requirement. Functions are described by defining the inputs to the function, the algorithm performed by the function, and the outputs of the function. We identify constraints and timing requirements for each function. The top-level system behavior is modeled to determine the functional performance of the system. Signal-to-noise ratios, detection ranges, probability of detection, and probability of false alarms are several examples of system-level behavior for a signal processing system. We develop system test and maintenance concepts. All functions within the system must be traced back to the system requirements.

System Partitioning

We allocate the functions of the system to subsystems as the functional requirements are established. At this point, we develop various system configurations and characterize them to determine the baseline system. Trade-off analyses are typically performed for the following areas: reliability, availability and maintainability; testability; LCC; schedule and technical risk; integrated logistics support; human factors; and system safety. All system requirements must be traceable to both functions and subsystems. The output of the system partitioning process is a set of executable specifications for each signal processing subsystem.

Task Concurrency

The system definition process steps of requirements analysis, functional analysis, and system partitioning are closely interrelated. We perform the tasks shown in Figure 3-2 concurrently, often *as a series of iterations* to trade-off alternative approaches and to successively provide greater detail. We initially examine the system requirements before functional analysis can begin. However, we develop the functional behavior of the system while the requirements are analyzed and refined with the customer and end-user of the system. A preliminary system functional baseline is required at the System Requirements Review. This corresponds to the first level of the spiral model. (The design reviews held during the system definition process are detailed in Section 3.1.5.) In addition, the system partitioning process begins after we identify the initial functional baseline. The limitations of various equipment configurations identified during system partitioning must be accounted for in the functional behavior simulations. The system definition process is an iterative process where requirements analysis, functional analysis, and system partitioning activities are performed concurrently to define the subsystem requirements.
The system definition process is closely coupled with the architecture design activities. As the signal processor design matures, quantified processor parameters such as throughput rates and precision must be back annotated into the system-level functional simulations. In addition, the system activities continue

throughout the signal processor design. We continuously monitor the allocation of system requirements to subsystems is to ensure that the development risks are balanced among the subsystems.

Risk Management

Risk management is an integral part of the system definition process as sources of technical, schedule, and cost risk are identified. We determine the probability of occurrence for each risk and the resulting impact on performance, development schedule, and LCC. We develop risk reduction plans for the risks likely to occur or those that have a critical impact on the program. We analyze risks to determine their interrelationships. The key to risk management is to identify and quantify the individual risk elements so that the overall system risk can be set at an acceptable level. We perform risk analysis for all trade studies as we develop and allocate the functional requirements during the system definition process. We identify and track key design parameters as the system is developed to monitor the risks throughout the program. Risk management activities are formally examined during all design reviews.

The RASSP methodology provides several key improvements over traditional system processes. The output of the system definition process is a set of executable specifications for the signal processor design activity. The definition and functions encompassed by the executable specification is receiving focused attention on the RASSP program. A hierarchical set of simulations is performed at each design level, and the results of these simulations are back annotated in the higher-level simulations. In RASSP, we emphasize considering LCC early in the design process and the reuse of library elements at the system and subsystem-levels.

3.1.1.1 System Requirements Analysis

System requirements analysis, shown in Figure 1-4, converts a user need into a combination of elements that satisfies that need. It involves analyzing customer documentation and conducting discussions with the customer to refine the purpose and manner in which users will operate the system. It is designed to determine what the system is to do and how the system is to be used. We then determine the system requirements from a user's point of view. We identify external interfaces to the system, usage scenarios, and capacities, and determine methods to verify each requirement statement. The process iterates with functional analysis and system partitioning efforts to assess feasibility and to structure the requirements cost-effectively. Unnecessary implied designs should not be incorporated in the requirement statements. This iteration also makes the verification process more accurate and cost effective by eliminating ambiguity in the requirements statements.

Baselines are used to control the development process. We establish the functional baseline following the system design phase, which prescribes: all functional and performance characteristics; all tests required to demonstrate achievement of functional and performance characteristics; interfaces between subsystems and their functional characteristics; and design constraints. Although the functional baseline is not established during the system requirements analysis phase, preliminary data is provided. We examine functional elements within the RASSP reuse library to determine whether any elements are applicable for the baseline. Several baselines can be maintained simultaneously while we evaluate system alternatives. Typically, the initial baseline is in the form of a requirements database. Formal configuration management is not used until a single baseline has been established. The baseline will be reopened whenever the subsequent design, integration, or test activities indicate the requirements cannot be supported or when customer change requests result in a modification to the requirements.

The system requirements analysis process, shown in Figure 3-3, is composed of three primary subprocesses: system requirements development, system specification generation, and system requirements review. Each subprocess is described in the following paragraphs.

System Requirements Development

We assess system requirements for completeness and consistency using the steps in Figure 3-3. Each step is performed iteratively to determine the system requirements. A complete set of

customer documents must be used to determine the system requirements because the contractor must understand how users intend to use the system. We define the system in terms of its system modes and states, functions, and interfaces. Trade-offs establish alternative performance and functional requirements to meet customer needs. Any potential conflicts in the trade-off results with the system requirements are resolved. We generate a workoff plan for all TBD/TBR items that identifies the responsible individual, schedule for resolution, risk analysis, and key trade-offs to be performed. Traceability of system requirements and decisions ensures that the trade-off decisions made in generating the system requirements can be tracked and that these requirements are completely and accurately reflected in the final design. Traceability is also used to assess the impact of changes at any level of the system. We trace all system-level requirements to their source during this process.

System Specification Generation

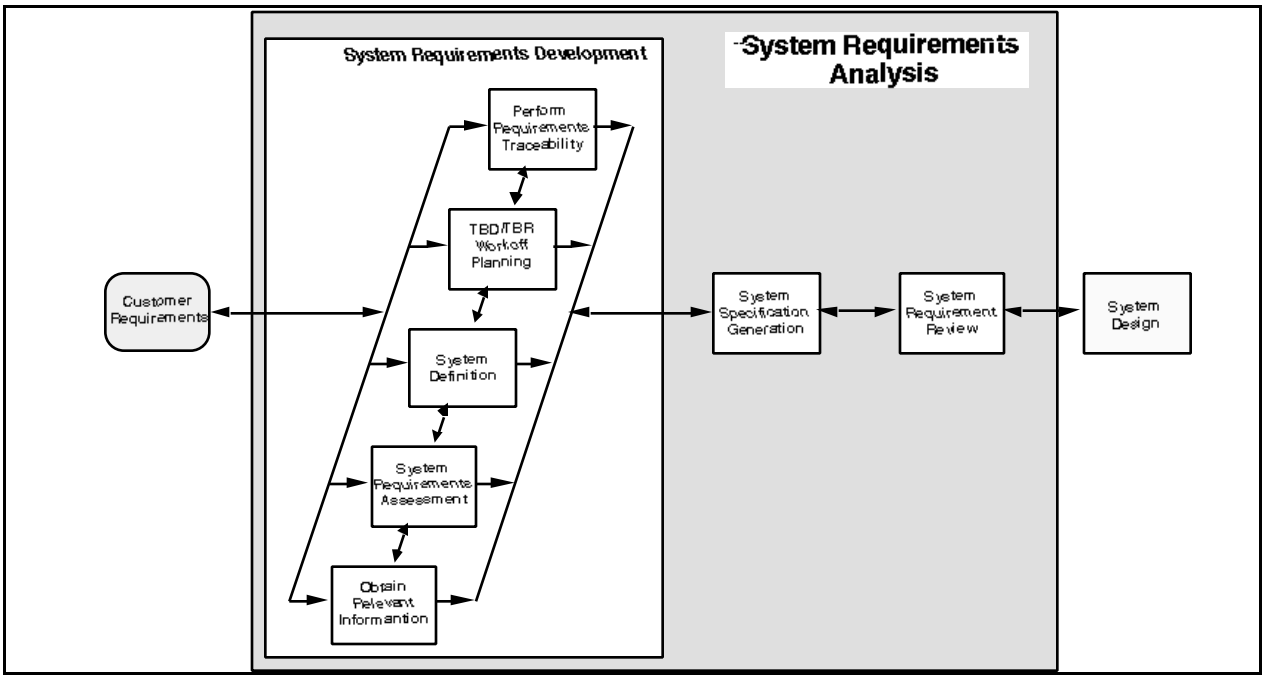
The processing system specification has the technical requirements for the system, allocates requirements to functional areas, documents design constraints, and defines interfaces between functional areas. It also states all necessary requirements in terms of performance, including test provisions to ensure that all requirements are achieved. Essential physical constraints and requirements for application of any known specific equipment must be included.

System Requirements Review

We conduct a System Requirements Review (SRR) to ascertain the adequacy of the contractor's efforts in defining the processing requirements. It is conducted when a significant portion of the system functional requirements has been established.

3.1.1.2 Functional Analysis

As we establish the system requirements, we define the system configuration (the components within the processing system) through functional analysis and system partitioning processes. Functional analysis is designed to identify the functions, decompose the functions into requirements, and allocate these requirements to lower-level functions. System partitioning takes the functions from the functional analysis process and allocates them to entities within candidate configurations. These allocations are analyzed to determine the performance, cost, and schedule impacts of the alternatives. We eliminate unsuccessful allocations and configurations and select the best alternatives as a final configuration.



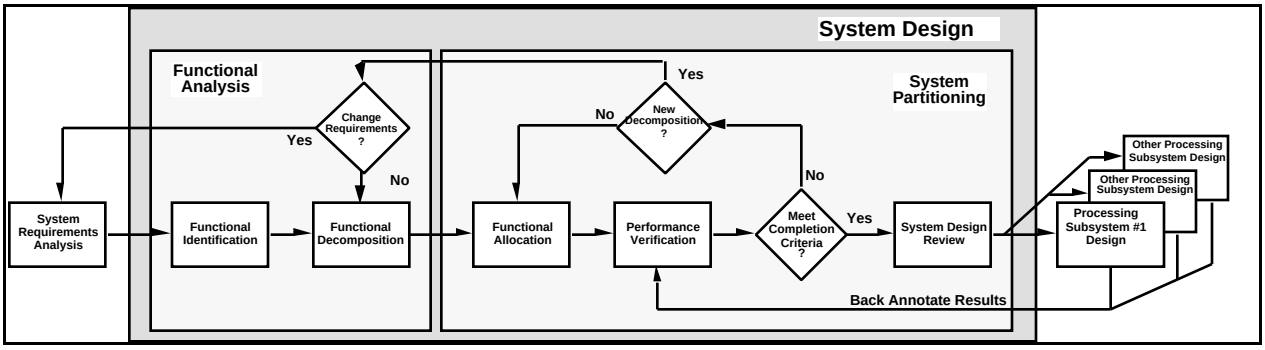
Obtain Relevant Information	System Requirements Assessment	System Specification Generation
• Obtain all relevant customer documents • Discuss system requirements with customer • Obtain data on applicable technology	• Assess functional & performance requirements, operational environment, system constraints & measures of effectiveness • Perform consistency & completeness analysis • Perform trade-offs to derive and balance system requirements • Identify clarification & change requests • Conduct internal review	• Analyze inputs • Prepare specification outline • Incorporate applicable documents & applicable standards • Prepare preliminary specification • Conduct internal review • Incorporate initial comments • Submit for customer review • Incorporate customer comments • Place under configuration control • Submit for authentication

System Definition	TBD/TBR Workoff Planning	System Requirement Review
• Define system modes & states • Define system functions • Define system configuration items • Define system interfaces	• Identify responsible individual • Perform risk analysis • Develop schedule for resolution • Identify potential trade-offs • Determine resolution criteria • Assemble workoff plan • Conduct internal review	• Prepare data package • Prepare meeting agenda • Prepare presentation material • Conduct review • Prepare minutes of meetings • Resolve action items

Perform Requirements Traceability
• Capture customer requirements • Trace system requirements • Prepare traceability matrix • Conduct Internal review

Figure 3-3. System requirements analysis.

The functional analysis process describes the requirements as a set of verifiable (simulatable) statements that can be used as a basis for system design. The functions, constraints and performance statements are decomposed to a detailed level that can be allocated to specific candidate design configurations. The output of this process is a functional baseline that describes the system functionality and is the basis for eventual customer sell-off. The functional analysis process is shown in Figure 3-4. This process is composed of two steps, functional identification and functional decomposition, which are described in the following paragraphs.



Functional Identification
• Analyze system requirements • Identify system functions • Develop functional block diagrams • Identify constraints • Establish performance timelines • Evaluate baseline for consistency and completeness • Perform traceability to requirements • Provide rationale for identification • Develop alternative functional configurations

Functional Decomposition
• Perform trade-offs to eliminate poor system configurations • Identify next tier functions • Develop next tier functional block diagrams • Identify next tier constraints • Establish next tier performance timelines • Perform traceability to upper tier functions and requirements • Provide rational for decomposition • Identify new library primitive elements

Functional Allocation
• Create candidate system configurations • Allocate functions to configuration • Allocate constraints to configuration • Identify interfaces • Perform traceability to functions and requirements • Provide rationale for allocation • Identify and quantify risks • Identify candidate prototyping activities

Performance Verification
• Develop metrics • Develop and configure models • Develop scenarios • Estimate unknown parameters • Execute models • Evaluate results • Perform trade-offs to eliminate poor configurations • Reiterate process if completion criteria not met • Perform traceability • Document results • Support make/buy decision for each subsystem • Conduct internal review

System Design Review
• Prepare data package • Prepare meeting agenda • Prepare presentation material • Conduct review • Prepare minutes of meetings • Resolve action items

Figure 3-4. System design.

Functional Identification

The functional identification process translates system requirements, customer heritage, and customer rationale into functional block diagrams that are used by subsequent processes to create and evaluate system configurations. This step refines and decomposes the functions identified from the system requirements analysis step. Functional identification also develops the design constraints under which the system must operate. We examine functional elements within the RASSP reuse library to determine whether existing library elements can be used. The key steps in the functional identification process include: analyzing system requirements; identifying top-level system functions; developing functional block diagrams; identifying constraints; establishing performance timelines; developing alternative functional configurations; evaluating the functional baseline for consistency and completeness; performing traceability of the functions to the system requirements; and providing the rationale for the top-level functional description.

Functional Decomposition

Functional decomposition creates lower-level functions, constraints, performance, and interfaces from those that currently exist. This more detailed information is used by the subsequent partitioning and architectural trade process. This process is structured and accomplished iteratively to eliminate poor allocation decisions as early in the analytic process as possible. We examine functional elements within the RASSP reuse library to determine whether existing library elements can be used. If any of the system functions cannot be described by existing library elements, then we identify new primitives for development. The specific steps in functional decomposition include: performing trade studies to eliminate poor system configurations; identifying the next-tier functions and functional block diagrams; identifying next-tier constraints; establishing next-tier performance timelines; performing traceability to the upper level tier functions and requirements; and providing the technical rational for the functional decomposition.

3.1.1.3 System Partitioning

During the system partitioning process, we define and evaluate candidate system configurations to determine which configurations most effectively meet the functional and system requirements. As many configurations as feasibly possible should be evaluated in enough detail to rank the alternatives. A structured method must be developed to quickly identify the feasibility of a specific candidate configuration. We typically develop measures of effectiveness for a specific program as the basis to evaluate trade studies. The output of the system partitioning process is the set of functional, performance, and physical requirements for each subsystem in the baseline configurations. For RASSP, an executable specification for each signal processing subsystem is the starting point for the architecture selection process. This executable specification is the first virtual prototype (VP0) of the subsystem. As shown in Figure 3-4, the system partitioning process is composed of three subprocesses: functional allocation, performance verification, and system design review. Each of these subprocesses is described in the following paragraphs.

Functional Allocation

The functional allocation process allocates functions from the functional block flows and constraints to entities in a specific candidate design. This process iterates until an allocated baseline can be established. This process is complete when all functions are allocated to subsystems and all requirements and constraints are mapped through functions to subsystems. This process may require many iterations to refine the system configuration. Trade-off analyses assess the risk and LCC impacts for each alternative system configuration. A key feature in RASSP methodology is the ability to quickly assess the LCC for various system configurations. Design decisions and rationale must be documented as the functions are allocated. The specific steps in the functional allocation process include: creating candidate system configurations; allocating functions and constraints to components; identifying interfaces between the subsystems; performing traceability of the system configuration to system and functional requirements; and providing the technical rationale for the functional allocation.

Performance Verification

The performance verification process supports the system synthesis process by providing evaluation criteria to determine which candidate configuration provides more effective performance. This effort must consider all factors of interest to the product development team: technical performance, risk, LCC, producibility, supportability, testability, etc. Critical inputs to the evaluation process are the operational concept, operational scenarios, operational maintenance concept, and usage rates expected by the user. The performance evaluation must reflect objective, demonstrable evaluation metrics and must assure the customer we considered all alternatives. The functional verification process models the allocated functions of candidate configurations to determine whether performance requirements are met. The set of mathematical elements within the RASSP reuse library is used (whenever possible) to quickly construct the functional-level simulation to verify technical performance. This process iterates with the functional

allocation process until the performance of a candidate configuration meets the completion criteria established during the systems requirements analysis process. If no candidate configuration meets the completion criteria, the procedure reverts back to the functional decomposition process for an updated decomposition. If no candidate configuration meets the completion criteria for multiple decompositions, the process reverts back to the system requirements analysis process for requirement refinements. The specific steps in the performance verification process include: developing metrics, models and scenarios; configuring models; estimating unknown parameters; executing models; evaluating results; eliminating poor configurations; reiterating process if completion criteria is not met; performing traceability to requirements; documenting results; supporting the make or buy decision for each subsystem; identifying candidate prototyping activities to reduce program risk; and conducting an internal review.

The RASSP architecture selection process (described in Section 3.2) transforms the processing requirements for each processing subsystem into a candidate processing architecture of hardware and software elements. The architecture selection process overlaps with the system definition process during the later portions of the system partitioning activity. A set of executable specifications is used to transfer the signal processor requirements to the architecture selection process. A hierarchical set of simulations is performed at each design level, and the results of these simulations are back annotated in the higher-level simulations to verify that performance is maintained.

System Design Review

A System Design Review (SDR) is conducted to ascertain the adequacy of the contractor's effort in defining the baseline system. This review is held after the performance of the candidate system configuration has been verified to meet system requirements.

3.1.2 Use of VHDL in System Definition Process

The output of the system definition process is the set of executable specifications for the DSP system. The signal processor specification contains requirements that can be divided into three categories:

1. Timing/Performance (e.g. processing latency, throughput, I/O timing)
2. Function (e.g., algorithms, control strategies)
3. Physical constraints (e.g. size, weight, power, cost, reliability, maintainability, testability, scalability, temperature, vibration)

VHDL is applicable for conveying system function, timing, and performance information and is used to develop a test bench and model of the signal processing system. Many efforts, some of which are being funded under RASSP, are focusing on conventions for associating physical information with VHDL models. As they become available, the methodology will expand to include their use with VHDL.

The system model captures the specification for the timing, performance, and function of the DSP system and its interfaces. It also contains a structural specification of its interfaces.

The test bench provides system stimulus and checks that the applicable system requirements are satisfied. As in all cases, the VHDL test bench is developed before the model it is intended to test. This ensures a thorough analysis and understanding of the requirements before and during design. In essence, the test bench is an interpretation of the aspects of the system requirements that can be described in VHDL, while the system model is an expression of the design solution that satisfies the requirements.

The system model forms an executable specification at the system definition level. The executable specification supports design-for-test (DFT), since the system test concepts must be considered and developed for its verification. The performance model is periodically back annotated with timing and performance data from the more detailed design levels. To continually ensure that system requirements are being met, the performance model is executed within the test-bench to verify continued compliance with the system requirements.

The executable specification consists of a test bench and a system model described in the following paragraphs.

Test Bench

The test bench provides test procedures, stimuli, and expected responses for verifying that the system model meets system requirements. The test bench is developed before, and is executed on, the system model.

System Model

The system model describes: system timing, performance, and, when possible, system functionality and physical constraints in a way that facilitates automatic testing to verify compliance with system requirements. Elements within the system model are summarized in Table 3-1. The algorithm descriptions are in terms of the basic arithmetic operations to be performed. The arithmetic operation sequence is typically expressed in the form of a data flow graph (DFG) that indicates control flow and data dependencies.

3.1.3 Design For Test Tasks in System Definition

Requirements are derived from customer and/or parent system requirements and maintained during the system definition process. All of the functions including test functions such as BIST are partitioned into processor system and subsystem functions. An overall model of the processor system is developed which is referred to as VP0. This model together with testbenches represents the inputs, outputs and transformations of the inputs by the processor system including latency. Key outputs of the DFT efforts are the consolidated test requirements (which promotes a singular test strategy across design verification, manufacturing test and field support), preliminary test strategy diagram, TSD0, and testability architecture, TA0. Figure 3-5

shows the DFT steps which occur during the system definition step. The key step, requirements analysis, is shown in more detail in Figure 3-6.

Table 3-1. Elements within the system model.

System Timing and Performance Data	System Functionality Data	Physical Constraint Data
• Signal Processing I/O Data - I/O Timing Constraints - I/O interface structures - I/O protocols - Signal levels - Message types • Signal Processing Latency - Data Acceptance Rate • Signal Processing Stimuli Response	• Algorithm Descriptions • Control Strategies • Task Execution Order • Synchronization Primitives • Inter-process Communication (IPC) • BIT and Fault Diagnosis	• Size • Weight • Power • Cost • Reliability • Maintainability • Testability (fault coverage, diagnosis, and BIST goals) • Repairability • Scalability • Environment Constraints - Temperature - Vibration - Pressure - Stress and Strain - Humidity - EMI/EMF/EMP

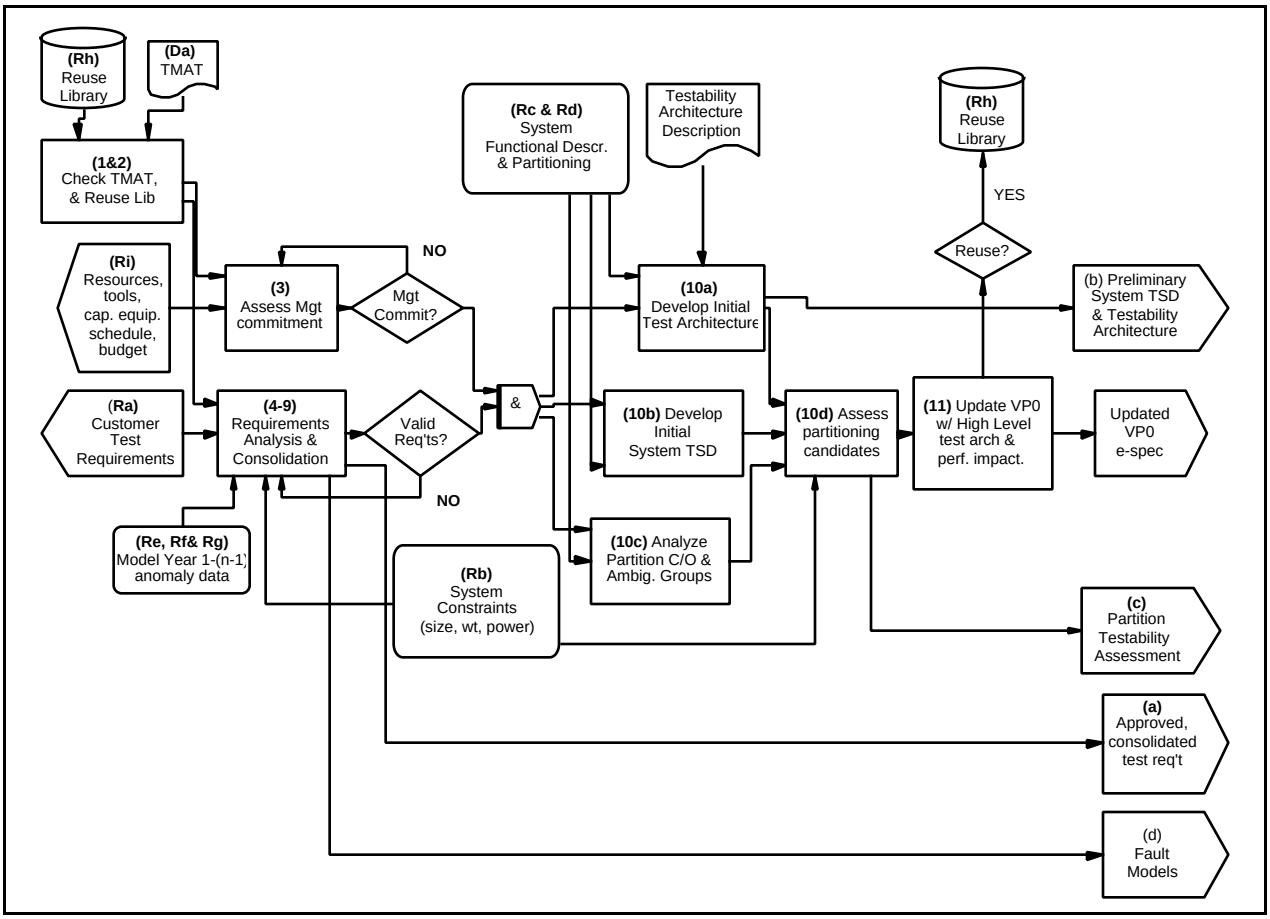


Figure 3-5. DFT steps in system definition flow diagram.

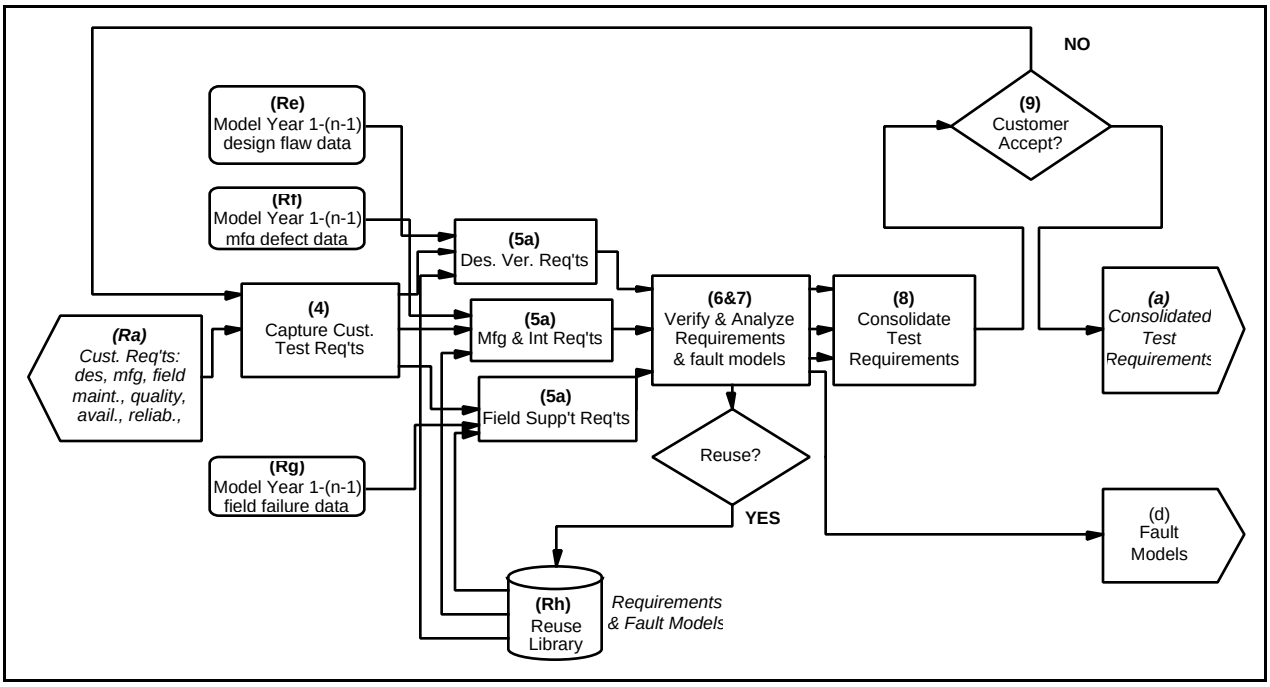


Figure 3-6. DFT requirements analysis flow diagram.

The presence of COTS may throttle attempts to reach a singular life cycle test strategy. One reason is that black box COTS elements are usually tested functionally (for defect detection and isolation), while most of the rest of the design that is not black box oriented is usually tested using a great deal of structural testing and some functional testing. An analysis of the degree of anticipated COTS usage and the extent to which current COTS incorporates test features, would tend to allow an early, preliminary assessment of what plateau of test architectures might be possible

The impact of using COTS components during system definition is to place practical limitations on the test requirements and subsequent test strategies. For example, it does not make sense today to specify a requirement that can only be achieved with 100% boundary-scan, if a large part of the system will use COTS boards that do not come in boundary-scan versions.

The development of the fault model in this step is also affected by the presence of COTS, particularly if the COTS element is a "black box" in terms of its structural composition. Several options exist for handling fault models for black box COTS elements:

1. Establish a structural fault model, in which the "node" is the lowest level input or output for which a description exists. This level will usually be at the inputs and outputs of blocks at the block diagram level or could be simply the inputs or outputs of the package itself (such as a chip) or board.

2. Attempt to define a "functional fault model." This is a fault model based on the function, rather than the structure of the item; and it describes the erroneous functional behavior that would occur in the presence of a fault. While this approach will work for simple functions, it is not likely to be practical for large VLSI based components or boards.

See Section 3.2.1 of the DFT Methodology for further details of the System Definition Step.

3.1.4 Role of the PDT in System Definition Process

A multifunctional PDT is required to perform the system definition process. The disciplines needed for this process in RASSP include systems, architecture, software, digital hardware, mechanical, manufacturing, test, producibility, sourcing, logistics support, and cost analysis. The role for each discipline is described in the following paragraphs.

- Team Leader — The team leader for the system definition process is the technical director for the project. The technical director is responsible for the overall integrity of the system-level design. The technical director coordinates all activities during the system definition process. The technical director leads system integration and technical management activities.

- Systems — The systems engineers have the lead role in performing the system definition process. The system engineers perform the system-level design, interpret the customer requirements, control development of the subsystem specifications, and coordinate system and subsystem trade studies.

- Architecture — The architecture engineers perform the hardware/software trade-offs to determine the processing architecture within the signal processor subsystem. During the system definition process, the architecture engineers participate in the system-level trade-offs that allocate the system functional requirements to each processing subsystem. The architecture engineers determine the impact on LCC on the signal processor for various functional and physical requirement allocations to the processor.

- Software — Software engineers have a limited role during the system definition process since allocation of functional requirements to software elements is not performed during this process. The hardware/software codesign activity is performed during the architecture selection process in the RASSP design methodology. During the system definition process, the software engineers analyze the system-level software requirements to determine the impact of these requirements on the system design.

- Digital Hardware — Digital hardware engineers have a limited role during the system definition process since allocation of functional requirements to hardware elements is not performed during this process. The hardware/software codesign activity is performed during the architecture selection process in the RASSP design methodology. During the system definition process, the hardware engineers analyze the system-level hardware requirements to determine the impact of these requirements on the system design.

- Mechanical — The mechanical engineers ensure the structural and thermal integrity for the system during all phases of the product's life cycle. During the system definition process, mechanical engineers analyze the operational requirements for the system and perform the top-level tradeoffs to determine the structural and packaging requirements for each subsystem. The mechanical engineers assess the impact of various packaging approaches on LCC. The mechanical engineers monitor the structural and thermal trade studies performed during the subsystem designs.

- Manufacturing — Manufacturing personnel define the integrated manufacturing program used to build the system. During the system definition process, manufacturing personnel examine the functional and physical requirements allocation to each subsystem and assess the impact of these requirements on current manufacturing processes. Manufacturing personnel participate in the make/buy decisions for each subsystem during the system definition process.

- Test — The test engineers develop the test procedures for the product throughout the life cycle. During the system definition process, the test engineers examine and determine the system-level test requirements needed to determine whether the product is functional working on the manufacturing floor, in the depot, and in the operational environment. The system-level test requirements are then decomposed and allocated to each subsystem during the system definition process. These test requirements must be an integral part of the entire design process for the system and each subsystem.

- Producibility — The producibility engineers ensure that the system can be safely built and maintained. During the system definition process, the producibility engineers examine the allocation of functional and physical requirements to each subsystem to make sure that the resulting system-level designs are producible. The producibility engineers ensure that the system meets all reliability and maintainability (R&M) requirements. Producibility engineers analyze the R&M requirements and perform top-level trade-offs to allocate these requirements to each subsystem. Producibility engineers work with each of

the PDTs throughout all phases of the design to make sure that producibility issues are considered as the design matures.

- Sourcing — Sourcing personnel acquire the materials and purchased services needed to design and fabricate the system. During the system definition process, sourcing personnel develop the overall sourcing plan for the project. Sourcing personnel participate in the make/buy decisions for each subsystem during the system definition process.

- Logistics Support — Logistics support personnel ensure that the system is supportable throughout its entire life cycle. During the system definition process, logistic support personnel examine the supportability operational requirements and perform trade-offs to determine the supportability requirements for each subsystem. Logistic support personnel participate in the LCC analyses for the system.

- Cost Analysis — The cost analysts analyze the LCC throughout the design cycle. During the system definition process, the cost analysts support the trade-off studies by performing cost estimates for each candidate processing system. Development, unit production, and support costs are estimated for each processing subsystem in the candidate baseline system configurations. These costs are tracked and refined throughout the design process.

3.1.5 Design Reviews in System Definition Process

Periodic technical reviews are held to demonstrate that required accomplishments have been successfully completed before proceeding beyond critical events and key program milestones. There are two main design reviews during the system definition process: System Requirements Review (SRR) and System Design Review (SDR). In addition, the system members of the PDT participate in the three design reviews during the signal processor development activities: Architecture Design Review (ADR), Detailed Design Review (DDR), and Production Readiness Review (PRR). The plan for conducting each of these reviews is contained within the system engineering management plan (SEMP). Each major review should address the following issues:

- System engineering process outputs and traceability to customer needs, requirements and objectives
- Product and process risks
- Risk management approach
- Cost, schedule, performance, and risk trade-offs performed
- Critical parameters that are design cost drivers or have a significant impact on readiness, capability and LCC
- Trade studies conducted to balance requirements
- Confirmation that accomplishments in the systems engineering management schedule (SEMS) have been completed.

A multidisciplinary team from the government, contractor, and applicable subcontractors should be involved in the design review process. The two design reviews conducted during the system definition process are described in the following paragraphs.

<u>System Requirements Review (SRR)</u>

The objective of the SRR is to determine that the customer requirements have been properly analyzed and translated into system specific functional and performance requirements. Typically, the SSR is held after the requirements have been analyzed, a preliminary functional analysis performed, and the requirements initially allocated to subsystems. During the SRR, we identify and quantify risks, and address approaches to manage these risks. We identify key technologies essential to system success and candidate technologies not viable for the system. We present the progress of on-going technical verifications, the preliminary functional analysis of the system, and the draft allocation of requirements to subsystems. The draft system specification is reviewed. During the SSR, we establish understanding of the customer requirements by identifying a complete functional baseline.

System Design Review (SDR)

The objective of the SDR is to ensure that the process used to determine the functional and performance requirements for the system is complete. We accomplish this by addressing the primary product and processes, by demonstrating a balanced and integrated approach to the development of the functional and performance requirements, by reviewing the audit trail established from the customer requirements, and by substantiating any requirement changes made since the SRR. The SDR is held after all system-level trade-offs have been performed, which resulted in an optimum allocation of requirements and functions to each subsystem.

During the SDR, we verify the performance, availability, and design suitability for critical products and process technology. We assess the adequacy, completeness and achievability of proposed system functional and performance requirements and present evidence to confirm that the system requirements can be met by the proposed system. The functional baseline for the system is established. We identify the draft specification tree, work breakdown structure, and risk handling approach for the next phase of the program. Pre-planned product and process improvements and acquisition strategies are presented.

3.2 Architecture Definition Process

The architecture definition process is a new hardware/software codesign process in the RASSP methodology for high-level virtual prototyping and simulation. Traditionally, the architecture definition has been performed partially in the systems design and partially in the hardware design. Since RASSP is attempting to formalize hardware/software codesign, we have redefined the major processes by product maturity versus functional area into systems, architecture, and detailed design. Hardware/software codesign encompasses architecture and detailed design. The primary concern in the architecture definition process is to select and verify an architecture for the signal processor that satisfies the requirements passed down from the systems definition process. Although the architecture must include everything required in the signal processor, including any control processors which may be required, this discussion

focuses on the approach to defining the number and connectivity of the signal processors required to meet the processing requirements. The overall task is to:

- Define and evaluate various architectures
- Select one or more for detailed evaluation that appear to meet the requirements
- Validate the chosen architecture(s) for both function and performance before detailed design

Concurrently, we evaluate each selected architecture for size, weight, power, cost, schedule, testability, reliability, etc. so that we can make a more informed assessment. We simulate software and architecture performance concurrently so that software performance has a direct impact on the selection of an architecture. This direct coupling through co-simulation immediately inhibits any inclination to design the hardware and make the software fit later.

The architecture process is library based and DFG-driven. Reuse of both architecture elements and software primitives significantly shortens the design cycle. We construct candidate architectures from library models and partition and map the signal processing software to the candidate architecture. We construct a VHDL simulation for the architecture, and then simulate it to estimate performance. This process is iterated at the high level to achieve one or more satisfactory designs.

The process is supported by an integrated set of tools coupled to both the higher-level system tools and the lower-level design tools to foster iterative design and risk reduction. This results in a more detailed evaluation of architectures (from all perspectives) at the early stages of design, which eliminates the false starts that are costly later in the design cycle.

At the conclusion of the systems/subsystems process, a processing requirements specification (both functional and performance) that describes the signal processing functionality is provided. Included in these requirements is a description of all the required interfaces between the signal processor and the outside world, such as mode transition requests, signal processing parameter observation and/or parameter setting, and I/O initiation or termination requests. All requirements flowed down from the systems process are traceable throughout the architecture definition process. We evaluate candidate architectures against these requirements during the hardware/software codesign process. Updated values of the requirements metrics for candidate architectures are passed back to the systems process. In the event that the requirements cannot be met, we conduct additional trades at the systems level. Note that these inputs to the architecture definition process are completely independent of architecture/processor implementation. It is the objective of the architecture process to transform these processing requirements into candidate architectures and to select and verify an implementation approach.

The architecture definition process transforms processing requirements into a candidate architecture of hardware and software elements, through hardware/software codesign and co-verification at all steps. As such, the architecture definition process incorporates both hardware and software aspects. This results in a detailed behavioral description of the processor hardware and the appropriate software required to verify these descriptions. In addition, this portion of the process also develops and verifies the implementation-specific portions of both the application (algorithm) and control/support software.

As part of the RASSP methodology, as much design, costing, testing, and manufacturing information as possible will be used throughout the architecture definition process through concurrent engineering that is supported by tool integrations and design advisors which cover a

wide range of disciplines. The architecture process supports a more formalized trade-off process in which all these disciplines provide valuable inputs.

3.2.1 Architecture Definition Process Description

The architecture definition process for RASSP, shown in Figure 3-7, is composed of three steps: functional design, architecture selection, and architecture verification. The process strives to provide a comprehensive hardware/software codesign capability, where 1) hardware and software are partitioned using interactive trade-off analyses, and 2) the partitioned software is verified (functionality and performance) using simulation before verification on the final target hardware. Likewise, hardware functions are verified via simulation before detailed hardware design. An iterative, hierarchical simulation process is used to perform this verification at several levels of complexity.

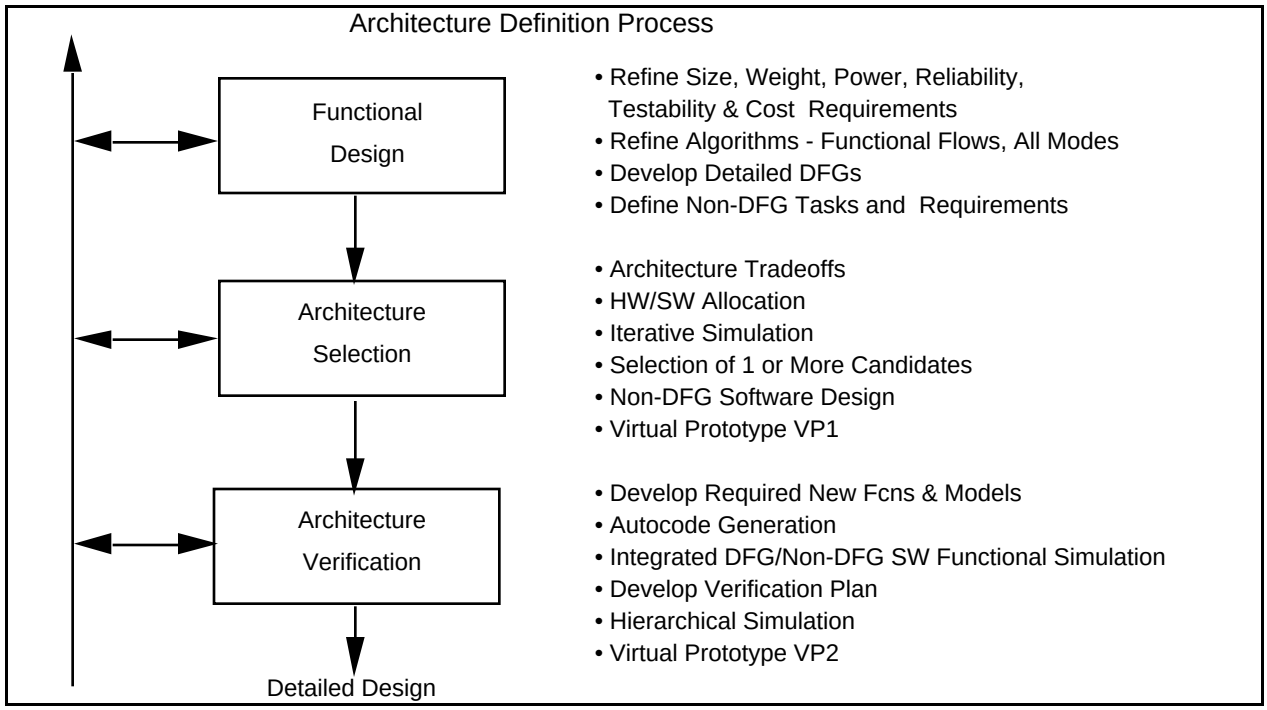


Figure 3-7. RASSP architecture definition process.

The functional design step provides a more detailed analysis of the processing requirements, resulting in initial sizing estimates, detailed data and control flow graphs for all required processing modes to drive the hardware/software codesign, and the criteria for architecture selection. The control flow graphs provide the overall signal processor control, such as mode switching (referred to as the command program). Functional simulators support the execution of both the data and control flow graphs. For complex control applications, these simulations can be coupled to ensure that all control is properly executed and results in the proper graph actions (e.g. mode transitions).

During architecture selection, we evaluate various candidate architectures through iterative performance simulation and optimize them to appropriate levels of detail. A trade-off analysis based on the established selection criteria results in the specification of the detailed architecture, and software partitioning and mapping. As part of the trade-off analysis, we use information from as many disciplines as possible (either manually or through design advisors) to populate the trade-off matrix. This portion of the process is heavily dependent on the reuse of architectural (hardware and software) components to provide significant time-to-market improvements. In addition, during architecture selection, we design all software not represented by the DFGs. Based on the requirements, the non-DFG software may include BIT, downloading, and diagnostics. The virtual prototype, VP1, produced during architecture selection is not a full system prototype, since function and performance are simulated independently and may or may not be coupled with the overall control mechanism.

During architecture verification, we develop the next level of detail for one or more architectures that meet the requirements and satisfy the established selection criteria. We develop and validate any required library elements (either hardware or software) at this time if they are not completed concurrently. Autocode generation is performed for the DFG based software, and the non-DFG based software (particularly the command program) is developed. The next level of performance simulation should include timing estimates for the generated code, along with a representation of the operating system services, scheduling, and run-time system overhead. We generate a hierarchical validation plan that ensures that all component interfaces are tested. Detailed hierarchical simulations are performed to verify both functionality and performance on the target architecture. The virtual prototype, VP2, produced during architecture verification, represents a functional and performance description of the overall design.

The arrow on the left side of Figure 3-7 indicates that the process has feedback within the architecture definition process, between the architecture and systems processes and between detailed design and the architecture processes. In fact, as shown in Figure 1-5, various portions of a design may be at different levels of maturity, which implies that more than one of the processes may be active concurrently. Note that as a design progresses, new development activities (mini-spirals) may be initiated. For example, during functional design it may be perfectly obvious to the designer that 1) custom hardware is required for some portion of the processing; or b) that the reuse library does not contain all the software primitives required to construct the detailed DFG. In either case, the architecture selection process can proceed by postulating hardware or software elements that can be used in the high-level architecture performance simulations (e.g., defining a new element with 2X performance of existing element). Concurrently, we can initiate new activities to start the hardware modeling effort required for the custom hardware or start the development and validation process for a new software library element.

A hardware and software component reuse library has models and data at various levels, as shown in Table 3-2. These models support concurrent codesign throughout the selection and verification process. The reuse library drives both the architecture synthesis and the software synthesis processes in an integrated fashion.

Table 3-2. Hardware and software reuse library.

Software Reuse Library	**Hardware Reuse Library**
• SW performance models	• Performance models
• Application code/code fragments	• Behavioral models
• OS kernel(s)/OS services	• DFG partitions and mappings
• Application DFGs	• Architecture configurations
• Control/support software	• Test plans and test sets
• Test data	• Documentation elements
• Documentation elements	

The following sections describe the steps in the architecture definition process.

3.2.1.1 Functional Design

Initial efforts in architecture definition include an implementation analysis of the algorithms to assess the required operations per second and memory and I/O requirements. Processing flows may be optimized based upon implementation experience and knowledge of reuse libraries. The two primary functions in functional design are formalizing the criteria to select an architecture and translating the processing flows to an architecture-independent DFG constructed from reusable library elements. The functional design process is also an opportunity for the signal processing architect(s) to assess the complexity of the required processing. The goal is to take the functional algorithms and translate these into preliminary implementation form. Initially, we size and establish a criteria to select the architecture, as shown in Figure 3-8. We translate processing flows for all modes to architecture-independent DFGs constructed from reusable library elements, which may represent either hardware or software. In addition, we translate control requirements into the appropriate control flows.

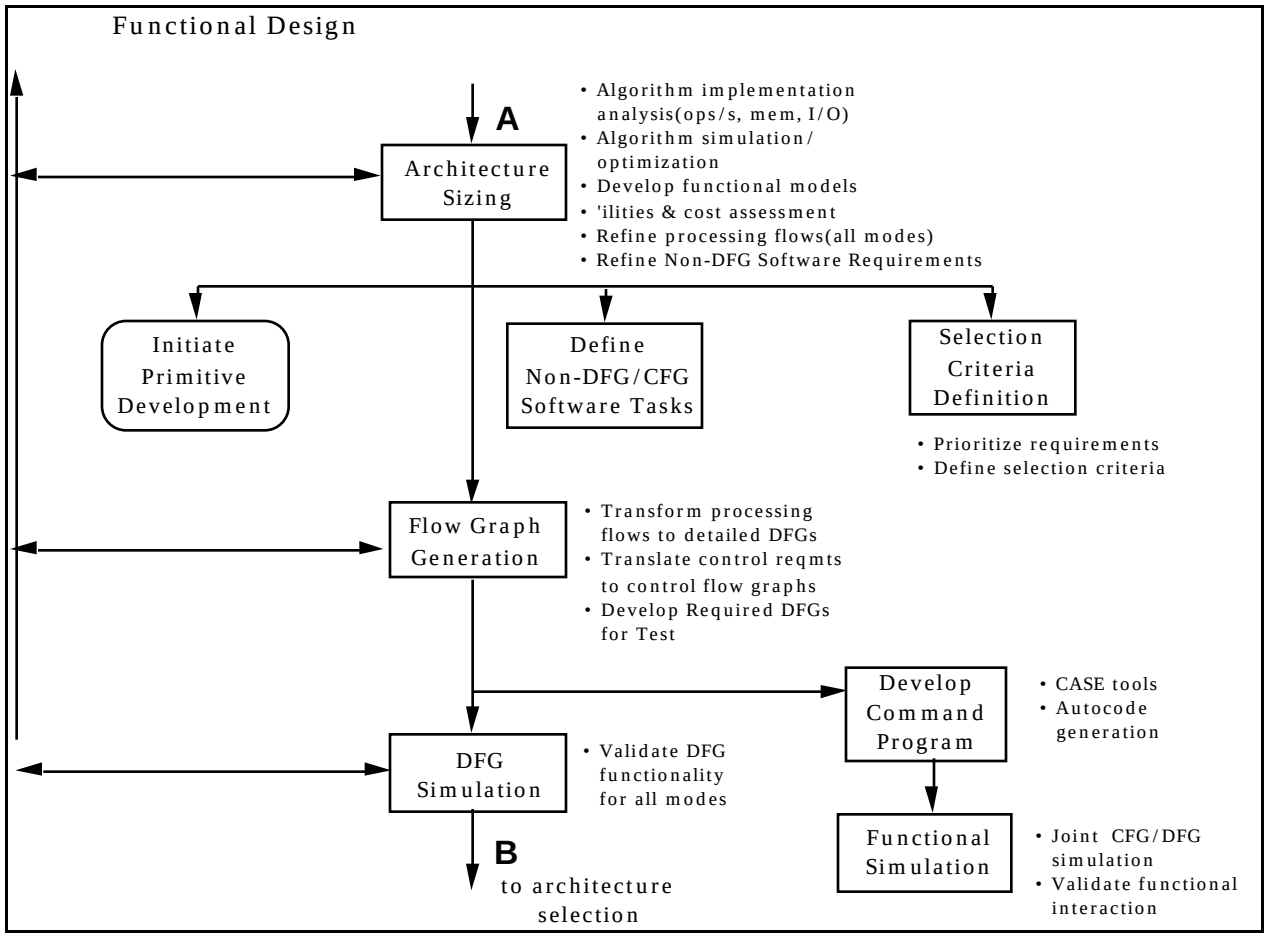


Figure 3-8. Functional design process.

Inputs to this portion of the process are the outputs of the system design activity. These include both signal processing requirements and physical requirements. The signal processing requirements include the algorithm flows for all modes of operation, the control flows (if any) used to control the initiation of processing modes and the transition between modes, and performance timelines which must be met. The physical and/or programmatic requirements include the size, weight, power, reliability, testability, cost, and schedule.

Outputs from this portion of the process include prioritization of the implementation requirements, the library-based DFGs representing the processing for each operational mode, software requirements for non-DFG processing (e.g., mode switching), and definition of the criteria and weighting used to select the architecture. Note that the selection criteria may be just as sensitive to cost and schedule as it is to performance.

Architecture Sizing

We analyze the system requirements and processing flows for all required modes in terms of estimated operations per second, memory requirements, and I/O bandwidths processing requirements. We can make initial size, weight, power, and cost assessments based upon rules of thumb and experience sizing estimates. We also develop a first-pass partitioning of hardware and software functionality at this point. We develop requirements for non-DFG/CFG software. Functional models may be developed where necessary and algorithm simulations and optimization performed to refine the processing flows and derive the detailed requirements for the signal processing. The resulting functional processing flows represent the detailed algorithms that must be performed for each required mode.

Selection Criteria Definition

We prioritize the overall system requirements and the derived requirements and establish a selection criteria. The selection criteria provides the necessary basis for subsequent architecture trade-off analysis. An example of typical parameters used in processing trade-offs studies is shown in Table 3-3.

Table 3-3. Typical processor trade-off criteria.

Architecture/Hardware	**Software**
Performance	Compilers (availability/efficiency)
Peak	Operating system/support SW
Sustained	functionality
Input/Output Bandwidth	performance
Intraboard	availability/maturity
Board-to-board	Programming tools/environment
Chassis-to-chassis	
I/O pin requirements	**Costs**
Memory (size/performance)	Non-recurring (design, DFT,
Cache	development, capital)
Local	Production
Global	Test Cost
Power dissipation	Life cycle
Static	
Dynamic	**Risk**
Testability	Technical
Fault Coverage	Schedule
Test Time	Cost
BIST Support	
Mechanical packaging complexity	**Quality**
Thermal packaging issues	Defect level
Scalability/Upgradeability	Reliability

We define a trade-off matrix to formalize the selection criteria for the architecture, as shown in Figure 3-9, which contains the top-level requirements allocated to the signal processor. Satisfying these requirements drives the hardware/software codesign of an architecture. We populate the matrix and iteratively update it as any given design progresses. Early in the process, the entries are less accurate than later on. The goal is to eliminate some designs early while carrying the best candidates to subsequent levels of detail. The entire Integrated Product Development Team (IPDT) has rapid access to the ongoing trade studies and participates in populating the trade matrix through various tools and design advisors. Once we have established the criteria, we develop its relative weighting for the particular application to ensure that the proper emphasis between performance, cost, schedule, and risk is reflected in the architecture selection. We use these weighted criteria to drive both the architecture selection and

the level of optimization and effort that is applied during this portion of the design. The exact content of the trade-off matrix and the maximum scores associated with different attributes is project dependent.

Architecture Tradeoff Matrices									
Architecture Attributes									
	Size	**Weight**	**Power**	**Schedule**	**Testability**	**Cost**	**Reliability**	**• •**	**Total Score**
Arch # 1									
Arch # 2									
•									
•									
Max Score	0-5	0-25	0-15	0-5	0-15	0-10	0-15	0-10	0-100
Architecture Scores									
	Size	**Weight**	**Power**	**Schedule**	**Testability**	**Cost**	**Reliability**	**• •**	**Total Score**
Arch # 1									
Arch # 2									
•									
•									
Max Score	0-5	0-25	0-15	0-5	0-15	0-10	0-15	0-10	0-100

Figure 3-9. Architecture selection criteria.

Define Non-DFG/CFG Software Tasks

We review the non-DFG/CFG software requirements and define the tasks required to fulfill these requirements.

Flow Graph Generation

We transform the finalized algorithm processing flows into the detailed DFGs as the first step in hardware/software codesign. In parallel, we optionally develop any DFGs which are specifically for test. These DFGs are based upon the Processing Graph Method (PGM) developed by the Navy. PGM is a specification for defining detailed DFGs for signal processing applications. The DFGs are made up of the reusable library elements, which may represent either hardware or software. The resulting DFGs represent the composite algorithmic requirements for all processing modes. The algorithms to date have been mostly functional in nature, without regard to specific implementation issues such as processing efficiency. For example, the DFG may include a filter. Resultant efficiency will be related to whether a time-domain or frequency-domain implementation is selected, which may in turn depend upon the processor under consideration. Note that if several architectures are considered, unique sets of architecture-specific implementations may be chosen.

The DFGs are the basis for both the architecture synthesis, the detailed software generation, and potentially custom processor synthesis. As indicated in Figure 3-10, validating the DFG is an important step that ensures consistency with the simulatable requirements passed down from the systems process. The left side of the figure represents a processing flow that has been simulated during systems definition to establish the baseline algorithm set for the application. If multiple processing modes are required, there would exist a processing flow for each mode. The right side of the figure represents the detailed DFG constructed from reuse library elements.

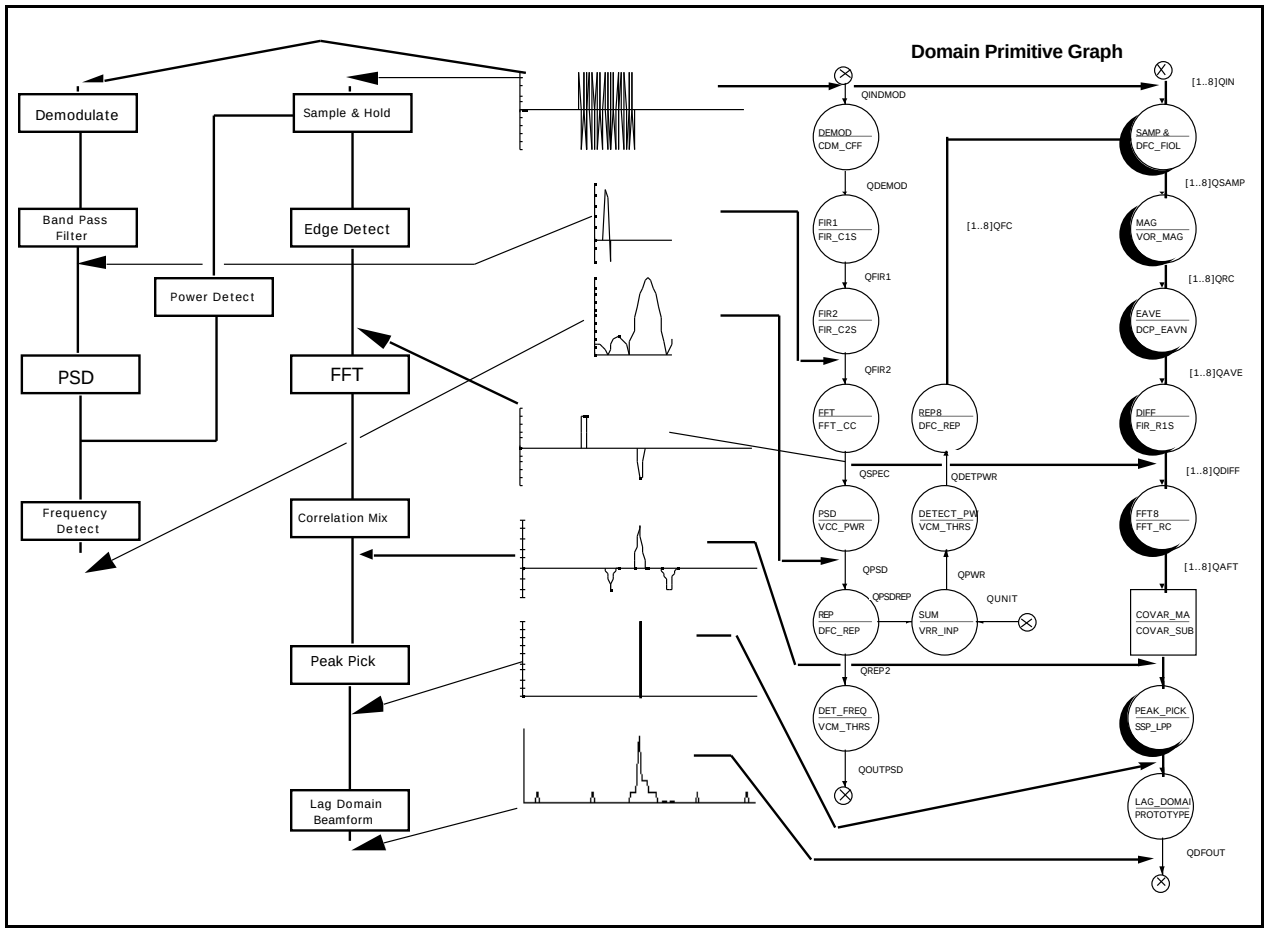


Figure 3-10. Correspondence of processing flows and domain-primitive graph.

Each detailed DFG is simulated to provide data for comparison with the algorithmic flows developed during the systems process. As part of this simulation process, the DFG may require modification until the correct functional results are achieved. Validating these two representations ensures that the simulatable functional requirements are captured in the DFGs that will drive subsequent codesign.

In some cases a suitable library component may not exist, which means designing either a hardware or software element. For software, this is referred to as a prototype element. The prototype element will permit the hardware/software codesign to proceed before validating the element for permanent inclusion in the reuse library. For hardware, a similar prototype is established by generating a high-level performance model of the desired component. In either case, the architecture process can proceed using estimates for these elements.

In addition to the DFG generation, the control flow requirements are transformed into the control flow graphs (CFGs) required to manipulate the DFGs according to a defined set of rules. This DFG control is referred to as command processing. Command processing is needed to incorporate signal processing into the overall system design. The signal processing functionality needs to be controlled in accordance with the other functionality dictated by the system design. The mechanism that exercises this control over the signal processing subsystem is the command program. The signal processing subsystem receives messages from the external environment and commands the signal processors to perform their detailed function. The command program performs the control operations of the signal processor subsystem. While DFGs describe the functionality associated with the signal processing subsystem, the command processing function is best described by an object-oriented methodology. The design of the two are interrelated, since design of the DFGs must provide access to or the ability to set graph parameters as dictated by the command program. Consequently these designs must be performed concurrently.

When we develop the signal processor code using the PGM, a set of operators are needed to control the resultant functionality. The signal processing graphs and their data structures are objects manipulated by the command processing program. The operations upon these objects have to be executed in both realtime and non-realtime. For example, we can initialize the primitive node and the resultant data structures in non-realtime. However, operations such as starting the I/O operation, disabling queues, mode changing, changing dynamic parameters, etc. are all realtime operations and are often specified by commands external to the DFG operation. The command program facilitates interface of the signal processor subsystem to the rest of the overall system. As part of the functional design, the detailed specification of these non-DFG processing requirements is completed. Conceptually, the command program manipulates objects. The objects are the DFGs and their data structures. The top-level requirements define a set of actions that have to be performed by these objects. The designer must specify the actions associated with each of the states of these objects, both for the normal and abnormal conditions. This information will be contained in a state model. Associated with all the states is a process model that expresses the procedures to be executed at each state.

Command Program Development

We transform the state and process models via autocode generation into prototype code that will be used with the DFGs to simulate the interactions among graphs, particularly mode transitions. The command program must be able to accept messages from outside the signal processor, interpret those messages, and generate the appropriate control information to stop graphs, start graphs, initiate I/O, set graph parameters, etc. We can develop the command program either
through standard software development CASE tools or through the tools that provide autocode generation capability from state transition diagram descriptions.

Functional Simulation

As part of the functional design process, we must simulate both the DFGs and the CFGs. We simulate the DFGs to validate that they represent the same processing as the mathematical description of the processing flows developed during the systems process. We also have to validate various aspects of the CFGs, which must interact with the DFGs. This includes passing parameter information between the external world and the graph management software, initiating or terminating I/O devices, starting and stopping DFGs, etc. Individual DFGs have been simulated previously and their functionality is not in question. However, it is desirable to simulate the interaction of the various DFGs, as dictated by the CFGs, to ensure that all interactions are functionally correct. In particular, it is necessary to know that mode transitions occur properly.

3.2.1.2 Architecture Selection

Architecture selection is an automation-aided process to rapidly evaluate different architectural designs and instantiations of these designs. For example, an existing architecture may be simulated with new DSPs to quickly evaluate inserting emerging COTS technology. This is the Model Year upgrade concept. An integrated toolset will facilitate rapid performance trade-offs to select and size a scalable architecture based upon the processing requirements. These trade-offs are tightly coupled with the performance of the software on the architecture. Automated multiprocessor partitioning and mapping, coupled with user intervention, provides a rapid optimization capability. The trade-off process supports the IPDT concept by integrating tools for DFT, cost analysis, and high-level design advisors. In addition, the architecture process is coupled through VHDL to the detailed synthesis of chips, boards or processors.

The architecture selection process, shown in Figure 3-11, represents the heart of the RASSP hardware/software codesign, which uses a library-based, DFG-driven approach to software development combined with iterative performance trade-off analysis to support rapid selection/analysis of candidate architectures. During architecture selection, various architectures are offered as candidates that are selectively optimized. For each candidate, the process includes the following steps:

- Developing a partitioning and mapping for the candidate architecture
- Performance analysis of the partitioning and mapping
- Optimization of mapping, resulting in processor instantiation
- Analysis of instantiation size, weight, power, cost, testability, reliability, risk, etc.
- Iteration of the above until one or more acceptable architectures are attained.

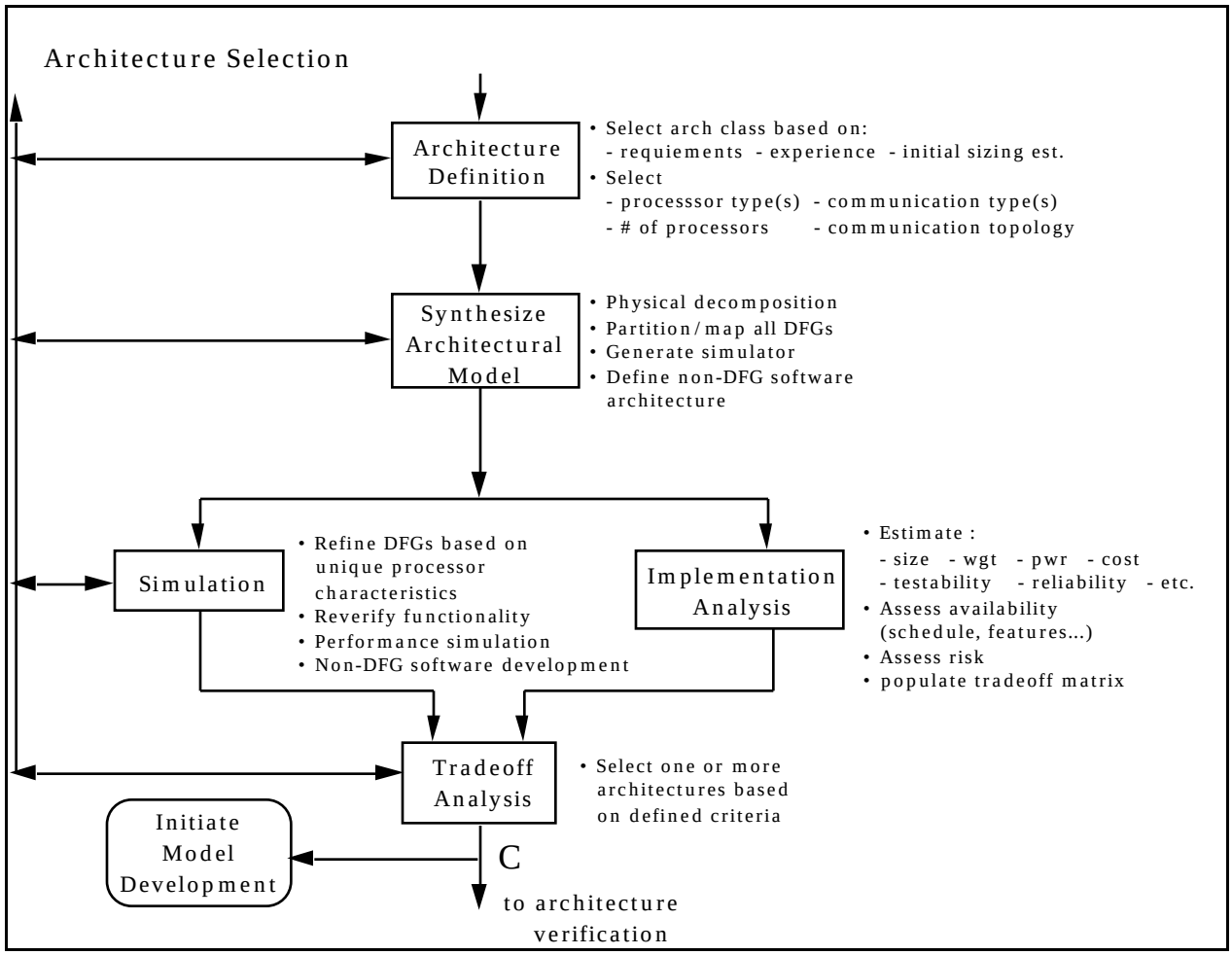


Figure 3-11. RASSP architecture selection process.

We use a trade-off analysis based on the established criteria to select the detailed candidate architecture and software. Architecture selection is iterative and intended to produce one or more architectures that meet the overall requirements. Ideally, we evaluate candidate architectures that span the design space.

Inputs to the architecture selection process are the prioritized processing requirements, the selection criteria, the required DFGs for all modes of operation, command program specification and other non-DFG requirements, and the hardware/software reuse library.

Outputs from the architecture selection process are the finalized DFGs and one or more architecture instantiations that we selected for more detailed functional and performance verification. Also output from the architecture selection process is the description of the DFG partitioning and mapping to the processors of the selected architecture(s) for all processing modes.

The steps involved in architecture selection shown in Figure 3-11 are discussed below.

Architecture Definition

The next step in the hardware/software codesign process is specifying an architecture. This includes both

selecting a class of architectures (e.g. MIMD, SIMD, etc.) and the design approach within the class (e.g. interconnect topology). This decision is generally based upon a combination of the signal processing architects' application domain experience and the system requirements, cost, availability, technology maturity, etc. The architecture should include any processor(s) necessary to satisfy the command program requirements.

Given the DFG (or set of DFGs) that describes the processing, the architect must postulate one or more designs that may satisfy the requirements. These architecture choices are based upon the domain experience of the design team. One of the goals of RASSP is to facilitate the ability to define and evaluate more alternatives than would otherwise be possible through semi-automated tools that assist the architect. It is important to note that the processing represented by the DFGs at this point has not been allocated to either hardware or software. However, as the architect defines an architecture for consideration, one or more nodes of the DFG may be allocated to custom hardware or to a postulated processor that currently does not exist. Figure 3-12 shows two simple alternative architectures. In Arch #2, one or more nodes of the DFG have been allocated to either an existing or postulated special-purpose ASIC. In either case, the architect made a decision regarding hardware/software allocation. In Arch #1, all processing is performed in software on one of the four processors, while in Arch #2 some functions are constrained to the ASIC, while all others are allocated to software on one of the two processors. These architectures illustrate the point. In reality, the RASSP methodology must be capable of dealing with architectures that range from simple cases (such as those illustrated) to multi-chassis configurations of hundreds of processors. Architectures can be constructed from existing entities ranging from single processors to boards.

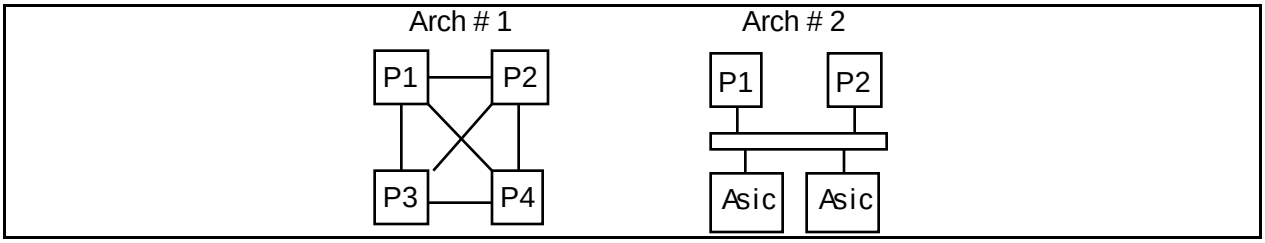


Figure 3-12. Example candidate architectures.

For the selected architecture type, we select specific processor type(s) (e.g., TMS320C40, i860, AD21060, etc.) and number of processors, along with a desired communication mechanism (e.g., bus, Xbar switch, etc.). We choose the number of processors based upon prior estimates of performance and/or validated benchmark data for library code fragments, and knowledge of the system requirements. We can perform a preliminary physical decomposition and do a high level study of routability, thermal issues, etc.

Architecture Model Synthesis

The process of defining the architecture is coupled with the allocation of the DFG to the architectural elements. For example, specific nodes in the DFG may be assigned to a custom hardware element such as an FFT chip. To support this capability, the library would contain a hardware model capable of performing the processing defined by the DFG node.

The portion of the DFG(s) allocated to software is partitioned and mapped to the available processors of the candidate architecture under consideration. The software partitions are defined by mapping the primitives in the DFG to the DSPs in the architecture. Figure 3-13 shows a DFG in which two portions of the DFG are allocated to hardware and the remainder of the DFG
allocated to software grouped into four partitions. This activity is supported by multiple, automated partitioning/mapping algorithms for graph assignment and a manual capability. This process may utilize 'what -if' experimentation to evaluate postulated architecture components, as well as software performance estimates for prototype primitives. In addition we postulate architectures for the non-DFG software. We construct a VHDL performance model for the architecture. It may be obvious that special-purpose hardware or a custom processor is required to meet the overall signal processor requirements. If so, we can start a mini-spiral development activity to embark on the required development. The performance of the new hardware may be estimated so that the overall architecture selection may proceed. As models are developed for the new processor, library models may be updated with new timing information and the architecture reevaluated.

Simulation

Based upon unique characteristics of the individual processors, we can refine algorithms to optimize performance that is verified again via simulation. We also begin high-level language non-DFG software development and simulation. The goal of simulation at this stage is to both verify the algorithms again functionally (if modifications have occurred), and to refine the anticipated performance of the candidate architecture using available throughput, memory, and I/O estimates for these algorithms**.** We estimate the overall architecture performance via VHDL simulation and analyze it with respect to meeting all signal processing requirements. The performance simulation reflects the established partitioning and mapping of the DFGs. We simulate a particular partitioning and mapping to estimate the overall timeline for the processing. These performance simulations are executed iteratively as we consider different partitioning and mappings in an attempt to optimize the execution timeline. The architecture may be optimized through iteration of architecture synthesis and simulation. After simulation, we can modify the architecture by changing or adding processors, changing communication types, changing interconnect topology or postulating new architectural elements ('what-if' analysis) that require subsequent development.

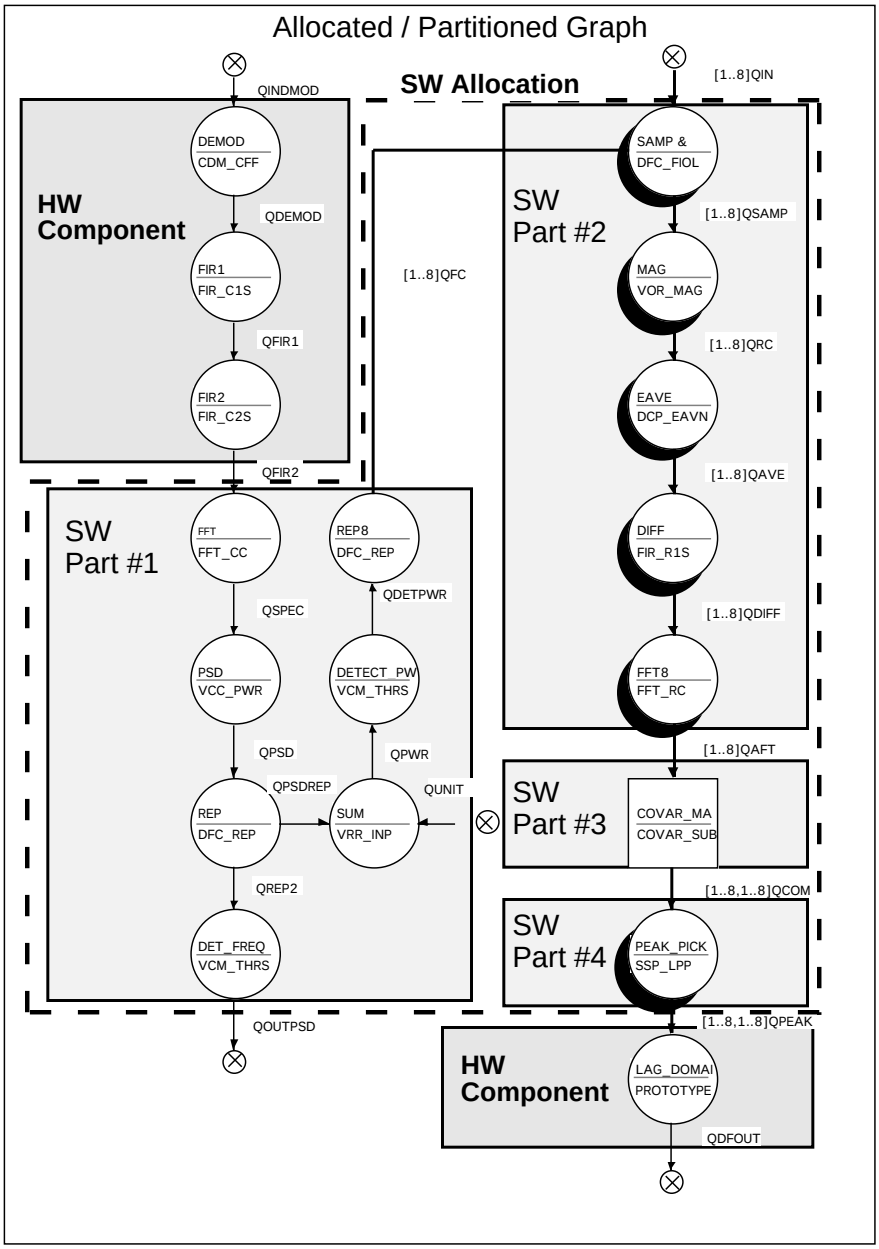


Figure 3-13. Graph showing hardware/software allocation and software partitioning.

Figure 3-14 shows example timelines for the two architectures in Figure 3-12 for the 6-node graph in the figure. The 6-node graph is a simplified representation in which n1 and n6 represent the graph partitions allocated to hardware in Figure 3-13, and n2, n3, n4, and n5 represent the software partitions. Timeline (a) represents the mapping of the six graph nodes to the four
processors shown in Arch #1 of Figure 3-12 and timeline (b) represents the mapping of the graph to Arch #2, which contains two-special purpose ASICs. The example illustrates a much improved overall timeline for the architecture when n1 and n6 are mapped to custom ASICs, all other processing being the same. The importance of the example is to illustrate that different architectures and graph mappings to those architectures can be evaluated quickly, which is especially valuable in large systems.

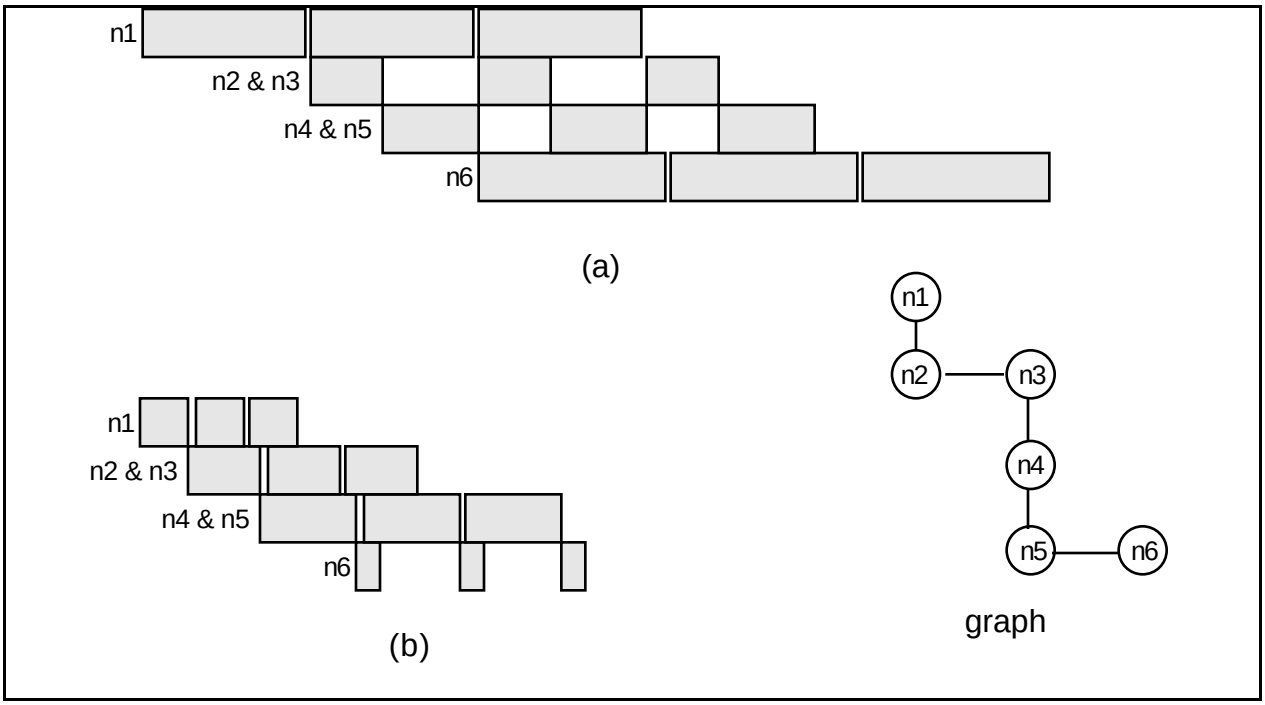


Figure 3-14. Timeline for two postulated architectures.

Implementation Analysis

Concurrent with the simulation effort to establish performance, we can proceed with implementation analysis of the architecture, if desired (depending on the degree of satisfaction with the architecture). Any given architecture, or multiple candidate architectures, may be in the process of analysis by various members of the PDT. For each candidate architecture, we postulate an implementation (using high-level synthesis tools and/or design advisors driven from a VHDL description of the architecture) and transform it into size, weight, power, throughput, and cost parameters, as well as schedule, testability, reliability, availability, and maintainability estimates. This process requires the collaboration of the IPDT to properly assess the attributes of the architecture. In addition, we assess component availability with respect to supporting the desired development schedule. The concurrent use of tools and design advisors speeds the process and leads to better-informed, early decisions. Based upon this analysis, we can assess implementation risk for the different architectures.

Trade-off Analysis

We iterate the architecture synthesis, simulation and detailed analysis process for each candidate architecture to obtain an optimized solution. These activities are directed toward populating an architecture trade-off matrix that is a record of the design process. The trade-off matrix is supported by design notes that document the rationale for the entries in the matrix. We analyze the information gathered from the architecture synthesis process with respect to the selection criteria and weighting established in the functional design process. Based on the selection criteria, we select one or more of the candidate architectures for further evaluation.

At this point of the design, it may become obvious to the design team that a custom processor or special-purpose ASIC is required to meet the program requirements. If custom hardware is required for a viable solution to the requirements, we can define and support ‘what-if' elements in the library and assign processing times associated with them for the DFG nodes. We can evaluate the 'new' element; if satisfactory performance is achieved, we can start the appropriate hardware model development in parallel with the architecture selection process. Or, if desired, we can invoke a processor synthesis tool that is driven from the DFG primitives. The synthesis tool outputs VHDL compatible with down stream detailed design tools.

In this iterative process, we can modify the partitioning, mapping, and the architecture as part of the optimization process. Tools support interactive design changes to both the architecture and/or the partitioning and mapping. Of particular interest is the ability for designers to quickly modify the processor interconnect topology for large architectures. The candidate architectures are each evaluated with respect to the established selection criteria, and then we may select one or more architectures for further consideration.

3.2.1.3 Architecture Verification

Architecture verification is the process of hierarchically simulating both the functionality and performance of a selected architecture candidate. Here, simulations are performed at a greater degree of detail than during architecture selection. An integrated framework supports mixed-domain simulation so that high-level performance and functional simulation can be coupled with ISA or RTL VHDL simulators, hardware emulators or hardware testbeds. The goal of the verification process is to validate operation of all architectural entities and the interfaces between them before detailed design. The specific verifications required depend upon whether the design is all COTS, mixed COTS and new hardware, or all custom hardware. Software partitions are autocoded to produce software modules translated from the processor-independent library elements to optimized processor specific implementations, which are interfaced through a set of standard services built on an operating system microkernel.

Inputs to the architecture verification process include the selected architecture instantiation, which includes all or a portion of the implementation partitioning/component list, the optimized DFGs, the CFGs, detailed software description, and the hardware/software reuse library. Note that overall functionality has not been verified before this point. The role of architecture verification is to verify functionality and more detailed performance of the candidate implementation using a combination of testbed hardware, simulator(s), and/or emulator(s) before detailed hardware implementation. This is the first real step in hardware/software codesign verification and is important to verify the virtual prototype defined to date. This process can iterate with the architecture selection process to 1) optimize the selected architecture and 2) update model performance in the RASSP library. As shown in Figure 3-15, the major steps in this process are autocode generation, performance simulation, reassessment of architecture attributes, new element development, component mix evaluation, definition of a verification approach, simulation development, and performance and functionality verification.

Outputs from architecture verification include new library elements, detailed specifications for hardware development, and performance and functionality verification.

Autocode Generation

The architecture(s) produced in the architecture selection process, the finalized DFGs, and the partitioning/mapping data, provide the necessary input to autocode generation. We use the architecture description and the partitioning and mapping data to automatically generate the software for each of the partitions. We generate the code by translating the processor-independent flow graph primitives to target-specific code which use the optimized math libraries for the specific DSP. The result of this process is definition of a new DFG node (called an equivalent node), which represents the entire graph partition. As part of this procedure, we simulate the resultant code to ensure that the resulting node satisfies the same test bench as the original subgraph. The resulting equivalent nodes from all the graph partitions can be combined into a new equivalent graph that has the same characteristics as the

original, but each node represents a software module representing the aggregate processing of a subgraph. We can verify the functionality of this equivalent graph again via functional simulation to maximize confidence in the overall translation process.

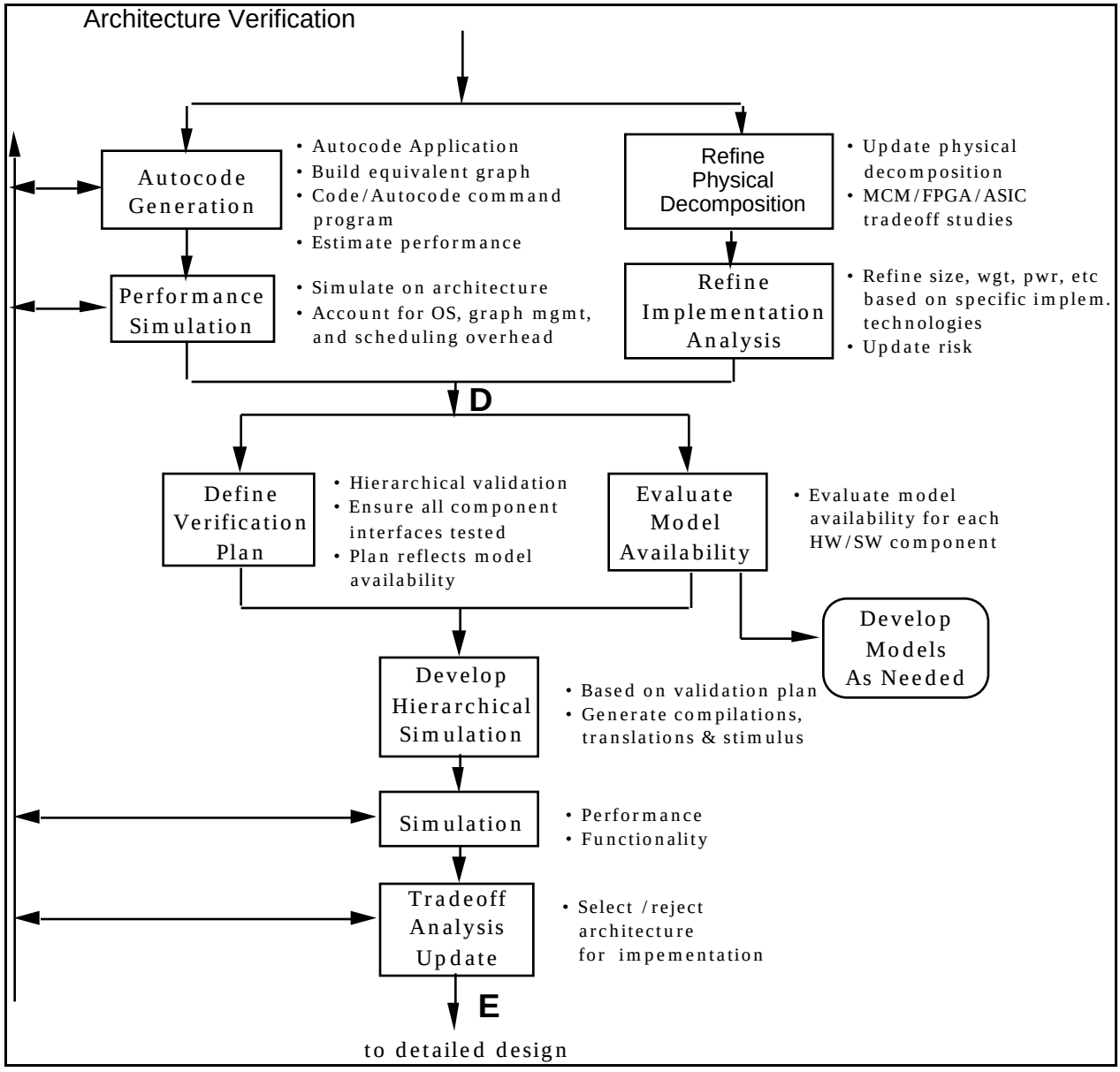


Figure 3-15. RASSP architecture verification process.

Performance Simulation

We generate timing estimates for the autocode-generated software, and performance simulation can be repeated using the new timing estimates. Ideally this simulation is at a greater level of detail than previous simulations. It should account for performance impacts due to the target operating system, the graph management system built on top of the operating system, and any scheduling overhead necessary. The equivalent DFG(s) performance is simulated on the candidate architecture, much like was done in architecture selection, but with a higher degree of fidelity.

Figure 3-16 shows the 6-node graph replaced by a 4-node equivalent graph in which the nodes are aggregated as dictated by the partitioning and mapping. The performance simulation at this level accounts

for communication protocol so that detailed network contention is evaluated. If necessary, we can modify the partitioning, the architecture, and/or the DFG until performance is satisfactory. Modifications, of course, require iterating through the process steps again.

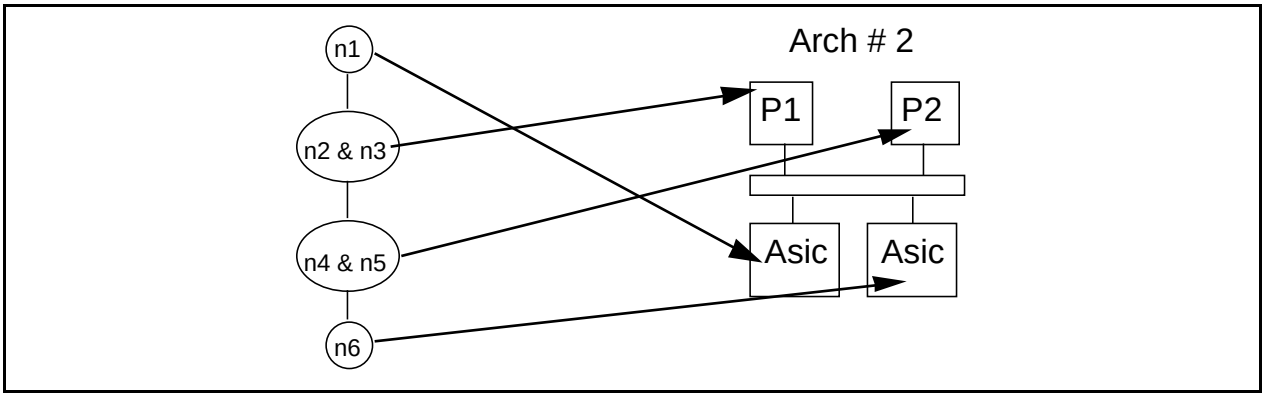


Figure 3-16. Mapping of equivalent graph to Arch #2.

Refine Physical Decomposition

We continue to refine the physical decomposition and implementation of the design. We perform MCM/FPGA/ASIC trade-off studies. We update the routability, thermal, and other analyses based on more detailed information.

Refine Implementation Analyses

During architecture selection, we produced the initial estimates for size, weight, power, cost, schedule, testability, maintainability, etc.. We used this information to select one or more candidate architectures for further evaluation. After initial selection of candidates, we can perform a more detailed evaluation of the architecture attributes. This may include reiteration of size, weight, power, and cost estimates based upon particular implementation technologies, such as MCMs, surface-mount technology, etc. This activity involves various disciplines from the development team. The architecture description of viable candidates may again be processed using hardware synthesis tools and various design advisors to produce the updated architecture attributes. We can assess increased costs due to architecture modifications necessary to meet reliability requirements, since these could impact final architecture selection.

NOTE: It can be argued that these three steps (autocode generation, performance simulation, and refine implementation analysis) could be allocated either to the architecture selection or architecture verification process.

Model Availability

Architecture verification is the process of hierarchically simulating both the functionality and performance of a selected architecture. In general, we perform the simulations at more detailed levels than previously done. In preparation for architecture verification, we must determine the level of hardware and/or models available for all architecture components. This becomes the basis for defining a functional verification approach. For each architecture component, the hardware may exist and be available, or various levels of models may exist in the library, including behavioral models, ISA models/simulators, RTL models, and/or performance models. In the ideal case, all software components in the reuse library have performance data for each processor supported in the library. In reality, this performance data is most likely generated on an as-needed basis. Consequently, for each software component not supported by performance data for a required processor, we must evaluate the availability of hardware and/or various level models to support validation of that software component. The availability of these hardware components and/or models provides constraints on the verification plan.

Verification Approach Definition

An architecture verification plan should be developed that ensures, to the maximum extent possible, that all hardware components will function and interoperate as expected and all software will execute properly on the architecture when built. This will likely require a hierarchical approach to the architecture verification. Virtual prototyping the entire architecture at the RTL-level requires an exorbitant amount of time. The plan must, however, ensure that all component interfaces are tested, all devices properly communicate with each other, and all software executes on the appropriate processor.

An integrated framework supports mixed-domain simulation so that high-level performance and functional simulation can be coupled with ISA or RTL simulators, hardware emulators or hardware testbeds. The goal of the verification process is to validate operation of all architectural entities and the interfaces between them before detailed design. The specific verifications required depend upon whether the design is all COTS, mixed COTS and new hardware, or all custom hardware.

Simulation Development

The objective of this process is to enable incremental functional and performance evaluation of hardware and simulation models throughout the design process. We do this by providing an integrated suite of tools that support a combination of testbed hardware, simulator(s), and/or emulator(s) to fully verify performance and code functionality before hardware implementation. The ideal approach is to integrate these disparate simulators through a multi-domain backplane that enables mixed-mode simulation. The multi-domain simulation capability should support the following elements:

- Testbed Hardware — The most critical element in enabling algorithms to be tested on target multiprocessor hardware is a robust distributed application environment. The environment should support user porting and analysis of algorithms on a single workstation and embedded multiprocessors. It will provide users with an application environment that supports multiprocessor mapping, instrumentation, and performance monitoring.

- Behavioral Simulation — Behavioral simulation will verify functionality and performance of new designs. The simulations will be VHDL-based, and will support a number of commercial

simulators. The results of behavioral simulation will be used to used to update network simulation model performance.

- Architecture (Network Performance) simulation, which is defined here to be non-functional data-flow-level simulation, determines the performance of large systems. RASSP efforts are focusing on providing interoperable VHDL performance models. Interaction of this capability with behavioral simulation and hardware testbed capabilities provides a method to verify system-to-subsystem performance. The performance results can be used to update the architecture selection processor performance models.

Constructing the hierarchical simulation is based upon the availability of models supporting the entities in the architecture. In some cases, there may be available hardware testbeds for COTS processors, in which case using them is the most effective approach. In general, there may be models available at different levels and in different languages. While it is a goal on RASSP to use VHDL as a common language, we are taking a pragmatic approach to enable use of all available modeling technologies. If sufficient models have not been developed during the design process, we must create them at this point. Based upon model availability, we map the architecture to appropriate simulation engines. Illustrated in Figure 3-17 is one of the simulation mappings that

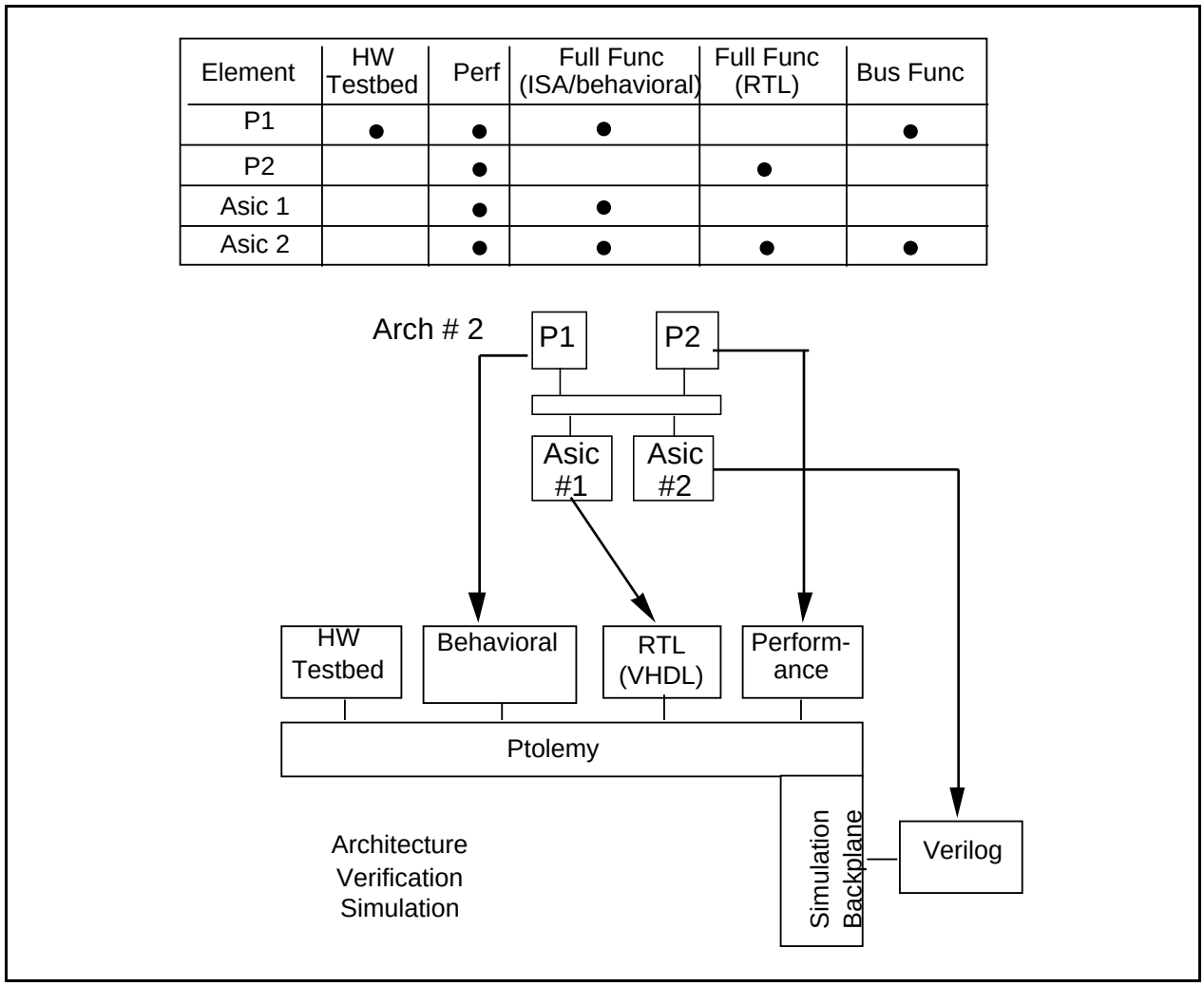


Figure 3-17. Mapping of architecture to appropriate simulation engine.

could be run; the process envisions a set of hierarchical mappings to efficiently verify the entire hardware/software suite. The interoperability and integration of various commercial tools operating in different domains provides a mechanism to perform efficient simulations for verification. The ideal is to support the interoperability of commercial tools to simulate a complete system in a seamless fashion.

In the event that a hardware testbed is available, detailed evaluation of the performance can be readily obtained through tools that provide detailed timelines of both processing and communication on the architecture. The ability to perform point-and-click mapping of functions to processors provides a way to fine-tune the application, as required. This approach is most useful when performing Model Year upgrades to an existing signal processing system.

Simulation

Executing the constructed, multi-domain simulation provides performance and functionality verification. The detailed simulation provides:

- Concurrent verification of detailed software on hardware and simulation models
- Hierarchical performance verification
- Detailed behavioral specifications for new design elements
- Performance/functional updates to RASSP library models.

Trade-off Analysis Update

The iterative process of simulating the various architectures and the graph mappings to them, along with the PDTs' estimation of the architecture attributes, using interactive tools where available, results in the completion of the trade-off matrix shown in Figure 3-18. The results of the various trade-off studies are reflected in population of the trade-off matrix, which is a record of the design process performed. The entries in the table should be supported by detailed notes to provide both designers and program management insight into the rationale behind the selections.

Architecture Trade-off Matrices									
Architecture Attributes									
	Size	**Weight**	**Power**	**Schedule**	**Testability**	**Cost**	**Reliability**	**• •**	**Total Score**
Arch # 1	500 in^2	1.0 lbs	20 W	8 mo	high	low	high		
Arch # 2	200 in^2	0.5 lbs	10 W	18 mo	med	med	high		
•									
•									
Max Score	0-15	0-25	0-15	0-5	0-15	0-10	0-15	0	0-100
Architecture Scores									
	Size	**Weight**	**Power**	**Schedule**	**Testability**	**Cost**	**Reliability**	**• •**	**Total Score**
Arch # 1	5	10	10	4	15	8	15		67
Arch # 2	12	20	15	2	10	5	15		79
•									
•									
Max Score	0-15	0-25	0-15	0-5	0-15	0-10	0-15	0	0-100

Figure 3-18. Example trade-off matrix.

We finish this iterative selection process when one or more candidate architectures satisfies the overall requirements. The IPDT reviews the choices and again selects one or more of the candidates for detailed design.

3.2.1.4 Software in Architecture Definition Process

This section provides a discussion of the software related activities in the architecture process. It is intended to provide a consolidated description of how requirements are translated to software in a DFG driven process.

During the system definition process, we develop the initial system requirements. These requirements typically include all external interfaces to the signal processor; throughput and latency constraints; processing flows by operating mode; all mode transitions; and size, weight, power, cost, schedule, etc. We define five items directly related to the subsequent software development:

- Algorithm Flows — Algorithm flows are block diagrams showing the high-level computations associated with an application. They are typically executable descriptions of the processing that must be performed (executable functional specification).

- I/O Requirements — The I/O requirements identify the amount of throughput the application has to handle, the modes of the data, and the source(s) of the data. The I/O requirements and the algorithm flows are the basis for building the architecture-independent domain-primitive graph.

- External Commands — These are the formal definitions of all commands allowable to the operator and/or command program.

- State Transition Diagrams — These diagrams show the major modes of system operation, and the state changes between them. They, along with the external commands, are the basis for the command program specifications. The external commands and the state transition diagrams are the basis for the command program specifications.

- Command Program Specification — These specifications are derived from the state transition diagrams and the external command definitions defined in the previous phase. They are required to build the domain-primitive graph and are iteratively refined in the production of the allocated graph. This refinement may occur in the architecture selection and validation phase.

This architecture definition phase of development involves the creating and refining of the DFGs that drive both the architecture design and the software generation for the signal processor. The input to this phase is the executable functional specification for a particular application and the command program specifications. We then partition and allocate the graph(s) to either hardware or software. During the architecture definition process, the DFG(s) of the signal processing is developed and the elements allocated to either hardware or software. We simulate the flow graph(s) from a functional standpoint and a performance standpoint until an acceptable hardware/software partitioning is achieved that meets the signal processing system requirements. This process contains many intermediate steps and graph objects defined in the following paragraphs.

Domain Primitive

Domain primitives are library elements that describe an element of processing to be done. The primitives are HOL programs that perform the graph interface and processing functions of a graph node. Graphs programmed at the domain-primitive level can be automatically translated into graphs executable on any supported target.

Domain-Primitive Graph

The architecture-independent DFG (domain-primitive graph) is a PGM graph refined to contain only domain-level primitives and subgraphs. The domain-primitive graph created during function design may be refined during the architecture selection process. This graph becomes the functional baseline graph for the application once it has been validated with the test vectors developed for the algorithm flows. The relationship between the algorithm flows and the target-independent DFGs (which are derived in part from the algorithm flows) is shown in Figure 3-19.

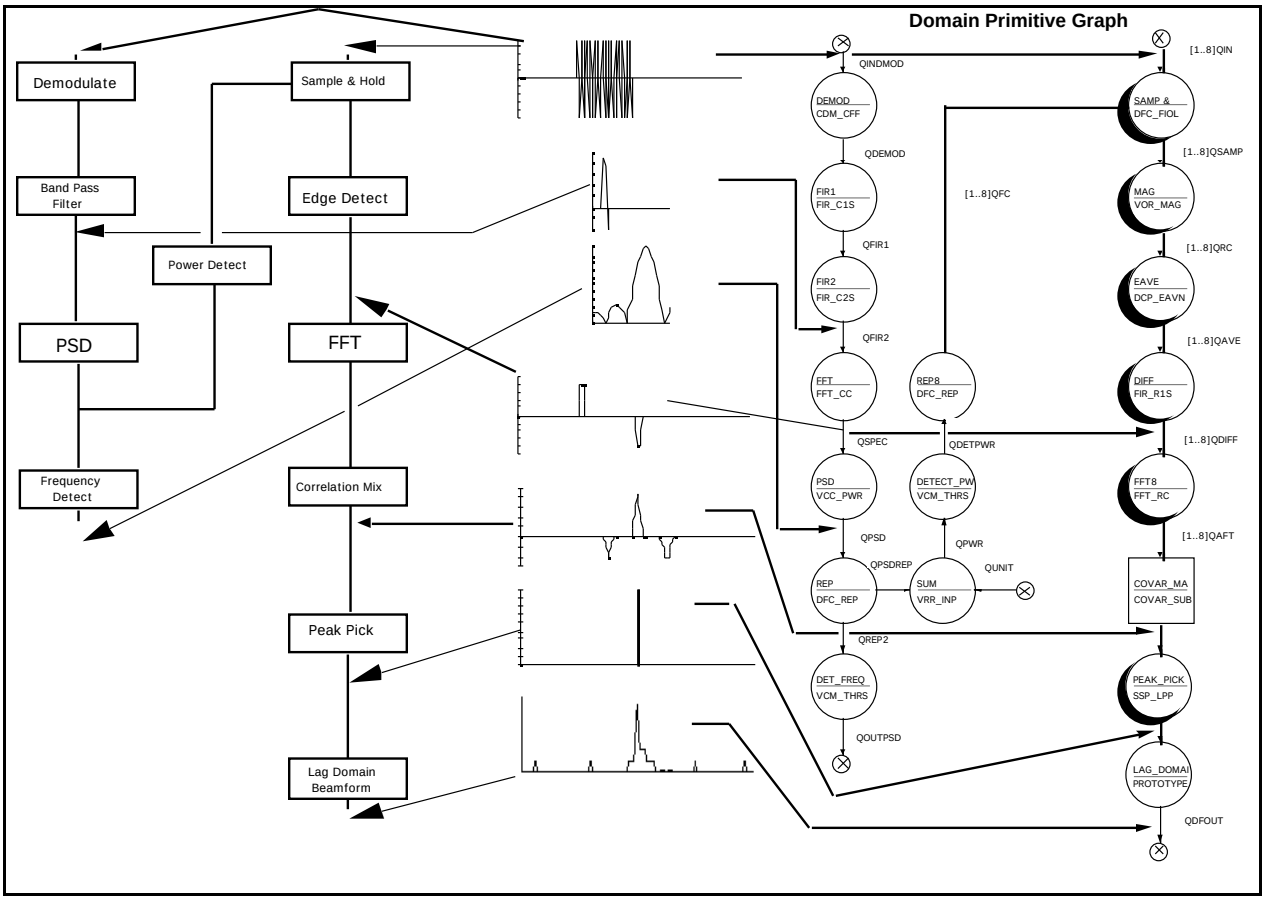


Figure 3-19. Algorithm flow correspondence with domain-primitive graph.

This figure shows an algorithm flow (shown as a functional block diagram) for a hypothetical application being translated into an architecture-independent DFG (domain-primitive graph). The domain-primitive graph is validated against the algorithm flows produced during system definition. A part of the validation of the domain-primitive graph is the creation of suitable test vectors. The domain-primitive graph is validated once the test vectors produce the same results predicted
from the algorithm flows. These test vectors are later used for graph validation throughout the application development process.

Generating the domain-primitive graph may involve an intermediate step in which the PGM graph is composed of domain primitives, domain subgraphs, prototype primitives, subgraphs, or any entity that can specify the workings of an algorithm. This permits continuation of the architecture selection process concurrently with generation of the domain-primitive graph when new primitives are required.

Allocated Graph

We produce the allocated graph by assigning each domain-primitive node or subgraph of the domain-primitive graph to hardware or software. Each node or subgraph must either be allocated to a hardware component contained within the candidate architecture or be identified as a software component to be run upon a processor class contained within the candidate architecture.

Figure 3-20 shows a candidate architecture and the segments of the domain-primitive graph assigned to either hardware or software components, which will be executed on the candidate architecture.

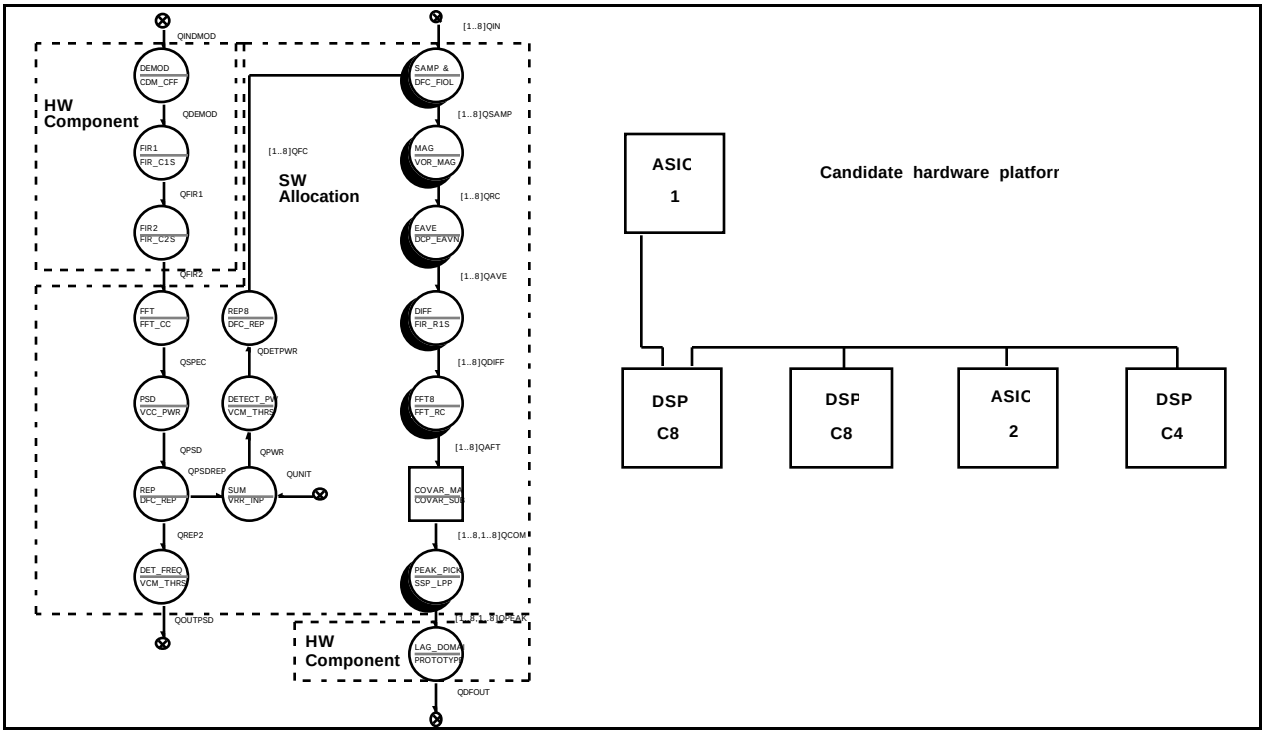


Figure 3-20. Graph allocation to hardware and software.

Partitioned Software Graph

The partitioned software graph is the software portion of the allocated graph. It has been partitioned into units of software that will be mapped to individual processors in the architecture. It represents all domain-primitive nodes and subgraphs allocated to software partitions. It is important in its own right because it may be partitioned separately from the full graph to optimize software partition performance.

Figure 3-21 shows the creation of the software partitions from the allocated graph. The hardware partitions are omitted for simplicity and are shown as ASICs. In this example, ASICs are assumed to be the final form of the hardware partitions as they apply to the code downloadable to the target processor.

The domain-primitive graph must be re-partitioned for each candidate architecture. The criteria we use in determining the optimal partitioning scheme included the system throughput requirements, latency, memory, etc. We can iterate the partitioning and the architecture to achieve improved performance, cost, reliability etc.

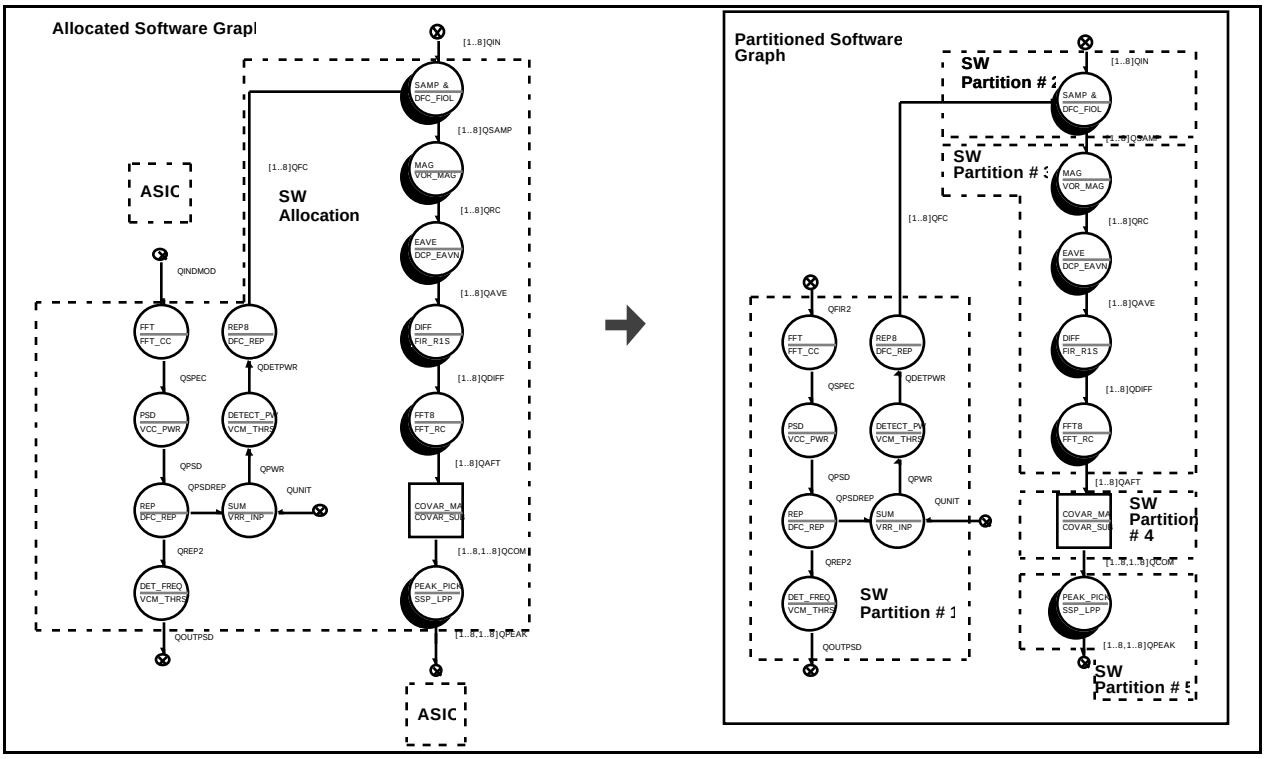


Figure 3-21. Software partitioning of the allocated graph.

Partition Graph

A partition graph is a stand-alone PGM graph that represents either a hardware or a software partition in the original domain-primitive graph. We build partition graphs in the autocode process, and use them to construct equivalent nodes for each partition. We also build simulation interfaces for each partition to simulate the application.

Figure 3-22 shows the derivation of an equivalent node for software partition #1. The software partition is transformed into a stand-alone partition graph that is represented by source code, a partition specification, and documentation. We use the partition specification and source code to create an equivalent node for the partition, which has the same characteristics as the subgraph represented by the partition. We use the equivalent node to build an equivalent graph in which each node represents one of the original graph's partitions, either hardware or software. The equivalent node is validated using the same test vectors used to test the graph partition. This process is equivalent to the traditional software unit test.

Equivalent Application Graph

We produce the equivalent application graph from the partitioned software graph by replacing each partition with its equivalent node. The equivalent graph is a PGM graph in which all partitions identified in the partitioned graph are replaced with equivalent nodes. The equivalent graph is executed using the same test vectors used during the domain-primitive graph

simulations, and it is considered functionally validated if the output vectors of the simulation match the output vectors obtained from the domain primitive graph simulations.

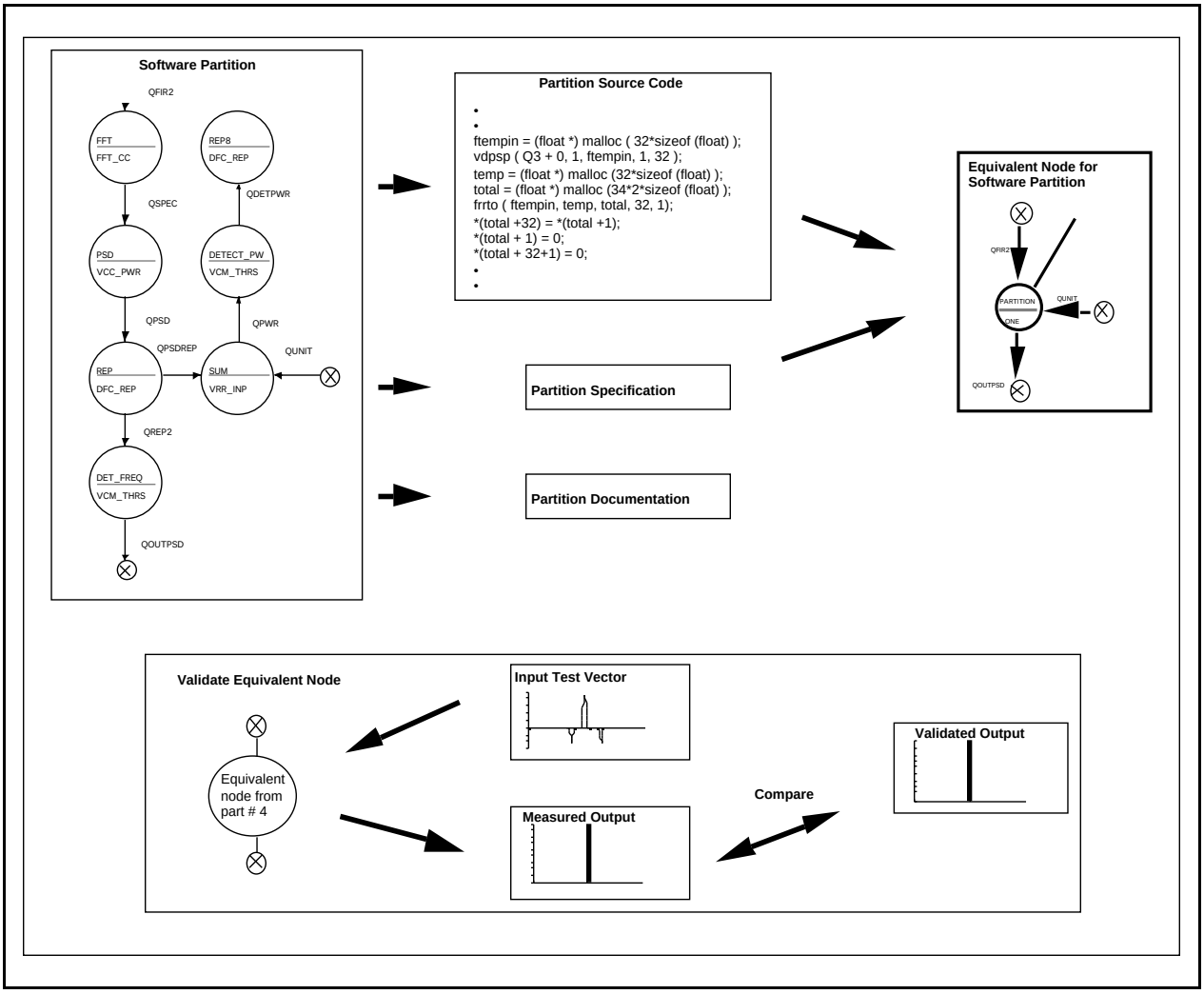


Figure 3-22. Creation of equivalent application node for software partition.

Command Program

In parallel with the graph allocation, partitioning, and performance simulation on a candidate architecture (which is aimed at concurrently optimizing the architecture and the software partitioning), we develop a command program that controls the behavior of the graphs. The command program can interpret the messages received from the external source (operator or higher-level command and control system) and execute the detailed graph commands that control operation of the signal processing graphs within the signal processor.

DFG/Command Program Functional Simulation

At any point along the development path, we can interface the command program with the DFGs representing the signal processing and these can be jointly simulated to ensure that the graphs are correctly managed within the overall context of the application. Of particular interest is the ability to properly execute mode transitions, which may require stopping, starting, and initializing different graphs; starting and stopping I/O; setting graph variables; or passing data and/or parameters to the external world. This joint simulation ensures that all modes operate properly and transitions between modes occur as required by the specifications.

Non-DFG Software

Non-DFG software requirements tasks and architectures are defined during the architecture process. We develop all code to the maximum extent possible.

3.2.2 Use of VHDL in Architecture Process

At this point in the process, we have specified DSP system network topology and partitioned the DSP system functionality onto architectural (the hardware/software) elements. We model functionality in terms of timing, resource usage, and algorithmic operations. VHDL is used to model subsystem timing and resource usage.

As we partition functionality between hardware and software, we model the hardware components in VHDL at the abstract behavioral-level; the software components are expressed in a pseudo-language, DFG, program design language (PDL), or ADA. We develop architectural performance (network simulation) models in VHDL. The software components exist in files that can be interpreted and executed by the VHDL models for co-simulation/co-verification of the candidate processor implementation.

The output of the architecture design process is a test bench and a model for each component.

- Test Bench — The test bench provides test procedures, stimuli, and expected responses to verify that the architecture meets system requirements. The test bench is developed before, and is executed with, the component models during co-simulation/co-verification.

- Architecture Model — The architecture model describes the system in terms of the structural interconnection of components whose functionality is described in terms of a) timing, b) resource usage, and c) algorithmic operations, as shown in Table 3-4.

Table 3-4. Elements within the architecture model.

Timing Data	Resource Usage Data	Algorithmic Data
• DSP subsystem component I/O - I/O timing constraints - I/O interface structures - I/O protocols - Signal levels - Message types • DSP subsystem component processing latency - Data acceptance rate • DSP subsystem component stimuli response	• CPU usage • Memory usage - RAM - ROM - Disk drives - Tape drives - Fast and slow cache - Local and shared memory - Network bandwidth contention	• Fast-Fourier Transform (FFT) • QR-decomposition • Finite Impulse Response Filter (FIR)

We must capture the timing and resource-usage behavior of the candidate architecture's hardware components in abstract/non-evaluated VHDL models. (Non-evaluated means that the actual data operations are not performed; rather, just the time required and the control aspects are modeled. Another term for this type of abstract model is "token-based".) Algorithmic operations that have been partitioned into software are expressed in software descriptions. The control aspects of these software descriptions must be interpretable by the non-evaluated VHDL models of the programmable hardware components.

The architecture model, taken as a whole, consists of the structure interconnection of the components, their abstract VHDL models, and the associated software descriptions. The architecture model forms the executable specification of the architecture design, and the VHDL component models form the executable specifications of the hardware components that are passed on to the hardware design process.

Associated with each VHDL component model is a VHDL test bench. The VHDL test bench is a set of VHDL modules that provide stimulus to the component model and check its response, so that we can verify the behavior of the component to meet its requirements. We design the VHDL test bench for each hardware component before the component model is designed. Since the architecture model is more detailed than the system-level performance model, the more precise results of its execution are back annotated to the system model. Likewise, as we obtain the results from the more detailed hardware models in the hardware design process, they are back annotated to the architecture and system models, which makes them precise.

3.2.3 Design For Test Tasks in Architecture Definition

Architecture Definition is comprised of three sub-process steps: functional design, architecture selection and architecture verification. The primary focus of the DFT activities is to develop a test strategy and architecture consistent with the requirements captured during system definition. DFT interacts with the functional process to affect architecture decisions and component selection. A virtual prototype of the test & maintenance architecture is developed and used to perform HW/SW codesign with the test software.

The functional design step provides a more detailed analysis of the processing requirements (including BIST), resulting in initial sizing estimates, detailed data and control flow graphs for all required processing modes to drive the hardware/software codesign, and the criteria for architecture selection. Figure 3-23 shows the DFT steps which occur during functional design.

During architecture selection, various candidate architectures are evaluated through iterative performance simulation and optimized to appropriate levels of detail. A trade-off analysis based on the established selection criteria results in the specification of the detailed architecture, and software partitioning and mapping. As part of the trade-off analysis, information is used from the required cross disciplines such as reliability and testability (either manually or through design advisors) to populate the trade-off matrix. Figure 3-24 shows the DFT steps which occur during architecture selection.

This portion of the process is heavily dependent on the reuse of architectural (hardware and software) components to provide significant time-to-market improvements. In addition, during architecture selection, all software not represented by the DFGs is designed. Based on the requirements, the non-DFG software may include BIST[1], downloading data and code, and diagnostics. The virtual prototype, VP1, produced during architecture selection is not a full system prototype, since function and performance are simulated independently and may or may not be coupled with the overall control mechanism. Several architectures may be selected for verification during the following step (i.e. primary candidate and risk reduction alternatives). The Architecture Selection level test strategy diagram, TSD1, and testability architecture, TA1, are developed concurrently with VP1 for each candidate architecture.

[1] At the time of this writing, the feasibility of capturing the overall system level BIST function with DFGs and previously developed primitives is being investigated. The goal is to develop BIST functions with the same tools/methodology as the other functions.

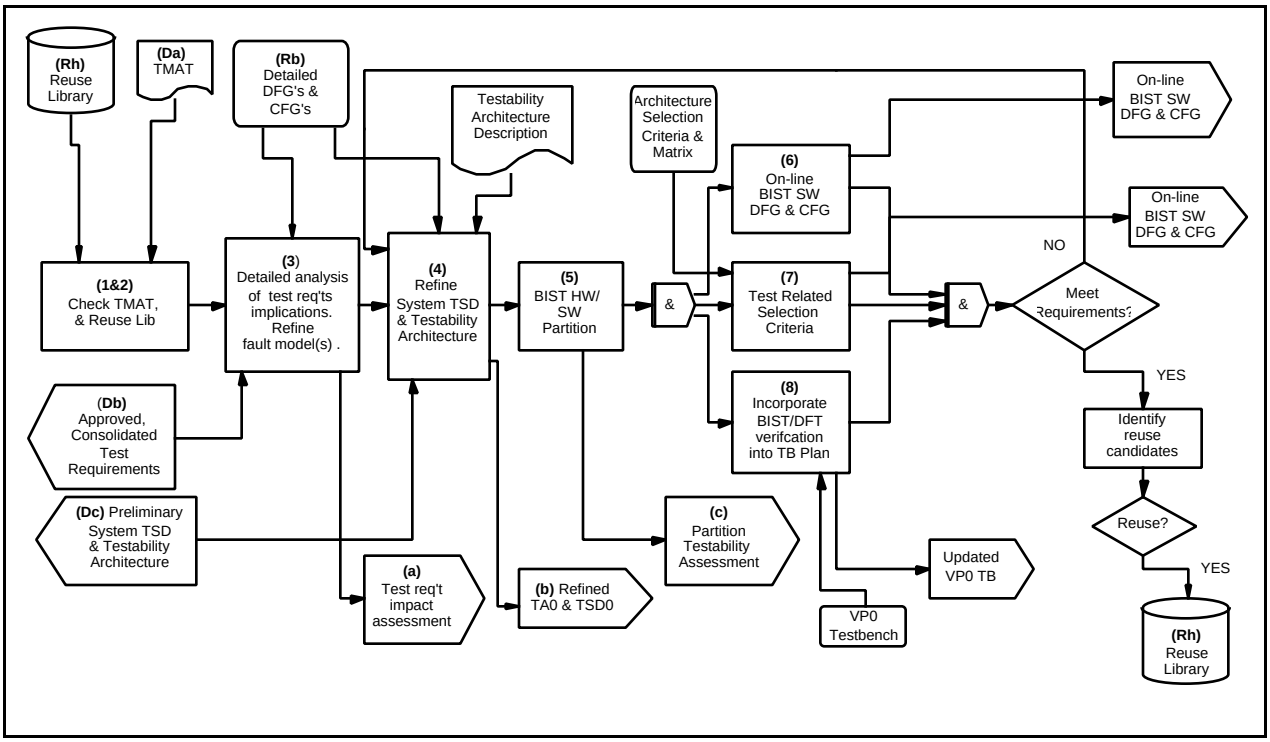


Figure 3-23. DFT steps in functional definition flow diagram.

Architecture verification is the process of hierarchically simulating both the functionality and increased performance detail of a selected architecture candidate. Up to this point the overall functionality has not been verified. An integrated, simulation-framework supports mixed-domain simulation so that high-level performance and functional simulation can be coupled with ISA or RTL VHDL simulators, hardware emulators or hardware testbeds. The goal is to validate operation of all architectural *entities including embedded test* and the interfaces between them before detailed design. Software partitions are autocoded to produce software modules translated from the processor independent library elements to optimized, processor specific implementations. The resultant virtual prototype, VP2, is passed onto detailed design. The Architecture Verification level test strategy diagram, TSD2, and testability architecture, TA2, are developed concurrently with VP2 for the selected and verified architecture. Figure 3-25 shows the DFT steps which occur during architecture verification.

During the architecture definition step, the "lead, follow, or get out of the way" strategy for COTS at the testability architecture level is developed. Controllability and observability analysis results are used to determine what solution(s) could be examined for sections anticipated to employ COTS. The possibilities are explored of absorbing some or all COTS functions (that are in non-testable devices) into ASICs or FPGA type devices that incorporate at least boundary-scan, if not BIST. A refinement of the COTS fault model also takes place.

See Sections 3.2.2-3.2.4 of the DFT Methodology for further details of the Architecture Definition Step.

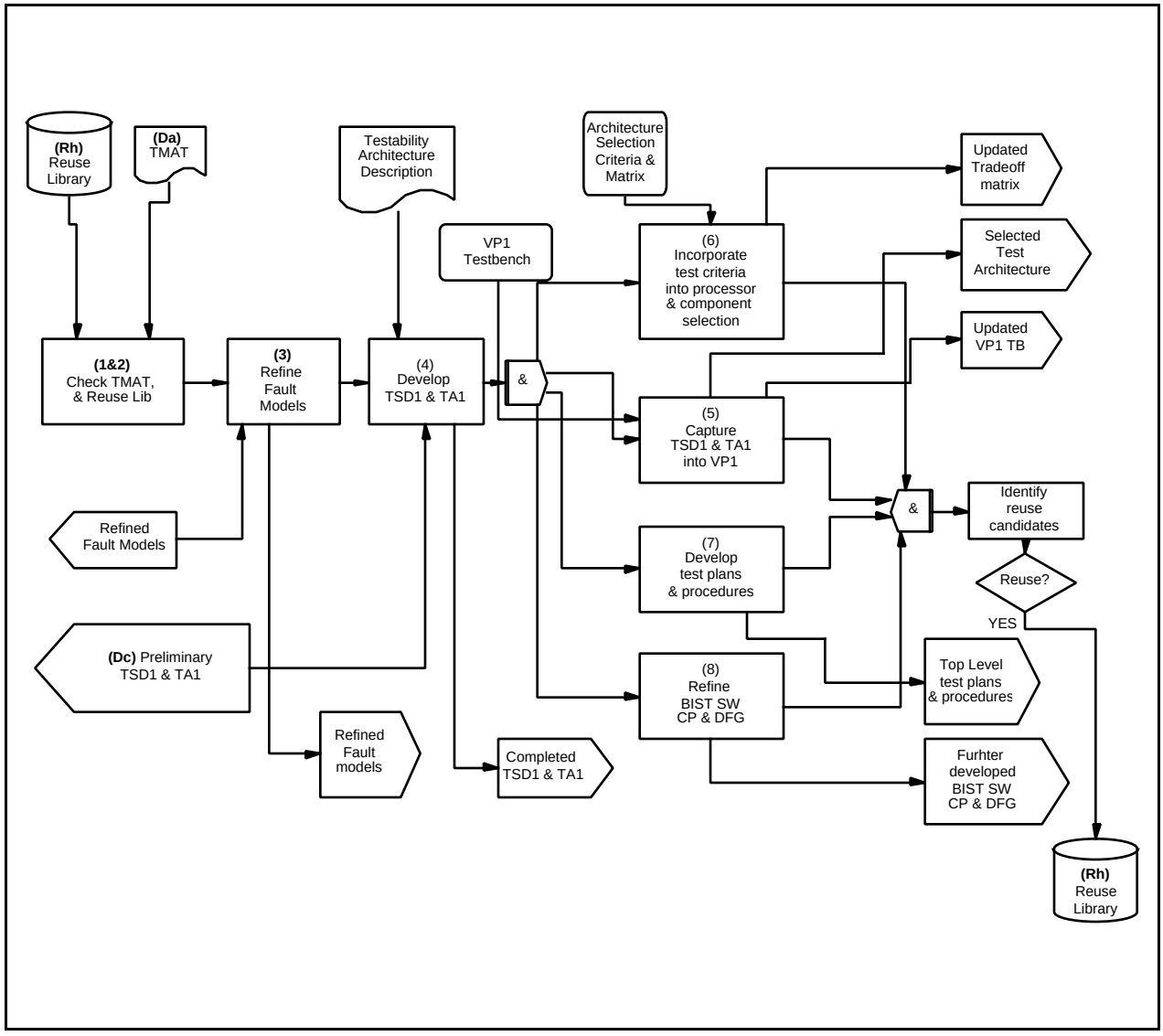


Figure 3-24. DFT steps in architecture selection flow diagram.

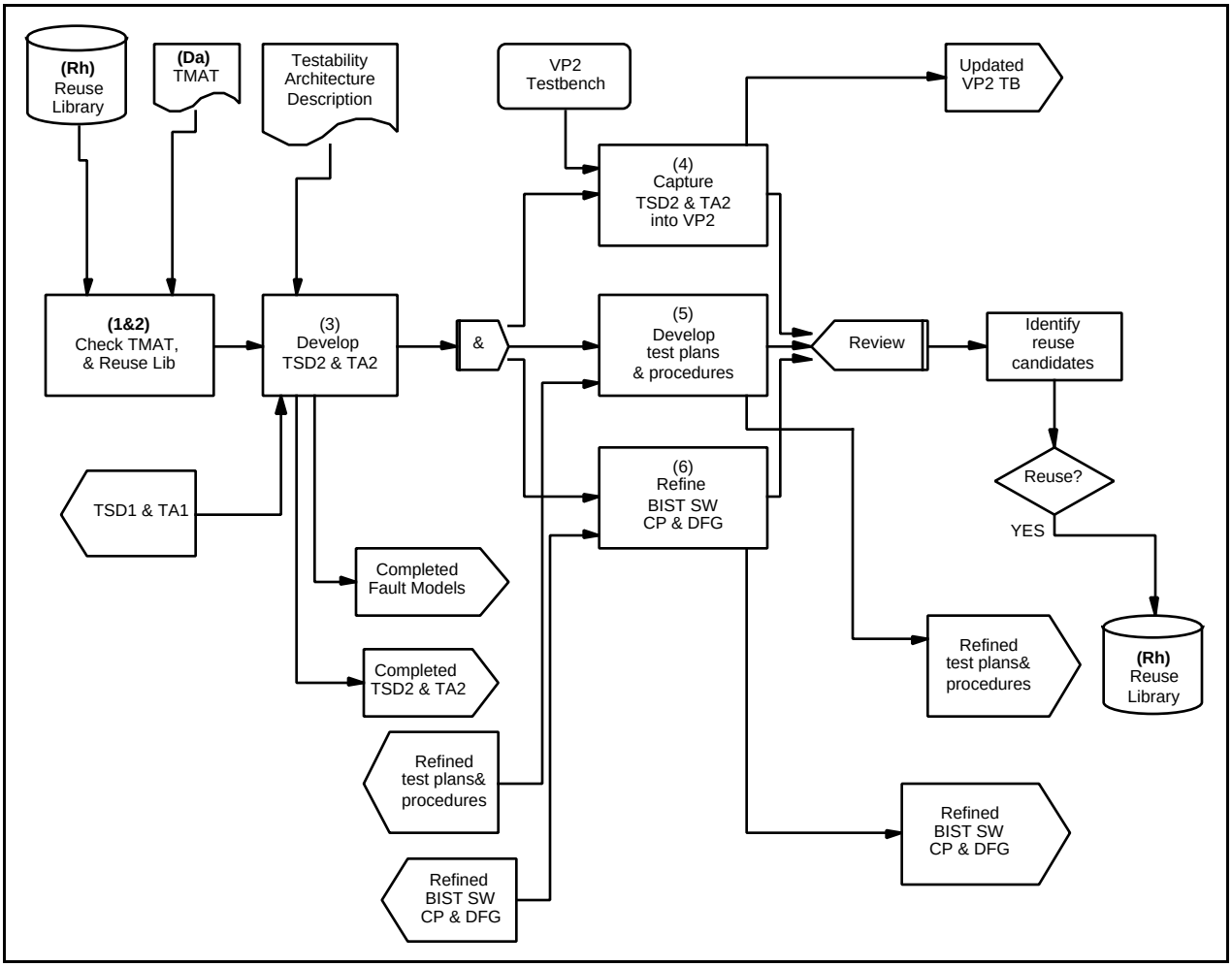


Figure 3-25. DFT steps in architecture verification flow diagram.

3.2.4 Role of PDT in Architecture Process

The architecture process is not only an ideal candidate for concurrent engineering, it is defined in such a way that concurrent engineering is mandatory. By definition, the process includes elements of systems, hardware, and software engineering, and it is supported heavily by the other disciplines, as reflected in the integration of hardware estimates, costing, mechanical complexity, thermal issues, software support, and RAM-ILS parameters into the architecture selection criteria. The following paragraphs describe the role of the PDT disciplines in the architecture process.

- Team Leader — The team leader is the principal interface with program management who monitors progress on the subsystem and ensures that all system programmatics are satisfied. It is assumed that the team leader has participated to some degree in the systems process and at a minimum has participated in the SDR. During the architecture process (which encompasses hardware/software codesign and verification of the selected architecture), the team leader and program management schedule the appropriate design reviews at various stages of the process. At the top-level there must be a review (PDR) of the selected candidate architecture, along with all the inputs that make up the weighted selection criteria. This provides a go/no-go gate before the detailed architecture verification. This is not a sequential process, however, since all the appropriate disciplines have been involved in the architecture selection process to some degree. Much of the effort during the architecture verification process is aimed at producing the detailed support for the subsequent Critical Design Review (CDR). In reality, many less formal reviews will take place regularly within the PDT to evaluate ongoing architecture development. Inserting automated tools into the architecture process provides the opportunity to

evaluate more architecture options, which increases the need for multidiscipline review. Program management representing the IPDT will hold meetings with the PDT leader(s) to assess status regularly.

- Engineering — The architecture engineers have the lead role in the architecture process. The architecture process is a new process that is being inserted into the design methodology. It includes portions of the processes traditionally performed in the system design and hardware design processes, along with key elements of the software process. The architecture process is truly hardware/software codesign in that the candidate architectures reflect the performance of the software on the architecture. Therefore, the composition of the engineering team is perhaps different from the traditional one in that it probably includes systems, hardware, software, and test engineers as principal contributors, along with support from mechanical, thermal, etc. During the architecture process, all aspects of the candidate architectures are considered to develop the weighted selection criteria that must reflect the overall system requirements. Systems engineering ensures that system requirements are met and performs additional system-level trade-offs based on the results achieved in the architecture selection process. Software engineering evaluates library element sufficiency and initiates both prototype element building and library element validation. Test engineering ensures that testability is addressed in the trade studies.

  Like the other disciplines, the mechanical engineering representative(s) on the PDT primarily contributes to defining the selection criteria parameters based upon the overall program requirements. Given the complexity of a given architectural design, it may be necessary for mechanical/packaging personnel to provide inputs regarding the desirability of one design over another based upon the amount and/or type of cooling necessary to implement the design within the form factor required. The total power requirement of each candidate architecture may meet the overall power budget allocated during the system design, but the power dissipation per volumetric unit may dictate different approaches for the different architectures. These characteristics may impact the overall packaging approach, accessibility for test, maintainability, and cost; all of these should influence the ultimate architecture selection.

- Manufacturing/Product Assurance — As part of the PDT, the manufacturing representative works with engineering to provide inputs that are factored into the selection criteria for any given architecture. The manufacturing representative provides inputs regarding the degree of difficulty of the manufacturing process for any candidate architecture or packaging approach. If a design requires that a new or special process be implemented, significant cost, schedule, and risk impact may be associated with the design. These inputs must be an integral part of the architecture process.

- Test — Test engineers evaluate the various architectural candidates with respect to test requirements. One aspect of test engineer participation includes providing recommendations or weightings for different architectural elements based upon their ability to support test. As an example, one processing element may support the JTAG standard for boundary scan, which increases the likelihood of available software for testing; another processing element may provide a non-standard boundary scan capability, which increases the cost of testing. Inputs may also be provided on the level of validation, which must be performed in the architecture verification process. Obtaining these types of inputs from test engineering as early in the process as possible optimizes the potential for minimizing test cost and schedules later in the program. The ability to evaluate alternative architectures quickly makes it possible early in the overall design process to make and evaluate product substitutions based on inputs from test engineering.

- Producibility — The architecture selection must be influenced by all aspects of the product development and manufacturing process. We anticipate that the primary method of incorporating these influences at the architecture level is to provide a mechanism for biasing the selection criteria.

- Product Assurance — Minimum input is required during the architecture process.

- Parts Management — Provide technical support to engineering during the trade study process, if necessary. Minimum input is anticipated during the architecture process.

- Reliability, Availability, Maintainability (RAM) and Safety — The RAM inputs are of particular importance during the architecture process because they are directly related to integrated logistics support and LCC. Provision in the methodology to obtain early estimates of these factors for various alternative architectures may prevent the need for additional design iterations later in the process. Ongoing reliability assessments will verify that final architectures meet requirements.

- EMI/EMC/TEMPEST — Provide guidance regarding the appropriateness of architectural library elements with regard to the customer requirements. If appropriate, restrictions can be placed on the usability of various library elements.

- LCC — Work with engineering and manufacturing to assure that the architecture decisions properly reflect the long-term impacts on the program.

- Sourcing — Sourcing personnel provide the interface between the PDT and the suppliers. During the architecture process, it will become obvious which architectural elements are being favored. Sourcing should provide information via the suppliers regarding the availability of components to support the product development cycle. If particular processing elements are available in limited quantities to support early prototyping, but cannot be available in sufficient quantity to support the required program schedule, this information is critical to the architecture selection process. In addition, sourcing may contribute to architectural component selection by providing current information regarding costs, or renegotiating volume prices based upon the design under consideration. All this information should be considered during the architecture selection process, since it can impact the overall program cost and schedule.

- ILS — Logistics personnel ensure that the PDT recognizes the implications of architecture decisions on the overall support requirements of the signal processor.

3.2.5 Design Reviews in Architecture Process

System Requirements Review (SRR)

Key members of the architecture design team participate in this review. The architects provide inputs during system design with respect to the appropriateness of the allocation of requirements to the signal processor.

System Design Review (SDR)

Key members of the architecture design team participate in this review. The architects provide inputs during system design with respect to the appropriateness of the allocation of requirements to the signal processor.

Architecture Design Review (ADR)

The architecture design team leads preparations for this review and prepare the design package. At ADR, we present all architecture trade-off studies performed and the selection criteria developed. We present and review the final recommendation of an architecture to be carried forward. The results of all preliminary performance and behavioral simulations that support the recommendation are included in the design package. We present all architecture verification simulations and any additional architecture selection trade-offs we conducted as a result of the verification process. We present detailed results from the verification steps to ensure that the architecture design meets all specifications before release to detailed design. We establish the influence of all disciplines contributing to the architecture selection and define all external interfaces to/from the signal processing system.

Detailed Design Review (DDR)

The architecture design team supports preparations for this review and supports the preparation of the design package.

Production Readiness Review (PRR)

The architecture design team supports this review based upon the participation in the integration and test activities and any product refinements that occurred since DDR.

3.3 Detailed Design Process

The main objective of the detailed design process is to transform the architectural description of the design into the detailed hardware and software components that we will develop, manufacture, and integrate into a prototype processor. Figure 3-26 shows the top-level view of detailed design.

We partition and design designs at the behavioral-level (RTL or higher) for processor-level nodes to the appropriate level for all necessary components. This includes ASIC, MCM, and board-level designs, as well as backplane/chassis designs. The process incorporates mechanical and manufacturing elements to ensure that designs meet all specifications for the particular applications. From an industry standpoint, the detailed design process is the most mature area of design, and the improvements on RASSP are optimizations to support rapid prototyping. Toward this end, the process heavily favors two areas:

- Design reuse of validated elements from the RASSP component library to minimize detailed design time
- Synthesis of chip and board-level components from the component libraries.

In the case of software, we must generate any support code for the signal processor that has not been completed. This includes initialization, bootup, diagnostics, test software, and any support software required for integration and test which is not part of the operational software.

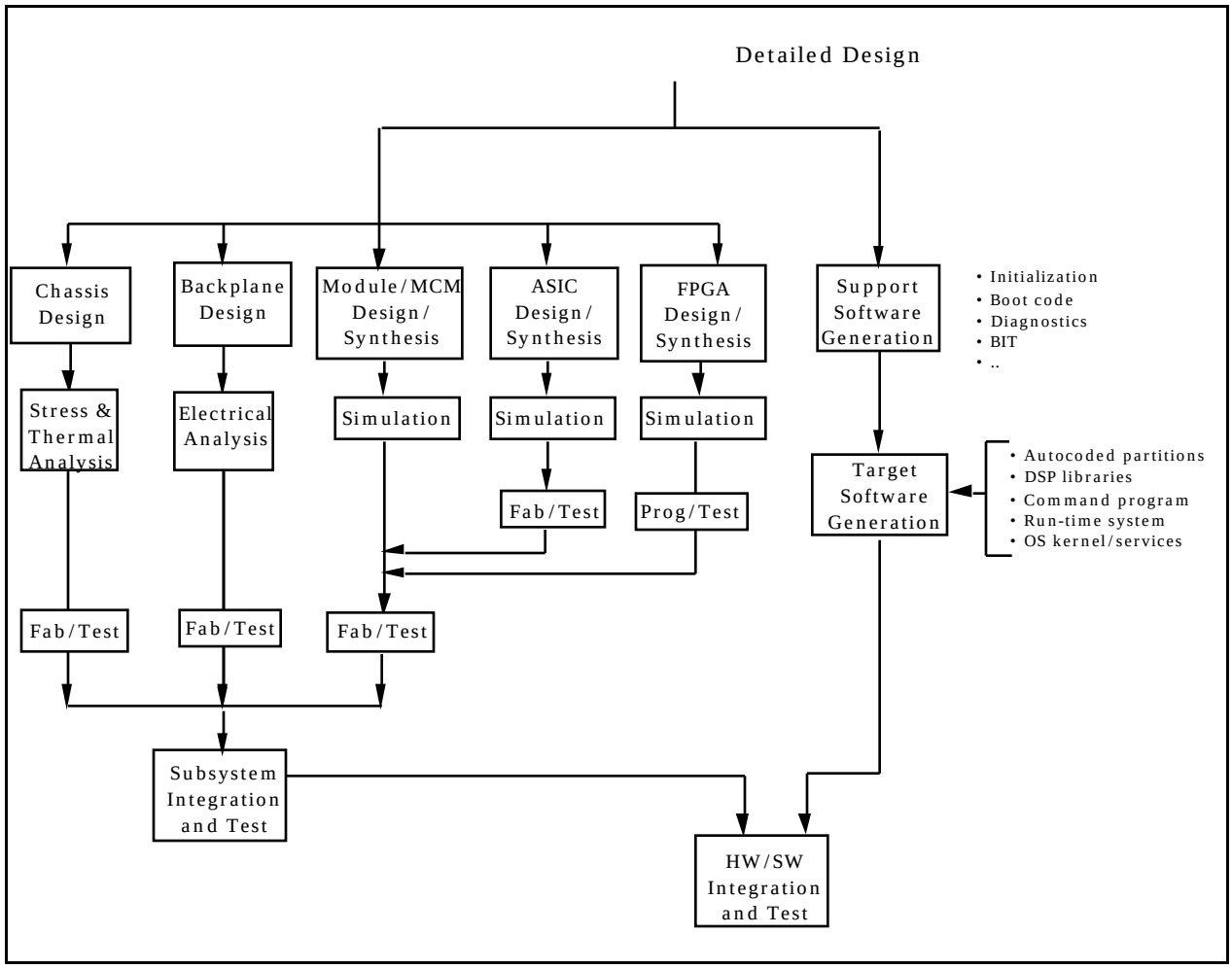


Figure 3-26. RASSP detailed design process.

Like the architecture process, this portion of the methodology supports hardware/software codesign, as these elements are fully verified together on the virtual prototype up until the design is released to manufacturing. We revalidate software that was verified during architecture verification as we develop more detailed models for any of the components. This process provides continual verification as the designs progress toward physical hardware. The transition of these designs to manufacturing is supported by the RASSP enterprise system to provide electronic linking capabilities supporting both concurrent interaction between design and manufacturing, and direct transfer of designs to manufacturing sites.

3.3.1 Software in Detailed Design Process

We performed much of the work normally associated with applications, control, and communications software development in the architecture process in the RASSP methodology. Detailed software design is concerned with aggregating and/or translating the compiled HOL code verified at the architectural level to downloadable target code. In addition, we will generate low-level software modules to support initialization, downloading, booting, diagnostics, etc. on the target hardware. Since many of the support software functions have not been tested in the architecture process, we must verify these functions via simulation during detailed design.

We generate downloadable code by taking the autocoded and hand-generated portions of application and control code, and combining them with the communications and support software into a single executable for each processor. The run-time system, which provides the reusable control and graph management code, will have all the hooks required to interface with the support code.

The software aspect of the detailed design process is the production of the load image. The items used and/or produced in this phase are described in the following paragraphs.

Graph Realization

A graph realization is a compiled version of the equivalent application graph. It is composed of data structures fully describing the software part of a graph, except for the values for control parameters and the actual parameters that will be bound to formal parameters. These actual values and parameters are defined at graph instantiation.

Partition Primitive Interface Description (PID)

A partition PID is the source code that represents a partition graph. The source code is written in the computational element's native language. Consequently, the architecture description and mapping the partitions to the architectural elements is required to produce the PIDs. The partition PID is produced from a partition in the partitioned graph and executes all the target processor-level primitives necessary to implement the domain-level primitives in the partition graph for the specific target processor. These partition PIDs are then compiled by the computational element's native language compiler so that they may be input to the load image.

Load Image

The load image is the end product of the software development process. To build a load image for an application, we combine the graph realizations produced for each equivalent application graph and their associated partition PIDs with all required microcoded primitives, the run-time object code, command program, and the kernel operating system for the selected architecture. The process is shown in Figure 3-27. This load image may then be downloaded to the target hardware and executed.

For the final test of the application, we run the test vectors used to validate the executable functional specification on the target architecture running the load image. The output vectors from this execution should match exactly with the test results from the executable requirement specification.

Run-Time System

A reusable run-time system will be provided that is built on top of operating system microkernel(s). This system is a set of run-time application programs that constitute a virtual machine which performs all graph execution functions and all system interface, control, and monitoring functions. These programs may be reconfigured to execute on appropriate processors of the candidate architectures. We will interface them to the kernel operating systems of supported processors as part of validating a computational or non-computational element of the Model Year Architecture. Translation software translates PGM graphs to forms executable by this system. Interfaces are defined for external data, control, test and debugging, and monitoring system functions. The run-time system provides reusable signal processing control/communication code based on standard operating system services. Availability of this reuse package will make a PGM implementation of the RASSP methodology economically practical for a range of medium to small system users who could not otherwise afford implementing a run-time graph execution system. Traditionally expensive run-time support development and testing will be reduced to run-time reuse package reconfiguration and new element validation in some cases.

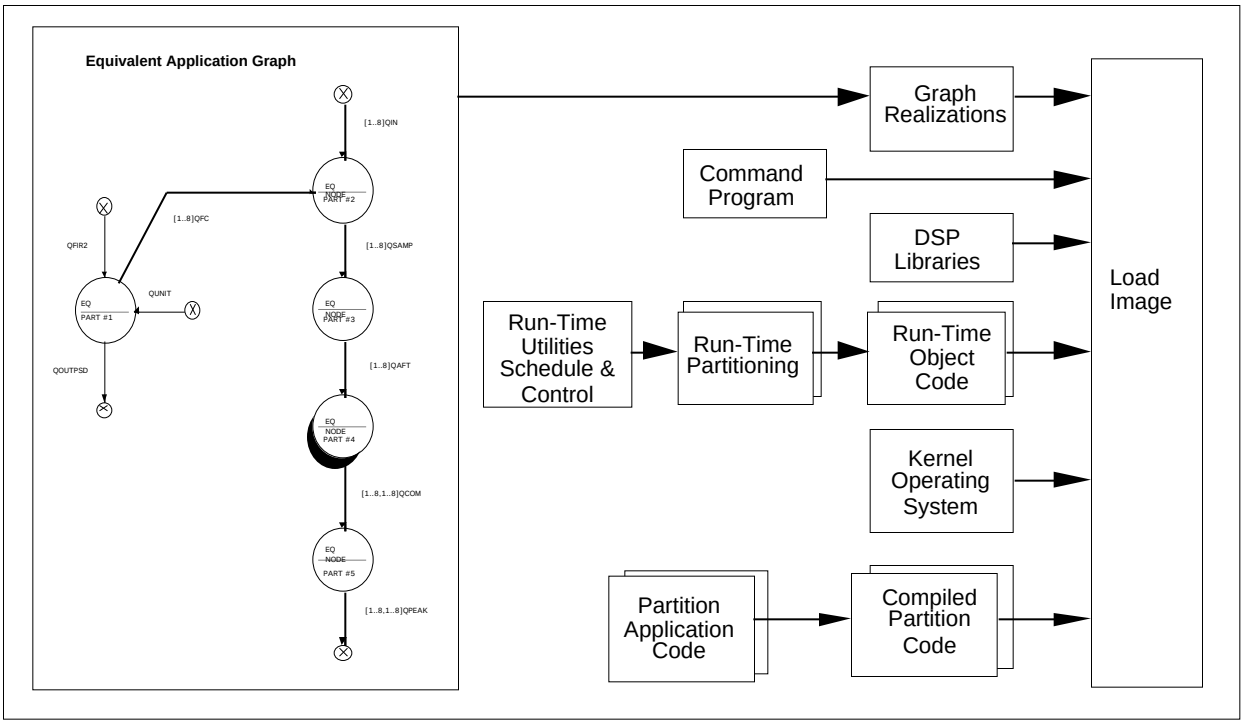


Figure 3-27. Load image building process.

Support Software

In addition to all of the DFG-related software, we will generate low-level software modules to support initialization, downloading, booting, diagnostics, etc. on the target hardware during the detailed design process. Since many of the support software functions were not tested in the architecture process, we must verify these functions via simulation during detailed design.

We generate load images by taking the autocoded and hand-generated portions of application and control code, and combining them with the communications and support software into a single executable for each processor. The run-time system, which provides the reusable control and graph management code, will have all the hooks required to interface with the support code.

3.3.2 Detailed Hardware Design

At the top level, the detailed design process for hardware is, for the most part, the same for boards, MCM, ASICs, etc. However, at the lower levels, these designs are quite different and use different tools. The hardware process is partitioned as a function of the hardware element, e.g., chassis, backplane, board, chip, etc.

We start the overall hardware design process with the subsystem (e.g., a signal processor) components definition requirements from the architecture verification process described in Section 3.2.1.3, at the level of detail of a Type C specification (MIL-STD 490). Architecture verification previously analyzed these requirements, performed architecture trade-off studies (supported by VHDL simulations at the algorithm and architecture level), partitioned the functional requirements into analog/digital/mechanical, and generated documents for all individual elements, including backplanes (frames/cabinets), modules, and ASICs.

The hardware design process flow is shown in Figure 3-28. In RASSP, partitioning is driven by component requirements from the architecture process, with a heavy predisposition toward using library-based components to support the in-process design concept. Where possible, we use off-the-shelf modules, MCMs, ASICs, chassis, and backplanes, and we will be aided by knowledge-based advisors with data on available hardware elements to drive synthesis-based approaches. We will specify

off-the-shelf hardware for procurement, and we will design custom hardware for fabrication. All hardware must be modeled or must have testbed hardware in place for joint hardware/software simulation and verification. The concept of hardware/software codesign at this level is a new element introduced by RASSP.

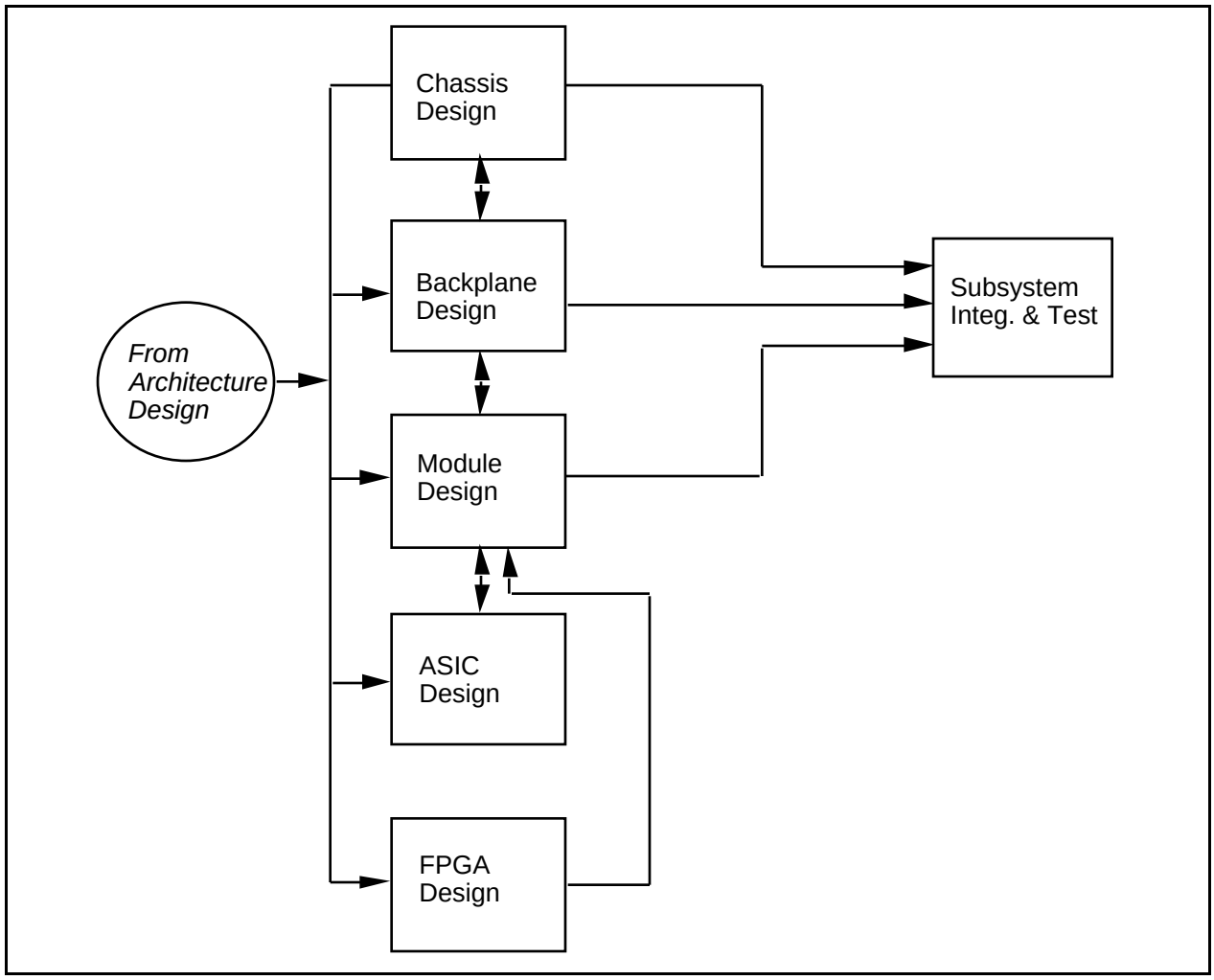


Figure 3-28. Hardware design process flow.

In interactive logic design, we develop a structural VHDL description from the algorithm architecture level-VHDL descriptions, which include DFT structures. We simulate this model to verify functionality and performance, then synthesize it, generate a layout, and perform a thermal analysis. Back-annotated simulations are done followed by a critical design review after verification. We then manufacture hardware and test it.

In the RASSP methodology, architecture design defines the top-level hardware architecture and major elements that go into it.

We decompose the overall hardware design process into several distinct yet interrelated subprocesses:

- Module design
- ASIC design
- FPGA design
- Backplane design
- Chassis design
- Subsystem integration and test

We start the module design process with inputs in the form of the Module Requirements Specification generated by architecture design, which partitions the module architecture into a combination of existing components and new or existing ASICs. The ASIC design process is spun off at this point.

The module is the fundamental line-replaceable unit (LRU) for the signal processor hardware. It is the primary focus of testability, reliability, and maintainability analyses.

ASIC design takes the Type C specification from architecture design, which may have been augmented by module design, and develops behavioral or RTL models of the ASIC using VHDL. This is followed by functional behavioral simulation. Next, we perform detailed design (synthesis) of the ASIC functional blocks, followed by gate-level functional simulation, using the same test vectors and expected outputs as in the behavioral simulation. The behavioral and gate-level simulations must agree 100 percent before proceeding to the next step.

In parallel with ASIC design, we perform the detailed module design involving interactive logic design, simulation, and timing analysis. Functions are partitioned to FPGAs, if applicable, and FPGA design begins.

We perform the first module-level simulations using the ASIC behavioral or RTL model(s) developed by the ASIC designers. As the ASIC design progresses, gate-level ASIC models, and FPGA models may be brought into the overall module simulation environment, depending on overall module complexity. A test bench that is usually generated by architecture design is used on the module as a check for correct requirements mapping and implementation. We then perform timing analysis and fault simulation. When the module is deemed correct (functionally and in terms of timing), after the Module Preliminary Design Review, module preliminary design is complete.

FPGA design, also in parallel with module design, involves the development of a programmable logic device from its beginning as a behavioral VHDL model through detailed design, synthesis, place and route, and programming using a device programmer. This process is similar to the ASIC design process.

We start final module design with the physical layout. We back-annotate added signal delays due to module routing to the simulation model, and the perform a complete module simulation. In some cases, we simulate multiple modules at the structural level to test interfaces and a higher level of functionality. The test bench used to stimulate a multimodule simulation is the same as for earlier behavioral simulations. After the Pre-Release Design Review, manufacturing fabricates and assembles the first-piece module and engineering tests it, using the same test vectors used during module simulation. The Final Design Review is held, after which all module documentation is signed off.

Backplane design, which began in parallel with module design, involves determining the intermodule wiring scheme (printed wiring board and/or wirewrap) for the collection of modules, and performing detailed analysis of the signal integrity of the backplane, using transmission line and crosstalk analysis tools.

Chassis design also begins in parallel with other design activities and encompasses the mechanical and electrical design and analysis of the structure that holds the backplane(s), power supplies, cable and harness, I/O connectors, and other miscellaneous hardware. With appropriate CAD tools, the chassis is designed and a 3D model is generated. This model is used to analyze stress, temperature, mechanical tolerances, and dynamic characteristics.

Also in parallel with module and ASIC design, we re-evaluate the chassis mechanical conceptual design approach against the program subsystem-level requirements (B2) specification. We develop a mechanical design concept for the enclosures and electrical packaging, then generate solid models and a C-level specification. This effort requires support from manufacturing for producibility and cost trade-offs. Producibility analysis is built into the mechanical design's early stages. The Preliminary Design Phase refines the chassis design concepts developed during the Conceptual phase. We generate and integrate

solid models and detail layouts, where applicable, into overall mechanical design layouts that incorporate key features, interfaces, dimensions, etc., for the particular mechanical design. This reduces risk due to misinterpretation or missed revisions, and provides the up-front discipline and integration to assure a correct initial database during the Preliminary Design Phase. A Preliminary Design Review (PDR) is held following completion of preliminary design and prior starting detail design. At this point, we have completed all detail and interface layouts, selected major parts, identified materials and processes, and completed and analyzed major chassis models. We begin preparing test plans and issue long-lead parts lists to manufacturing. The PDR ensures that these tasks have been satisfactorily completed, preliminary design is complete, and detail designs can be generated from the preliminary design.

In the final design phase for the chassis, we generate the actual detail design and assembly drawings used to fabricate and assemble hardware end-items. Designs are transferred electronically from solid models into 2D or 3D formats used to fabricate hardware without unnecessary modification. Outputs for the Final Design Phase include issue of Release Packages that allow manufacturing and sourcing to efficiently fabricate or procure hardware. A Final Design Review (FDR) is held before issue of the Release Packages to ensure that hardware designs are correct and meet the requirements of the subsystem level specification. The major portion of the mechanical engineering effort is complete following the FDR and issue of Release Packages.

Subsystem integration and test is the last process in the overall hardware design process. We follow a phased integration approach where we integrate fully-tested individual pieces into the next higher level. Initially, we integrate the modules into backplanes, then test them; the tested backplanes are then integrated into frames and tested. The eventual result is a complete, fully tested subsystem. Detailed test procedures are followed in all phases of this process.

The following sections describe these elements of the process in more detail.

3.3.2.1 Module/MCM Design Process

A module or MCM is a design that encompasses a logical function (or a group of logical functions) that can be implemented with circuit devices on a single circuit board. A module may include CPLDs/FPGAs or ASICs with their own design process. The module designer is responsible for the integrity of the outputs from these lower-level processes and for verifying their proper operation within the target digital module. The module/MCM design process is detailed in Figure 3-29.

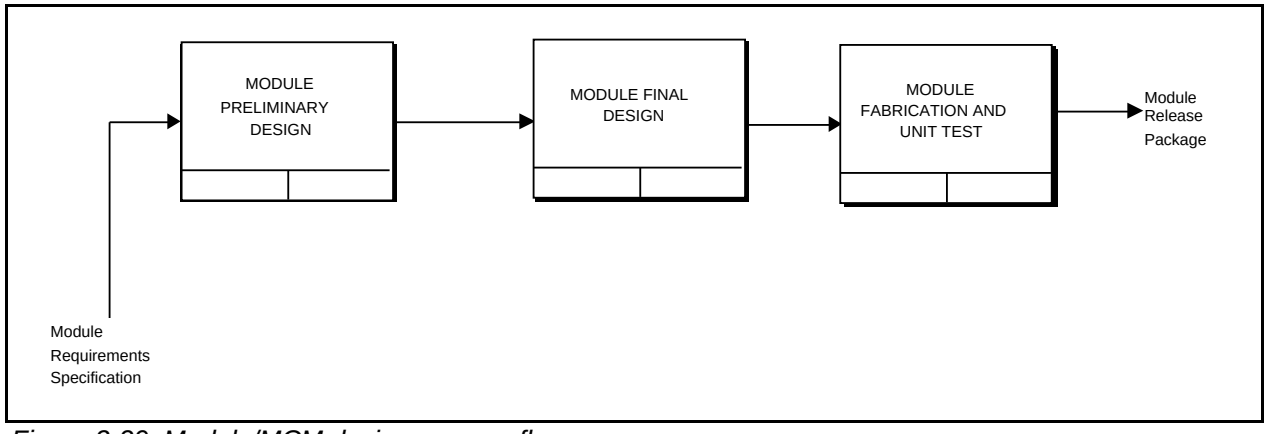


Figure 3-29. Module/MCM design process flow.

Module Preliminary Design Phase

The module design process starts with preliminary analysis and review of the Module Requirements Specification. We search the RASSP design reuse library concurrently with the preliminary analysis after reviewing the module requirements. If no existing module meets the requirements, then we assess the feasibility of implementing the function and develop the overall architecture of the module. Sizing and

partitioning the logical functions required on the module must be undertaken to:

- Verify that the function can be implemented on one module with existing technology within cost constraints
- Allocate the logical functions between CPLDs/FPGAs, discrete logic, and ASICs specified by architecture design.

We review the test requirements, testability goals, BIT requirements, and target tester for this module. This step includes fault coverage goals, fault isolation goals, identifying replaceable units (RUs), type of testing to be performed (i.e., functional, in-circuit), type of test machine that will test the module, design-cycle time, tester-cycle time, level of test (module, system), and information on any standard test busses that may be required for the system.

Another important aspect of testability goals is whether and how a circuit must be probed. For example, it may be acceptable to probe a board in factory test (before conformal coat), but probing may not be allowed for field test. It is important to know and analyze all these constraints and requirements before developing a testability concept and plan.

We generate a concept for designing-in testability for this module. As we identify the test objectives, we select the best method for implementation, such as off-line (external stimulus and response), built-in (internal stimulus and response), and self-checking (e.g., parity checking during operation). The output is a preliminary testability concept document that the designers use when performing the detailed electrical design.

We may need to conduct trade-offs with architecture design and backplane designers following this analysis to achieve the desired module requirements. Manufacturing and mechanical engineering should be involved to ensure that the proposed design can be produced within project, technology, and cost constraints. Reliability and maintainability engineering should be consulted so that the overall architecture and parts chosen meet reliability/maintainability needs. Sourcing should be notified of any possible long-lead parts selected to determine if the procurement time for these devices meets schedule constraints.

- Develop Module Behavioral Model — All modules are an integral part of a larger system and perform some unique subsystem function. In most cases, the architecture design engineer modeled the unique architectural elements using VHDL to gain confidence that all functional blocks will interact properly. In these cases, each partition (i.e., the module definition) should have had a high-level model developed that can be used by the module design engineer as the starting point for behavioral model development.

  The first step in designing a behavioral model of a module is to finalize and define, in detail, the interconnections between functional blocks in the block diagram of the module. After we define all the I/O to each functional block, we can then develop the behavioral model for the function or further break the function into smaller blocks. Incomplete interconnection definition increases design time and increases interdependence of functional block designs. A module requiring multiple design engineers should have very accurate inter-block definition; changes of I/O should be immediately documented for hand-off and future review.

  When developing the behavioral model, the module design engineer should reflect the inherent structure of the design. If, for example, the module has a pipelined architecture, the pipelines and the functions performed by each pipeline stage should be evident from the model. The purpose is to produce a model with sufficient detail to guide gate-level design.

  The design database is continually updated with more detailed descriptions of each function as the information becomes available. The I/O definition of each block is detailed enough to allow independent, concurrent design of each function. We then compare functions to the reuse library functions to eliminate duplication of effort. We enter new, common behavioral models into the library database as they are developed and validated.

Also included in this step is development of the ‘tester’ model, which applies input vectors to the model and collects and analyzes the results. This is a simple model written in the same language as the behavioral model or a complex suite of programs integrated into a complete simulation environment.

- Generate Module Test Plan — We develop the Module Test Plan in accordance with local standards unless a different format is called for by the customer contract. The Test Plan should define all steps in the simulation process, including the strategy for generating test vectors that exercise all logic blocks, the method of performing comparisons with higher-level simulations, and the merging of ASIC behavioral and gate-level models and FPGAs into the simulation environment. The plan for meeting fault simulation stuck-at-1 and stuck-at-0 detection requirements is also addressed in this document. The Test Plan should be sufficiently detailed to allow efficient transfer of all information regarding the simulation effort to anyone associated with the module design. An informal peer review of the test plan is desirable to identify and

correct any deficiencies in the plan before the start of simulation. Test engineering and manufacturing should be invited to the peer review.

- Perform Behavioral Functional Simulation — We simulate the behavioral model of the module to verify that its performance meets the subsystem and ASIC requirements. This effort may be performed jointly by the module design engineer and the ASIC design engineer to ensure full coverage of all subsystem and module requirements. It is quite common for the module design engineer to discover design flaws in the module simulation that are difficult to detect in a single-chip simulation. This may include the detailed interaction of the ASIC during module reset or instruction abort procedures. Using the latest version of the ASIC behavioral model, the module design engineer can discover potential errors before module integration and test.

- Generate Module Functional Test Vectors — A test vector is a set of bits used to test the operation of the module. In general, the bits define inputs to the module (stimulus) and the expected outputs from the module (response). As each vector is applied, the actual response of the module is compared to the expected response. If these match, the module is performing as designed. Test vectors are required for both simulation of the module and for testing the completed module.

  Generating test vectors for a module generally involves more than just specifying bit patterns. In most cases, we must define a ‘tester’ logic function to generate stimuli to emulate, for example, a bus interface protocol. When a standard microcomputer is involved, the test vectors are limited to the relatively small number of signals needed to reset the processor, plus the (sometimes extensive) firmware that must be downloaded to RAM for subsequent execution by the processor, which in turn generates bus activity, etc.

  There are generally two classes of test vectors: functional and production. Functional test vectors verify that the design works as intended. Production test vectors differ from functional vectors in that the production vectors detect manufacturing defects. The functional vectors may also be used as production vectors, but are usually augmented with additional vectors to detect module faults that the functional vectors do not reveal. Production test vectors generally also use any designed-in-test features of the design, such as controllability and observability points and boundary scan. These vectors must also consider the target tester's restrictions, which include simulation speeds, timing limitations of the tester's drive and detect channels, pattern length, and pattern depth.

  In this step, we emphasize generating functional test vectors, although some attempt at vectors designed for fault detection can be made. In "Perform Fault Simulation," the designer grades the vector set on its ability to detect module faults. Additional vectors needed to increase fault coverage will be produced in that step.

  We perform this step concurrently with "Perform Functional Simulation." As we develop each group of test vectors designed to test a specific function on the module, the simulation is usually run using that group to verify the function instead of generating all the test vectors before running any simulations.

- Generate ASIC/FPGA Requirements — The module design engineer generates a high-level description of each function that is implemented as a FPGA. This description may include possible device types for implementation based on an evaluation of the approximate number of inputs, outputs, and product terms for each function. Both the module and FPGA designers review the FPGA requirements to ensure that the documentation is complete and consistent. Typically, the module design engineer and the FPGA design engineer are the same person.

- Search Design Reuse Library — Concurrently with the preliminary analysis/design step, we search the RASSP design reuse library to determine if an existing (legacy) design can meet the requirements specified for the target module. In some cases, we can make a minor modification to an existing module to meet current requirements.

  The module designer considers all available options before proceeding with the design process. If the library search produces an existing module that meets the requirements, no further analysis is needed. If an exact match is not made, we can check the library for other designs that could be used, either whole or in part. The library also contains entries that make it possible to contact the original designers, who could be valuable consultants. We use the results from this review of existing designs and the cost goals to determine the actual module design approach.

  The Module Requirements Specification, generated primarily during architecture design, is a high-level description of the module being designed. We can generate it after completing a firm definition of the architecture. This document should include the following minimum subset of topics:

  1. Functional Requirements
  2. Functional Block Diagram
  3. Preliminary Parts List
  4. Register level Block Diagram
  5. Preliminary Layout
  6. I/O Pin Utilization
  7. Physical Characteristics
  8. Environmental Requirements

  The detailed electrical design process involves the interconnecting the logic devices required to implement the module function. This portion of the process also includes placing the parts on the circuit board, analyzing module characteristics (Critical Path Analysis, generation of FPGA requirements, etc.), and producing a design database. This step is broken into sections explained in the following paragraphs.

  The module electrical design process presented here assumes that we will simulate modules after the logic schematic is fully captured or the logic synthesized and a netlist generated. A more structured, top-down approach can also be accommodated. The following steps can either be completed for the full design, as shown, or they can be repeated as each functional block of the design is generated during the functional requirements generation step. If a behavioral model was generated previously, mixed-level simulations can be run as the logic for the different functional blocks is generated.

- Interactive Logic Design — We perform the detailed logic design of the module on each logic block concurrently, as appropriate. By designing each functional block separately, we can distribute the design of complex modules among multiple engineers to shorten the overall design cycle. However, each functional block must interface with its neighbors. Therefore, a functional block of the module cannot be considered an isolated design. Information must flow freely between the team of design engineers. Testability should be addressed at the start.

  As the detailed design progresses, it should be captured with the appropriate schematic capture tool.

If architecture design specified the use of any ASICs, any additional ASIC requirements must be provided to the ASIC designers in the form of additions to the ASIC Requirements document produced by architecture design.

We must now refine the testability features of the module introduced in the conceptual design. Additional hardware may be required to provide for testing of all functions.

Information is also required regarding the target module tester. This data minimizes the amount of translation required when the production test vectors are generated. This data should include information about restrictions on device names, signal fan outs and loading, and power and ground pins (for standardization of test fixtures).

- Preliminary Layout (Parts Placement) — After entry of the schematic into the workstation, we must do a preliminary parts placement to ensure that all devices in the design fit on the module. If there is insufficient room to fit the module function on the circuit board, then we must conduct trade-offs. These trade-offs can be alternative logic implementations using logic minimization techniques, using different part types (e.g., CPLDs), or repartitioning the module functionality to other modules.

  Representatives from mechanical engineering and manufacturing should be consulted during this step. The mechanical engineer can help with power dissipation and the distribution of heat across the module. These issues can affect parts placement. Manufacturing and mechanical engineering can handle other concerns related to parts placement, such as the minimum distance parts can be placed from each other, and the positioning of Pin Grid Array (PGA) devices so that solder inspections can take place.

- Critical Path Analysis — The next step in the design process is timing analysis of critical paths. The purpose of the timing analysis is to analyze the timing performance of the design, primarily to determine producibility. Unlike the idealistic typical simulation, a real board is composed of parts whose timing performance varies from typical. In this analysis, we attempt to identify portions of the design where a set of fast or slow parts, or a combination of fast and slow parts, causes the board to function improperly.

  We begin the timing analysis by identifying the design's critical paths. A path for this analysis is the set of wires and components between any two pins in the design. A timing path is one that begins at the module's primary inputs or any internal register's outputs and ends at the module's primary outputs or any internal register's inputs. The path delay is the accumulation of propagation times through each wire and each component on the path. Note that the propagation delay through a component is not a constant, but is a function of the capacitive load on the driving pin. The critical paths of a design are the set of timing paths that have the greatest path delay values, and thus determine the limit of the design's timing performance.

  We use a static timing analyzer for timing analysis in this step. The static timing analyzer differs from a simulator primarily in that the analysis is independent of the stimulus applied. A static timing analyzer determines the path delay for every timing path in the design, and analyzes each for timing violations similar to those identified by the simulator. Timing problems are identified by timing violation reports. One advantage of the static timing analyzer versus the simulator is that it identifies timing problems and critical paths that are not sensitized by the input stimulus. A disadvantage is that it often reports many false timing errors that would not occur in the actual module.

Critical path analysis using a static timing analyzer provides a quick approach to finding timing problems in the design, such as setup or hold time violations. It does not find all timing problems, especially those involving complex CPLD or controller designs. A worst-case timing analysis that uses a comprehensive set of functional test vectors may be required to broaden the timing analysis.

- Perform Power/Loading Analysis — Power analysis determines the module's electrical power requirement. There are maximum power specifications for a module, usually based on its size, which cannot be exceeded. If power specifications are not met, some high-power components may need to be replaced with functionally-equivalent, low-power devices.

  Loading analysis determines the electrical loading on the outputs of module devices, and includes DC and AC components. A device's DC loading is affected by DC current from other devices that flow through its output. A device's AC loading is related to the capacitance presented to its output due to the capacitance of other devices and interconnections connected to it. Excessive DC loading can cause power dissipation to exceed device specifications; excessive AC loading can cause timing problems and signal distortion.

  We send the results of the analyses to various other members of the overall subsystem team: thermal analysis data to mechanical engineers involved in the design of the cooling system; and power and loading data to the backplane designer specifying power supplies and interconnecting modules in the system.

- Perform Functional Timing Simulation — We use simulation to confirm with a high degree of certainty that the first version of a module will function as desired without modifications. We can work our problems with the design on the simulation without the cost of producing a prototype, finding the problems, and reworking the module. We can also assess various design approaches without building a module.

  The purpose of this step is to simulate the module to test its basic functionality, using the previously-generated test vectors. ASICs, if present, will be modeled behaviorally or with hardware models. We use worst-case timing values for all components. We also broaden the timing analysis by performing a worst-case minimum/maximum timing analysis using test vectors and the simulator.

  For the timing analysis, we use the simulator to simulate the minimum and maximum delays of components simultaneously. When an input to a component changes, two transitions are scheduled on the component outputs. The first transition is to an unknown state and occurs with the minimum propagation delay. The second transition is to the final state, which occurs with the maximum propagation delay for the component.

  Simulation output reports all functional timing hazards, such as setup and hold-time violations, illegal edge violations, and excessively 'long' signal paths based on user specifications. Also reported are structural timing hazards (circuit problems arising from the structure of the circuit).

  Once the gate-level design is complete, we must verify the accuracy of the design. The first step is to perform a gate-level functional simulation. This is similar to the behavioral model functional simulation performed previously, except that we use the gate-level design database for simulation rather than the VHDL model. The functional simulation is also concerned with the timing of the gate-level design. The purpose is to verify the functional correctness and timing of the logic.

Mentor's QuickSim II has a mixed-mode capability that allows VHDL and gate level representations to be simulated in the same design simultaneously. Because of this, it will probably be common to perform this step in parallel with the interactive logic design. We compare the output of the detailed gate-level simulation with the previously-verified behavioral simulation to determine the functional correctness of the detailed design. We must isolate discrepancies between the two models and correct them until both simulations produce the same results.

We can design a subset of the entire VHDL model at the gate level and verify it using the test vectors we developed during the behavioral model functional simulation. Once this function is verified, we design the next function at the gate level and verify it, and so on until the entire design is complete.

- Select Parts — This step is required to control parts usage on RASSP module designs. Prudent part selection by the module designer enables designers to take advantage of economies of scale in purchasing; it also enables lower inventory costs to the company and its customers. Procurement should be consulted for availability of long-lead items. Also, some new parts are introduced in the literature well before they are actually available for purchase. Procurement or the local components engineering group should be consulted regarding the availability of new parts. We must also consider the cost of parts chosen for the design to meet cost goals.

- Generate Production Test Vectors — In most cases, we augment the functional test vector set used in functional simulation of the module to support fault simulation and the eventual needs of production test.

- Perform Thermal Analysis — Thermal analysis identifies problems with heat caused by the power dissipation of the module components. Mechanical engineering sets requirements for heat distribution across a module, and if any areas on the module violate this requirement, we may need to move components to more evenly distribute the heat.

- Perform Fault Simulation — A fault simulation indicates how well the stimulus tests the module's interconnections for manufacturing. It can also provide information on the set of potential failures based on a particular failure indication (e.g., pattern of incorrect outputs). The fault simulation provides this 'set of potential failures' through a fault dictionary. A fault dictionary lists faults for each output vector that would cause that output vector to fail. The intersection of all sets of potential failures for each failed vector indicates the set of potential faults on the module under test.

  The results of a fault simulation often indicate design areas that are inadequately tested and that require additional or modified stimulus to fully test. We use this information to create an adequate set of test vectors to fully test the module and also to identify a reasonably small set of potential failures through the fault dictionary. The test engineer who constructs the module test program uses these vectors, and the associated fault dictionary, to construct the test program concurrently with the module design.

  The fault simulation process involves running multiple logic simulations, but each simulation has a fault introduced into the design before it begins. If the response from the faulted simulation differs from the response of the ideal simulation, the fault is either detected or potentially detected. The fault is detected if the response of the ideal simulator was a '1' while the response of the faulted simulator was a '0' or vice versa. The fault is potentially detected if either simulator's response was unknown while the other's was known. The fault coverage is the percentage of all faults that are detected.

Many types of faults, such as opens, shorts, and faults internal to components, are possible in the physical module. It is impractical to simulate all types of possible faults, so we usually limit a simulated module fault to a pin on the module that is grounded or connected to power.

If the fault coverage for the set of test vectors does not meet project requirements, we generate additional test vectors to meet the requirement.

- Generate Module Preliminary Design Document — The Module Preliminary Design document includes all information pertinent to the module design. This document has sufficient data for any engineer or technician who must later work with the module. For example, test engineers use it to debug the test program, and manufacturing uses it to repair boards.

  A designer should include at least the following data items:

  1. Parts List
  2. Block diagram
  3. Logic schematic
  4. Module layout
  5. Timing diagrams
  6. Functional Description
  7. FPGA description
  8. ASIC description
  9. Results of all analyses (power, thermal, timing, reliability, testability, etc.)
  10. Interface Protocol Timings
  11. Power Requirements and Consumption
  12. Simulation Results and Data

- Conduct Preliminary Module Design Review — After completing the module design and before committing the design to fabrication, we conduct the Preliminary Design Review. At a minimum, the project manager, the architecture designer, the module and ASIC designers attend. This review provides a detailed critical technical evaluation of the module design and resolves any outstanding issues. Testability features are evaluated. We review any ASIC or FPGA elements within the module. All disciplines that interact with module design should have representatives present at this review.

Final Design Phase

The final design phase addresses the transition from a functional design to a physical design. During this phase, we accurately place and route the module and analyze the parasitic effects of the module on signal integrity. The simulations performed address actual signal loading, signal crosstalk, line reflections, additional line delay, and skew. We finalize the test vectors and conduct the Pre-release Design Review before releasing the documentation and analysis results.

- Module Place and Route — In this step, we determine the location of the parts on the module based on the preliminary placement completed previously. The interconnections between parts (routing) are also made. In some cases, we must conduct placement trade-offs due to routing concerns. We may have to move parts to minimize the length of critical interconnections such as clock lines.

- Update Module Preliminary Design Document — We update the Module Preliminary Design Document and other related documentation to reflect the results of the behavioral simulation and the electrical analyses performed on the gate-level design. We add the final verified version of the behavioral model to the document. We also update other sections of the Preliminary Design Document that have changed. Such items may include the functional block diagram and other details that might have changed as a result of the modeling and analyses.

- Perform Final Design Functional Timing Simulation — This functional simulation is identical to that performed in preliminary design, except that we use the design database back-annotated with layout

parasitics for final verification.

- Perform Final Design Thermal Analysis — This thermal analysis is identical to that performed in preliminary design, except that we use the final layout.

- Perform Final Design Critical Path Analysis — This critical path analysis is identical to that performed in preliminary design, except that we use the design database back-annotated with layout parasitics for final verification.

- Perform Production Timing Simulation — Module layout parasitics (trace length, number of vias traversed, signal crosstalk, trace parallelism, transmission line effects, etc.) can significantly effect module timing. This step allows the predicted delays caused by layout parasitics to be back-annotated to the simulator for a final simulation run that includes the best known timing information.

  We use a transmission line analysis tool at this point. This tool predicts delays, waveform distortion, and ringing due to transmission line effects. Crosstalk, the coupling of signals from one interconnection path (net) to another, is also evaluated with a suitable tool. The engineer can correct these problems by adding terminations, changing parts placement, and making other changes. We must evaluate the impact of any design changes on the module's cost, manufacturability, reliability, and testability.

- Generate Module Test Procedure and ATE Test Vectors — In this step, we generate the module test procedure document and test database. The test procedure contains the step-by-step procedure for testing the module. It includes such details as module tester setup, assignments of tester pods to module I/O signals, and filenames of test vector files. The information in the Test Plan produced previously can be included in this document. Test engineering and manufacturing should review the test procedure document.

  Also in this step, we produce the test database, which is the set of test vectors/expected responses in the correct format to drive the module tester. If the simulation is properly designed, in this step we simply reformat the existing simulation vectors. It is therefore important to design the simulation so that the simulation vectors can be easily used for module test.

  We use the module test procedure and module test database, along with the post-processed vectors, to generate the necessary files and procedures for the target tester. We then verify the test program on the target test system. Any changes that may be necessary are reflected back into the module test procedure and/or simulation database. Any conceptual changes are noted in the Testability Document and the Design-for-Testability Design Review Checklist, if necessary.

- Design and Build Test Adapters — The test engineer is responsible for specifying the electrical requirements for any hardware used to interface the module to the target test system. Depending on program requirements, this hardware may consist of either a totally unique design, or more generally, a generic test fixture design that is common to several module designs, plus unique interface hardware that provides module-specific signal routing, stimulus conditioning and output loads.

- Generate Module Artwork and Manufacturing Tools — In this step we produce the tools for board fabrication. Artwork is used to generate printed circuit interconnection information. Other tools are needed to control equipment to drill holes in the boards, etc.

- Prepare Module Release Package — We prepare the module drawing package, designated a 'Module Release Package.' This package contains all pertinent data for building the module and is released to internal manufacturing or an external supplier. The documentation package must fully specify the module to the manufacturing organization. It includes assembly drawings, logic schematics, parts list, and machine tool data.

- Conduct Pre-Release Design Review — The Pre-release Design Review is the last design review of the completed design of the module before first-item production. This design review must verify that all

architecture and module functional requirements have been met. We examine the results of all simulations and address any concerns, including all critical path and timing analyses, and all test-related issues, such as design-for-test and fault coverage. We check all details of the engineering documentation being released. The procedures to follow for this review are found in the local site Design Review Policy.

We evaluate the estimated module recurring engineering costs against cost goals and report actual productivity data at the review.

Hardware Fabrication, Assembly, and Test Phase

Engineering fabricates the first-item module, inserts components, and tests the module.

- Fabricate First-Piece Module — Manufacturing or an outside supplier fabricates the first-piece module.
- Assemble First-Piece Module — We assemble the first module by inserting components in this step.
- First-Piece Module Engineering Test — During this step, we test the first module using the module test equipment according to the test procedure document. We perform design verification in which we check timing, performance under variations in voltage and temperature, and other design performance criteria. Design verification determines if the design will meet environmental requirements. If any problems with the design are discovered, the module design engineer makes any design changes needed to correct them.

  We test and characterize the module on an automatic tester. All tests are performed over the full voltage and temperature ranges. Module characterization measures all electrical parameters. We analyze the data to determine module quality and capability.

  The ATE test vectors used in the functional and parametric tests are finalized to incorporate any adjustments needed to satisfy module, adapter, and tester interfacing anomalies. We analyze module test results to determine the integrity of the module-to-tester interfacing.
- Update Documentation and Release Package — The first-piece module engineering test may reveal deficiencies in the design that need to be changed. If this is the case, then a documentation and Release Package update is needed.

- Conduct Final Design Review — The FDR is the last design review of the completed engineering drawing package for the module. We check all details of the engineering documentation being released. This review is crucial, because any errors/discrepancies in the release package that are not found will affect production of the final system. There may be a large number of this type of module in a system, so the effects of an incorrect release package can be great.

- Engineering Signoff — After the FDR, engineering signs off the drawings for release to the drawing system. The module design cycle is now complete.

3.3.2.2 ASIC Design Process

This process, similar to module design, has three major phases. In the preliminary design phase, we generate a detailed specification for the device based on the architecture design effort. In this phase we develop, simulate, and verify a behavioral model, and develop a gate-level design (manually or via synthesis) and functionally-verify of the design via simulation. In the second phase, we translate the gate-level design to the supplier format, and perform all supplier-related simulations and pre-release procedures.

In the third phase, we fabricate and test the chip with the ASIC supplier. The ASIC design process is detailed in Figure 3-30.

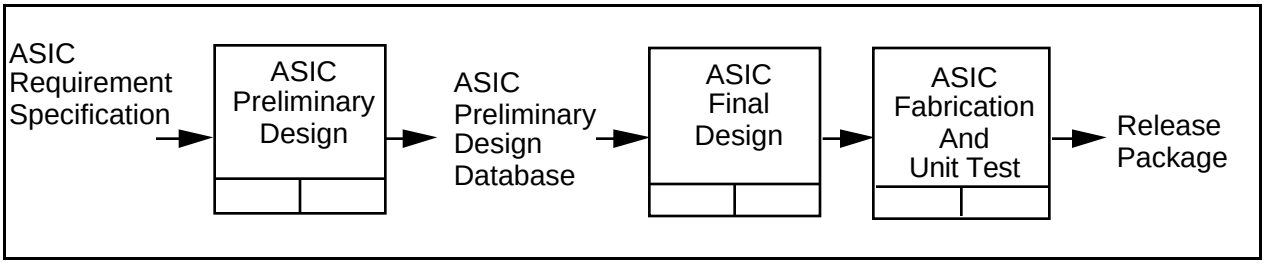


Figure 3-30. ASIC design process flow.

Critical to success of the ASIC design process is generating an accurate specification, following DFT practices, performing multi-level simulations, and conducting thorough design reviews.

ASIC Preliminary Design Phase

We start the ASIC design process with preliminary analysis and review of the ASIC Requirements Document compiled by the architecture and module designers. We search the RASSP design reuse library concurrently with preliminary analysis after reviewing the ASIC Requirements Document. If the library search reveals that no existing ASIC meets the requirements, we assess the feasibility of implementing the defined ASIC, and develop the overall architecture of the ASIC. This includes defining and sizing a high-level functional architecture, defining the I/O interface signals, developing the instruction set architecture (when applicable), and defining special test features. Technological, packaging, and cost constraints are considered.

We may have to conduct tradeoffs with architecture and module designers following this analysis to achieve the desired ASIC requirements. Manufacturing and mechanical engineering should be consulted to ensure the proposed design can be produced within project, technology, and cost constraints. Reliability engineering should be involved, along with the supplier, to ensure

that the fabrication process and packaging meets specified reliability requirements. The test engineer can provide information on the IC tester requirements and limitations.

- Search RASSP Design Reuse Library — We search the RASSP design reuse library concurrently with preliminary analysis/design to determine if an existing (legacy) design meets the requirements specified for the target ASIC. This step is most likely to be performed during the initial proposal phase of a design, when the subsystem designer attempts to configure the subsystem and calculate the cost.

- Generate the ASIC Design Specification — We generate an ASIC Design Specification that specifies the functional, timing, testability, and testing requirements for each unique microcircuit. This specification should combine the ASIC requirements from the subsystem/module design with the results of preliminary analysis and design to produce a complete description of the ASIC for use in behavioral and gate-level design.

- Develop ASIC Behavioral Model — All ASICs are an integral part of a larger system and perform some unique subsystem function. In some cases, the architecture designer modeled the unique subsystems using a high-level hardware description language to gain confidence that all functional blocks will interact properly. In these cases, each subsystem partition (i.e., the chip definition) should have had a high-level model developed that can be used by the ASIC designer as the starting point for behavioral model development.

  When developing the behavioral model, the ASIC designer should reflect the inherent structure of the design. If, for example, the ASIC has a pipelined architecture, the pipelines and the functions performed by each pipeline stage should be evident from the model. The purpose is to produce a model with sufficient detail to guide gate-level design.

  In this step, we also develop the 'tester' model, which applies input vectors to the model and collects and analyzes the results. This may be a simple model written in the same language as the behavioral model, or a complex suite of programs integrated into a complete simulation environment.

- Generate ASIC Test Plan — The ASIC designer prepares a detailed test plan that has two functions. First, it details the step-by-step procedure to verify the functional operation of the behavioral model. This includes the function(s) to be tested, the method and order of testing, and the data to be used. Second, it details the procedures to test the final ASIC devices on an IC tester. This includes all parameters to be tested, the test conditions, the vectors to be used, and the test limits. Timing diagrams for all inputs are included in a format compatible with the target tester. The test vectors are developed during chip simulation and provided to the test engineer along with the Test Plan document.

- Generate ASIC Test Vectors — We usually perform this step concurrently with the behavioral model functional simulation. Using the test plan as a guide, the ASIC design engineer generates vectors to test the functionality of the behavioral model. These vectors are used later in the gate-level and fault simulations.

- Perform Behavioral-Level Functional Simulation — We simulate the behavioral model of the chip to verify that its performance meets the architecture and module requirements. This effort may be performed jointly by the module design engineer and the ASIC design engineer to ensure full coverage of all architecture and module requirements. It is quite common for the module designer to discover design flaws in the module simulation that are difficult to detect in a single-chip simulation. This may include the detailed interaction of the ASIC during module reset or instruction-abort procedures. By providing the module designer with the latest version of the behavioral model, potential errors can be discovered before module integration and test.

- Update ASIC Design Specification — We update the ASIC Specification as required to reflect the results of the behavioral simulations. In particular, we add the final verified version of the behavioral model to the ASIC Design Specification. In addition, we update any other sections of the ASIC Design Specification that have changed, including the functional block diagram, the instruction set architecture, and other

details that may change during the model development.

- Perform Interactive Logic Design/Synthesis — The ASIC design engineer performs the detailed logic design, either by capturing the logic interactively or by using logic optimization and synthesis tools that take VHDL as an input. Logic design usually follows a bottom-up approach, where we develop and verify simple logic functions, and create successively larger functional blocks from these smaller blocks.

- Perform Testability Analysis and Generate Additional Test Vectors — In parallel with the interactive logic design, the ASIC design engineer performs a testability analysis on the preliminary netlist. The purpose of this step is to identify sections of the gate-level design that, because of controllability or observability constraints, may be difficult to test. The ASIC designer may then wish to develop dedicated test logic for these problem areas.

  If additional logic is developed in this step, the ASIC designer updates the behavioral model to reflect the new functions, and develops additional test vectors to test the new logic. These vectors, combined with the behavioral vectors, form the nucleus of the final set of test vectors for the chip, and fully exercise all functions of the chip. The updated behavioral model is sent to the module engineer for module simulation.

- Perform Gate-Level Functional Simulation — Once the gate level design is complete, we must verify the accuracy of the resulting netlist. The first step is to perform a gate-level functional simulation. This is similar to the behavioral model functional simulation performed previously, except that the gate-level netlist rather than the VHDL model forms the simulation database. The functional simulation is not concerned with the timing of the gate-level design. The purpose is to verify the functional correctness of the logic.

  We use a simulator that has a mixed-level capability that allows VHDL and gate-level designs to be simulated in the same design simultaneously. Because of this, it will probably be common to perform this step in parallel with the interactive logic design and synthesis. We can translate a subset of the entire VHDL model to a gate-level netlist and verify it using the test vectors developed during behavioral model functional simulation. Once this function is verified, we can translate the next function and verify it until the entire design is complete. The final gate-level design produces the same results as the behavioral model for a given set of test vectors.

- Compare Behavioral Versus Gate-Level Simulation Results — The ASIC design engineer compares the output of the detailed gate-level simulation with the previously-verified behavioral simulation to determine the functional correctness of the detailed design. The ASIC design engineer must isolate and correct discrepancies between the two models until both simulations produce the same results.

- Perform Gate-Level Timing Analysis — The ASIC designer performs a timing analysis simulation on the chip design, or portions of it. The designer compares the results of the analysis with the requirements imposed by the device specifications. There are two ways to assess the performance of the electrical design. The first method is to use a critical path analysis tool to identify critical paths. The second method is to put timing information in the gate-level models and perform functional simulation. This can identify setup, hold, and other timing violations.

  In this step, we assume accurate timing information is available from the supplier, including loading and wire length estimates. In addition, many suppliers have special macrocells or megafunctions as part of their cell libraries. To perform accurate timing analysis at this stage in the design cycle, the timing analyzer must have access to the timing information for any supplier specific functions. If this is not the case, it may be necessary to delay this step until after the design has been translated to the supplier format.

  Since the timing results from this stage are only estimates, the designer should not attempt to squeeze the last nanosecond out of the chip performance in this step. The purpose is to identify obvious problem areas, and make the changes necessary to get the performance as close to the requirements

as possible. The detailed timing analysis will be done on the supplier toolset.

There are usually be two types of changes required to increase performance. The first type is fine-tuning the gate-level design, such as replacing a carry look ahead adder with a carry select adder. We can make this type of change to the netlist only, since the basic structure and functionality are not affected. The second type of change is a major architectural restructuring, such as adding a pipeline stage. We make this type of change to both the netlist and the behavioral model. In both cases, we rerun the functional simulation to verify that the changes did not affect the functional operation of the design.

- Perform Fault Grading/Fault Simulation — The ASIC designer performs a fault analysis on the functional test vector set, and generates additional vectors until the final set of test vectors provides some predefined fault coverage. As in module design, fault simulation serves two purposes. The first is to provide some quantitative measure of the quality of the test vector set. The second is to obtain a set of vectors that detect flaws in the fabricated device.

- Conduct Preliminary Design Review — After completing the ASIC design, and before committing the design to the supplier for pre-layout simulation and signoff, we conduct a PDR to provide a detailed critical technical evaluation of the ASIC design and to resolve any outstanding issues. All disciplines that interact with module design should have representatives present at this review.

- Translate Design Database to Supplier Format — Most suppliers require the netlist and test vector set in a particular format. We can translate at either the designer's or the supplier's facility. The ASIC designer must also generate any other data about the chip required by the supplier.

- Perform Preliminary Layout (Floorplanning) — Many suppliers allow the designer to perform a preliminary layout or floorplan at this stage. This allows the designer to place major functional blocks of logic on the die, and to assign the ASIC I/O signals to the physical pins of the ASIC package. This step serves two main functions. First, it provides the supplier with a routability analysis of the ASIC. This determines how easy or difficult it will be to perform detailed layout. Most suppliers require a certain routability number before they accept the design for layout. The ASIC designer may have to delete gates or use the next larger die size if this number cannot be reached. Second, it provides updated delay results based on estimated wire lengths from the placement of the functional blocks of logic and the location of the I/O pads. This new delay information provides more precise estimates of the chip performance.

- ASIC Supplier Simulations and Other Supplier Specific Steps — All suppliers require a set of simulations and procedures to be completed before accepting a design for layout. At a minimum, this includes a functional simulation, and timing analysis using the estimated wire length data from the floorplanning performed in the previous step. The functional simulation verifies the functional operation of the design by comparing the supplier simulation results with those from the simulator being used by the designer. The timing analysis includes critical path analysis and setup and hold-time violations for minimum/maximum conditions. In addition, the I/O delays can be calculated with various external loading parameters.

  The exact set of requirements varies according to the supplier, but the following additional simulations are commonly required. Design verification simulation checks the design for proper design rule use. A test file is usually produced that converts the test vector set into a format compatible with the supplier's test facility. A toggle test verifies that the vector set toggles every net in the design at least once. The parametric simulation is used to verify proper voltage levels and/or drive current in the I/O signals.

- Compare Supplier Simulation Results to Contractor Results — We perform this step to correlate the supplier's simulation results with our simulation results to verify functional equivalence across the two sets. Discrepancies should be isolated and corrected so that the two simulation environments produce the same results.

- Conduct Pre-Release Design Review — We review the final design to verify that all subsystem and

module functional and performance requirements will be met. We examine the results of the supplier simulations and discuss any concerns. This includes all performance-related issues, including special critical path routing requirements, minimum/maximum timing, and special layout concerns such as the clock tree. The Module Designer examines the final bonding diagram, and the supplier certifies that all pre-layout requirements have been met.

- Prepare ASIC Release Package — The ASIC designer prepares a Release Package containing all the information necessary for the supplier to fabricate, assemble, test, and deliver the ASIC devices. A minimum set of requirements for this document should include a netlist, test vector set, bonding diagram, packaging information, quality level of the ASIC (MIL, Commercial, etc.), performance requirements, and I/O drive capability.

ASIC Final Design Phase

- Perform Detailed Layout — We perform the detailed layout at the supplier's design center or fabrication site. Using the information from the netlist, the floorplanning, and the bonding diagram, we place and route the detailed netlist using an automatic layout tool. This process does not actually result in the physical device, but in a description of the final ASIC routing mask. This provides detailed, accurate data on the lengths of every net in the design to be used in the next step.

- Perform Post Layout Simulation — Using the detailed wire length data from layout, the ASIC designer reruns the functional and timing analysis simulations to verify that all performance requirements are met. This step is basically a repeat of the ASIC supplier simulations performed previously; the only difference is that we replace the estimated wire length parasitics with the actual layout parasitic data.

  If the performance does not meet the requirements, the designer makes changes and sends the new netlist back to the supplier for layout. This process continues until the performance is acceptable.

ASIC Fabrication and Unit Test Phase

- Fabricate, Assemble, and Test ASIC Device(s) — The ASIC supplier fabricates, assembles, tests, and delivers the specified number of prototype devices.

- Perform Final In-House Test — The ASIC test engineer, using an in-house IC tester, performs final test and characterization procedures on the prototype devices according to the ASIC Test Plan. The vectors used for this step should be the same as those supplied to the supplier.

3.3.2.3 FPGA Design Process

This section discusses the Field Programmable Gate Array design process which is becoming increasingly popular among designers of Digital Signal Processing hardware. This process has not been “re-invented” by the RASSP program, but rather reflects current industry standard methods and COTS tools.

The FPGA process described here is kept generic intentionally and may be applied to various classes of programmable logic devices including Field Programmable Gate Arrays, Complex Programmable Logic Devices (CPLDs), and Erasable Programmable Logic Devices (EPLDs). The inputs to the FPGA process are a Requirements Specification, which usually is a part of a higher-level module (board) Requirements Specification and a VHDL behavioral model of the FPGA function. Major outputs from the process are a fusemap used to program the device and test vectors used to verify the functionality of the design. The design can be divided into three phases: Preliminary Design, Final Design, and Programming and Unit Test. These phases depicted in Figure 3-31 are described in the following paragraphs.

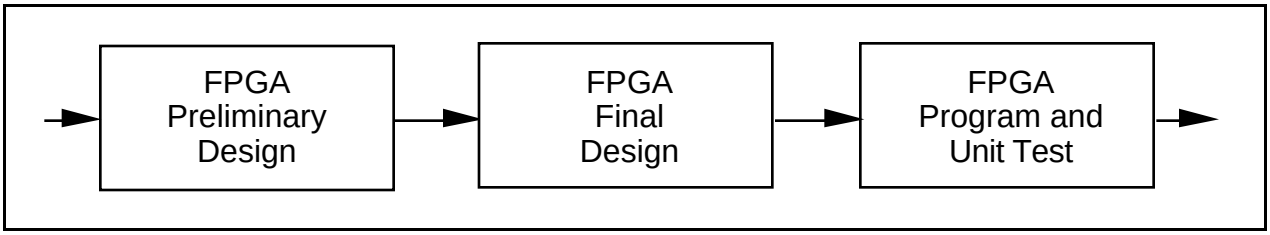


Figure 3-31. Phases of FPGA design.

FPGA Preliminary Design

Functional Design

The purpose of this step is to refine an existing VHDL behavioral model of the FPGA function in sufficient detail so that it can be synthesized down to the specific programmable function unit level. The VHDL behavioral model is generally provided by the architecture definition process or by the module/MCM design process. The FPGA requirements document should contain the behavioral model, any requirements derived from module design, such as real estate restrictions, timing requirements, power restrictions, or any other module-related information.

Three styles of design entry using VHDL are possible: behavioral, Boolean, and structural. Tabular descriptions are also possible. A complex VHDL design can be built by successively combining building blocks in related layers. Up to this point the designer has not implied any particular hardware. The assumption is that he has captured the scope of the design into one abstract level VHDL description. This description may be realizable in a broad range of hardware devices. Memory, interface, microprocessor, one or more programmable logic devices, and a variety of other integrated circuits may be required to realize the design. Therefore, the design must be decomposed into those specific devices which the designer chooses for realization.

Using a CAE tool the designer can explore the variety of device implementations by interactively varying design criteria - including cost, preferred parts, device count, speed, and power - to find an optimum design solution.

Assuming that partitioning has been performed and a particular FPGA has been selected for a certain function, further decomposition tasks must be performed. Clocks may have to be connected to a clock distribution network and high fanout inputs must be connected to high drive input cells, if desired.

Functional Verification

During this step the refined FPGA behavioral VHDL model is verified to ensure that it meets functional and performance requirements. This is accomplished by performing a functional simulation using the VHDL FPGA model. The Testbench used for this verification should be the set of test vectors passed down from module design or architecture design, usually augmented by the FPGA designer. Outputs from the simulation are compared to the expected values
passed down from the module designer or architecture design.

FPGA Final Design

Logic Design / Synthesis

After functional verification of the behavioral model the FPGA is synthesized from the refined behavioral model. Synthesis is the transformation of the design input at a high level of abstraction to a lower level. The VHDL description is turned into an unoptimized Boolean level description of the design. This unoptimized Boolean description is flattened, that is, it has no hierarchy and all intermediate variables are removed. At this point the design is not device-specific and is in a sum-of-products form. Next, the synthesized design is transformed into a device-specific netlist. This netlist is passed to the place and route tool.

Placement and Routing

This step involves the actual placing and routing of the design into the actual device. The place and route software accepts the output of a technology mapper and generates a file for use by the programming equipment and a timing model for use by downstream analysis tools. The place and route determines the actual cell choice for each logic function to be implemented and the wire connections for the signals which pass and from the logic cells and I/O cells. Once the place and route is completed, a performance analysis is usually done to verify that the design will meet expected performance.

Functional and Timing Simulation

After place and route of the target design, it must be resimulated with backannotated delays to verify that it still functions as expected. A full timing gate-level simulation model is assembled using the backannotation delay information. Test vectors generated for verifying the behavioral VHDL model are used to exercise the model. The resulting post-route simulation outputs are compared against the simulation results generated during the functional model verification. Once validated, the full-timing gate-level model can be made available for use in module-level simulations.

Fault Simulation

A fault simulation can be optionally performed on the FPGA gate-level design to detect stuck-at-one, stuck-at-zero fault conditions. The procedures for this fault simulation are the same as those for the ASIC design process.

Refine Test Plans/Procedures

The detailed test plan and test procedures for the FPGA are refined as necessary. The plan describes the method and the procedures define the step-by-step the instructions to be used to test the device standalone, before it is inserted onto a module.

Conduct Design Review

This design review is optional, and usually is made part of the module design reviews. This review provides a detailed technical evaluation of the FPGA design function and performance, and serves as the focal point for identifying any outstanding issues. The design review is conducted in accordance with local policies and procedures. A detailed design package is generated by the designer for inclusion in a module-level design review.

Program Device and Unit Test

After the FPGA or module design review, the devices can be programmed using JEDEC files that are downloaded to the device programmer. Then the individual devices can be unit tested using the same set of test vectors and expected outputs that were used during functional, timing, and fault simulations.

3.3.2.4 Backplane Design Process

The architecture design activity provides the basic requirements for backplane functions. The purpose of the backplane design process is to determine how to interconnect the modules to perform those functions.

There are several terms which are used interchangeably when describing backplanes; these terms include “backpanel” and “motherboard”. Although “backplane” and “motherboard” are sometimes used to differentiate between wire wrap and printed wiring, respectively, there is no industry wide standard for these terms. This document will refer to “backplanes” only.

Commercial or Industry Standard backplanes, such as VMEbus or Multibus II, are backplanes which have been manufactured to a specification or standard, with some or all pins predefined, and are available off the shelf.

The backplane design process described here applies to the design of custom backplanes of varying complexities with wirewrap and/or PWB interconnections. It may also encompass the design of backplanes that use off-the-shelf or modified, off-the-shelf backplanes as the basis for a new design. The backplane design process flow is detailed in Figure 3-32.

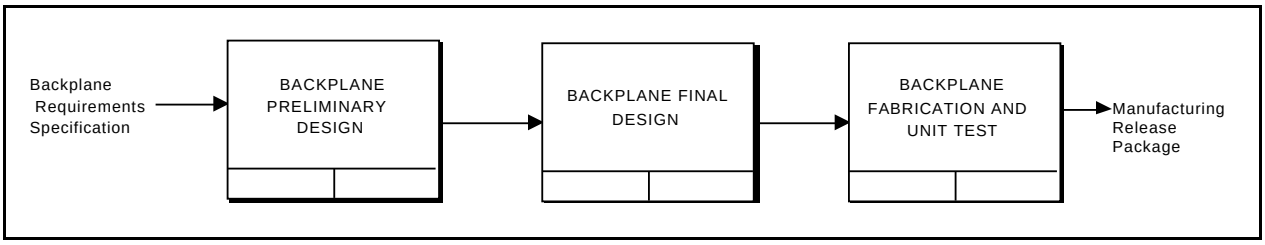


Figure 3-32. Backplane design process flow.

Backplane Preliminary Design

We begin the backplane design process by forming a design concept, which is then documented in a Backplane Specification that describes how the backplane design engineer plans to implement the required functions. If the design concept calls for any new or modified modules, the backplane design engineer provides mechanical engineering with preliminary mechanical requirements at this stage. Backplane designs should, however, use previously-designed backplanes to the greatest extent possible.

Partitioning can significantly affect the size and complexity of a backplane. Poor partitioning can increase the number of layers and thus the cost of the backplane, while good partitioning can efficiently use the available space and minimize the number of layers or wiring. Poor partitioning can also lead to noise problems due to long signal runs or wire lengths.

A commercial/industry standard backplane fits the COTS philosophy. It requires no manufacturing time and minimal lead time, and a standard backplane can be chosen so that interface problems in a system can be minimized. It does, however, limit the user to the system functions that are provided in the standard.

Commercial or Industry Standard backplanes have many pins predesignated, and some that are user-defined. It is important for the designer to understand which of the pins are predefined by the standard, and which are left for the backplane designer to designate; obviously, the designer should not use any of the standard-defined pins for uses other than what the standard intended.

- Perform Backplane Detailed Electrical Design — We complete this step when the backplane design engineer determines the partitioning of printed circuit interconnects and wirewrap interconnects on the backplane and establishes a power distribution approach. With this information in hand, the engineer uses a workstation to interactively perform the preliminary partitioning of the backplane schematic design. It is at this stage that the final details of the schematic design must be captured. Signals must be analyzed to determine the signals to be interconnected by PCB and the signals interconnected by wirewrap. Module preliminary placement should consider such things as estimated power dissipation by the modules, voltage requirements, and interconnectivity.

  Detailed design work proceeds at this point, based on the design concept. This task includes input/output definition and the design of timing/control functions.

- Develop Backplane Test Plan and Test Procedures — We generate a detailed test plan and procedure for functional and parametric testing. The plan should contain strategies for all test types, parameters, methods, conditions, and special resources required. Evaluation procedures should include critical signals to be tested and characterized, ands signals to be checked for cross-talk, reflections, etc.

- Perform Preliminary Electrical Analysis — Preliminary electrical analysis consists of evaluating and simulating critical signals on the backplane based on loading characteristics, worst-case signal lengths, and signal interaction. This step helps determine module placement, drive characteristics, and interface requirements by allowing early characterization of critical signals.

- Generate PWB/Wirewrap Netlists — In this step, we extract preliminary PWB and wirewrap netlist data from the schematic design database created during backplane interactive logic design. The objective is to extract all design data from the schematic design database and to generate an ASCII file representing the design.

- Conduct Backplane Preliminary Design Review — After completing the detailed design of the backplane, and before committing the design to PWB layout or wirewrap wiring, we conduct a Preliminary Design Review. Participants include the project manager, the subsystem design engineer, the local Design Review Board, and the backplane design engineer. In addition, all disciplines that interact with backplane design should have representatives present at the review.

  The purpose of this review is to ensure that the functional requirements of the backplane specification have been met. It will provide a detailed critical technical evaluation of the functional electrical design, and we will resolve any outstanding issues.

Backplane Final Design

If the backplane design includes a PWB section, we perform the basic placement and routing steps as in module design. We perform a detailed electrical analysis to verify that proper design rules were followed in devising the interconnection scheme. We then generate artwork and tools for the PWBs, performed as in module design.

- Design Printed Wiring Board — Designing the PWB portion of the backplane includes optimizing placement of the module connectors and the interconnecting printed wires. The complexity of this dictates whether we apply rules during place and route or during the analysis process after routing. The activities are a joint effort between the backplane design engineer, the PWB layout specialist, and the subsystem engineer.

- Perform Electrical Analysis of PWB — Effective backplane system design requires balancing signal density against electrical performance, cost, mechanical reliability, and producibility. As the clock frequency increases and signal rise times decrease, backplane interconnections become more complex and can no longer be considered as ‘short circuits.’ A basic understanding of transmission lines is vital to analyze and improve these interconnections.

- Generate Artwork and Manufacturing Tools for PWB Fabrication — We generate the artwork and manufacturing tools necessary for fabricating the PWB.

- Perform Wire Wrap Place, Route, and Electrical Analysis — We perform this step to convert the wirewrap netlist into a connection list without routing and levels. The primary function of this processing is to break up the list of pins in each net into a discrete set of wires. We perform circuit analysis to verify that the resulting circuits are functionally correct, i.e., each net contains one output, clock wire length is not less than the longest wire, etc.

  Once we have processed the wirewrap netlist, we route and level the wires using the user-specified rules for optimizing parallelism, line reflections, and wiring density. Wire routing entails searching for the optimum routing path while considering the electrical interaction of wires that were previously routed. For wires that fail the allowed criteria for parallelism, the design engineer can automatically convert the single wires to grounded twisted pair, or manually designate a different route for the wire.

  After we complete routing and leveling for each wire, the wire connection list contains the necessary information to generate manufacturing tools.

- Conduct Backplane Pre-Release Design Review — The Pre-Release Design Review is the final design review of the completed backplane Release Package. All details of the engineering documentation being released are checked. The review is crucial, since any errors or discrepancies in the release package that are not found affect the fabrication and assembly of the first-piece backplane.

Fabricate/Assemble/Unit Test Phase

- Fabricate and Assemble First-Piece Backplane — This step is a manufacturing function, not an engineering function.

  Once the tools are generated, manufacturing uses the NC data and reports to assemble the backplane. Manufacturing either purchases or manufactures the bare board, and mounts the pins or connectors on the board. An objective of this step is to add the wire-wrapped wires to the backplane and, optionally, to test the wires as they are added to ensure that the proper connections were made. Depending on the design requirements, we test the backplane for continuity and isolation at various stages of assembly. It may be tested before wire installation, after initial hand wires and automatic wires are added, after all the wires are installed, or at all of those stages. Following the completion of this step, we test and debug the first-piece backplane.

- Perform First-Piece Backplane Engineering Evaluation — During this step, we test the first backplane using lab test equipment according to the Test Procedure document. Design verification is also performed, in which we evaluate the backplane for crosstalk, line reflection characteristics, the effectiveness of the power distribution system, settling time, and signal overshoot/undershoot.

- Update Documentation and Backplane Review Package — This step is used when a design engineer needs to change either the backplane PWB design, the wiring design, or any ancillary documents, such as the Test Plan, descriptive documents, etc. We perform it after those documents are released into a controlled system and it is usually done during and after the first-piece test and debug, or any time thereafter.

- Conduct Final Design Review — After completing the backplane design and evaluation, we conduct a review to provide a detailed critical technical evaluation of the backplane design and to resolve any outstanding issues. All disciplines that interact with backplane or module design should have representatives present at the review.

3.3.2.5 Chassis Design Process

We define a chassis as the top-level hardware assembly containing the electronics and mechanical components of a subsystem such as a digital signal processor. A subsystem could be made up of more than a single chassis. Across the industry there is no standard terminology in this area. Other terms used to describe this hardware are “subassembly”, “frame”, and “box”. A chassis usually contains at least one “backplane” which provides the interconnection for enclosed electronic printed wiring boards (also designated “modules” or “circuit card assemblies”) as described in the following sections. The chassis design process can be broken up into two major phases which we designate Preliminary design and Final design. These processes are described in the following paragraphs and depicted in Figure 3-33.

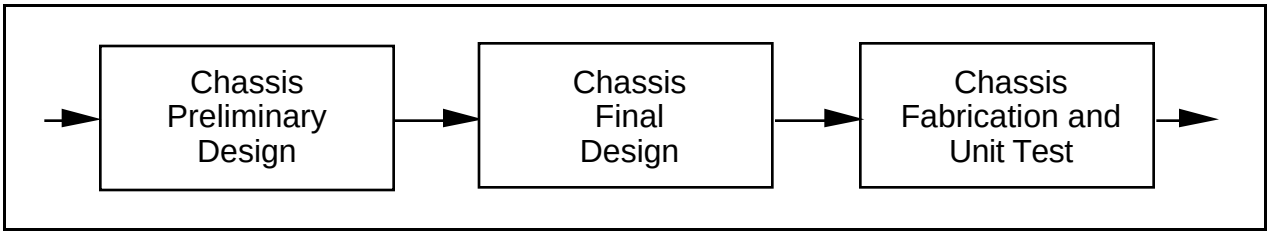


Figure 3-33. Phases of Chassis design.

Prior to chassis Preliminary design the mechanical and electrical designers review the proposal concept solution against the latest revised requirements. A new look is taken at the previous design and sub-contractors or co-contractors are consulted to explore the advantages of common approaches to packaging design and to determine that requirements that affect packaging have been met.

Chassis Preliminary Design

- Select Parts, Materials and Processes - The mechanical engineer prepares a list of the chassis parts, materials, and manufacturing processes required for the design, and notifies purchasing of any long-lead items needed. The source used for selection is the RASSP Reuse Database. If any new process is required, a process readiness activity must be successfully completed. This step is necessary to ensure that project schedules are not jeopardized by long-lead ordering cycles and/or development problems associated with the introduction of new processes.

- Perform Mechanical Design Analysis - Analyses of weight, stress, tolerances, and thermal distribution are conducted using appropriate models of the chassis.

- Establish Test Requirements - The mechanical, electrical, and manufacturing engineers determine the electrical and environmental test requirements and design or procure the needed test fixtures. Test plan documents are generated.

- Document Structure/Thermal Interface - At this time enough is known about the contents of the chassis to commit to an outline and structural mounting and thermal interfaces. For the design of the chassis to proceed without delay, we must provide firm information on maximum dimensions and dissipations to structural designers for the next higher assembly, if one exists.

- Document Harness Interface - At this point in the design cycle, we have sufficient data on signal flow and electrical inputs and outputs to establish connector types, quantities and locations, and possibly complete definition of the harness interface. For the harness design to proceed in a timely fashion, we must commit to connector locations and types as early in the chassis design cycle as possible.

- Determine Assembly Processes - Working with manufacturing producibility engineers, the mechanical engineer decides on the sequence of events in the assembly of the hardware, and prepares to document the detailed design. The final drawing package defines only the finished configuration of the hardware, but when properly documented, can indicate an assembly sequence that eliminates synthetic subassemblies which create added paperwork for the manufacturing planners.

- Conduct Preliminary Design Review - The electrical and mechanical engineers participate in this review. Peers, technologists, and responsible engineers from other disciplines review the design and make constructive suggestions for improvement. Others (producibility, sourcing, quality, reliability, etc.) participate as required to ensure cross functional support for the design.

Items reviewed include layouts and solid models of major items, test plans, structural, thermal, tolerance, and weight analyses, source selections, updates to conceptual/proposal data, and interconnection layouts. We conduct this review in accordance with the local policy and procedure guidelines for internal design reviews, and following standard checklists. This step confirms that specifications can be met within established cost objectives, and should reconfirm the acceptability of designs and approaches generated in architecture design.

Chassis Final Design

The final design phase encompasses the detailed level mechanical design of the chassis. Detail design includes the interconnection cables and harnesses. These items require complete definition including, but not limited to: all dimensions and tolerances, material and finishes, marking requirements, test or acceptance criteria, etc. In this step, we begin the formal documentation of the chassis, the end product of which are drawings and parts lists necessary for a supplier or manufacturing to fabricate prototype or

production hardware.

- Perform Detailed Design of Chassis - Solid model layouts are finalized, based on the latest changes and models are reduced to the detail part level. The design is driven by stress, thermal, and survivability considerations, with the goal of optimizing weight and producibility. This step is essential to the proper functioning of the electronics during and after exposure to environments. The chassis must provide protection to the sensitive electrical parts and interconnections from the effects of shock, vibration, internal and external heating, radiation, and magnetic effects. Electrical and mechanical test procedures are also generated during this phase.

- Perform Analysis of Chassis - In this step we begin verifying the detailed design. We again perform thermal, stress, tolerance, and weight analyses and compare the results with subsystem and environmental requirements, to ensure that we established adequate margins to provide proper performance, even under worst-case conditions. Cable and harness designs are also finalized.

- Perform Top Assembly Detailed Design - This step integrates the chassis design with the elements of the parallel detail design activities (backplane and module) which are discussed in later sections of this document, into the assembly which is the end product. This step pulls together the lower level detail designs and verifies that they will work together mechanically to meet the intended assembly function. Potential interferences are checked, dynamic clearances determined, and access for tooling, wiring, and assembly processes is established so that manufacturing can proceed smoothly and test problems will be minimized.

  The top-level assembly is subjected to thermal, stress, EMI, shielding, dynamic, venting, weight, and other special analyses to assure that adequate margins have established to provide proper performance, even under worst case conditions.

- Prepare Release Package for Chassis - Next, the required documentation or release to manufacturing or to an external supplier is generated. Solid models of parts are converted to drawings. Detail drawings, assembly drawings, and parts lists are generated, in accordance with design and manufacturing standards and local documentation practices.

- Conduct Pre-Release Design Review - Electrical and mechanical designers (among many others) participate in this review which, if successfully completed, will affirm the readiness of the design for production. The review is conducted in accordance with local policy and procedures for formal internal reviews.

Chassis Fabrication, Assembly, and Unit Test

The bare chassis is manufactured and assembled locally or by an outside supplier. In either case, electrical and mechanical testing is performed according to the test plan and procedures developed previously.

3.3.2.6 Subsystem Integration and Test Process

In the overall RASSP design process, the subsystem (e.g., a signal processor) is usually the highest level of integration proposed. Subsystem integration and test is the last process the equipment undergoes. This process requires significant up-front planning in the form of integration plans, test plans, and test procedures long before the equipment is ready for actual integration and testing. Schedules for the availability of the tested subassemblies from the other design processes and manufacturing have to be negotiated. The subsystem integration and test process flow is detailed in Figure 3-34.

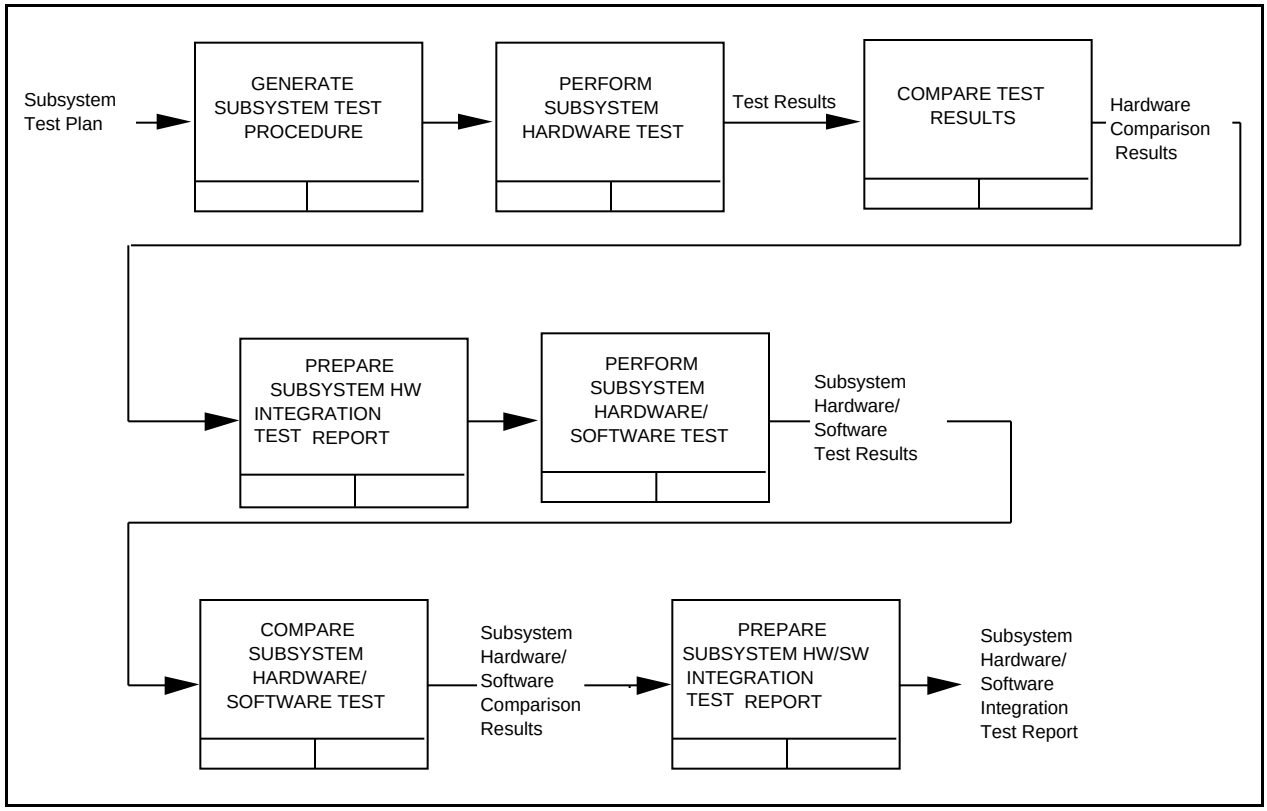


Figure 3-34. Subsystem integration and test process flow.

Generate Subsystem Integration Plan

In the first step in the subsystem integration and test process, we generate a subsystem integration plan. This plan lays out the events that will occur, and their order and timing, to achieve an integrated and tested subsystem within the cost and schedule allocated to this process. The integration plan defines the interactions and delivery schedules negotiated with other design activities and manufacturing. It also defines the state in which equipment will be delivered to this process.

Generate Subsystem Test Plan

In this step we generate a plan to test the integrated subsystem. This plan specifies all tests that must be performed to verify that the subsystem satisfies the B2 specification. The plan is driven by Section 4 (Quality Assurance) of the specification, which identifies the method (analysis, test, inspection, etc.) by which each of the requirements is verified. For each of these verification procedures, the test plan specifies the inputs, conditions of test, outputs, number of samples, etc., in qualitative terms. It also identifies the test equipment required and how it will be used. Block diagrams of equipment configurations are useful to clarify their intended use.

Generate Multichassis Test Plan

In large subsystems that have more than one chassis, an intermediate integration and test step between chassis test and full subsystem test is useful. This step develops the test plan for this multichassis test. There may be more than one of these multichassis tests. The goal is to integrate as few chassis as possible to make a subset of the subsystem that performs a testable function.

We use the subsystem test plan as a starting point. For each test identified at the subsystem level, we attempt to define a similar test at the multichassis level, although this will not always be possible. As in the subsystem test plan, the multichassis test plans identify, in qualitative terms, the inputs, conditions of test, outputs, number of samples, etc., for each test. They also identify the test equipment required and how it will be used. Block diagrams of equipment configurations are useful to clarify their intended use.

Generate Chassis Test Plan

In this step, we generate the test plan for the lowest level of integration that includes a testable unit. In most subsystems, this is the chassis level, although for some subsystems it is the backplane level. As in previous steps, the goal is to find and solve problems in the equipment as early in the test and integration process as possible. We generate a separate test plan for each unique chassis in the subsystem; again, these identify, in qualitative terms, the inputs, conditions of test, outputs, number of samples, etc., for each test. They also identify the test equipment required and how it will be used. Block diagrams of equipment configurations should be used to clarify their intended use.

Generate Test Procedures for Each Test Plan

In this step, we generate detailed test procedures from each individual test plan. Since these procedures are used for actual testing, they cover all testing configurations. However, test procedures do not normally specify the testing and debug approaches required to solve a problem when a test fails. The test plans and procedures should describe the purpose of the

test and how it functions in sufficient detail to make a debug approach apparent. The test procedures specify the test parameters outlined in the test plans. Expected outputs for each test and pass/fail criteria must be specified. Equipment figures and block diagrams from the test plans should be expanded and detail added. Forms should be included for each test which, when filled out, provide the basis for a test report.

Conduct Test Procedure Review

We review the test procedures to verify: (1) that the procedures accurately evaluate the subsystem's performance; (2) that the tests are done at as low a level in the design as possible; and (3) that the test procedures are complete. Incomplete test procedures have a significant cost impact on the testing floor. In addition to this internal review, many customers require approval of the subsystem level test procedures.

Integrate Backplanes and Test

This step is the first phase of actual integration and test. We integrate assembled and tested modules with the physical structure, backplane, and firmware into a testable chassis. We then use the test procedures developed for this configuration to test the equipment. In a large subsystem there may be several unique chassis tested during this step; each is tested according to its own unique test procedure. We use subsystem-level expected results generated in the architecture design process to verify satisfactory performance of the equipment. Following integration and test, we prepare a test report, and the tested equipment is then ready for the next level of integration. The next level of integration should not proceed until all tests have been completed and problems resolved.

Integrate Chassis and Test

This step is the second phase of actual integration and test. We integrate assembled and tested chassis with the physical multiframe structure into a testable multiframe configuration. In a large subsystem, there may be several unique multichassis configurations tested during this step; each is tested according to its own unique test procedure. In small subsystems there may not be a need for this level of integration and test. We use subsystem-level expected results generated in the architecture design process to verify satisfactory performance of the equipment. Following integration and test, we prepare a test report, and the tested equipment is then ready for the next level of integration. The next level of integration should not proceed until all tests have been completed and problems resolved.

Integrate Subsystem and Test

This step is the final phase of actual integration and test. We integrate assembled and tested multichassis configurations into the subsystem, which is then tested according to its test procedure. We use subsystem-level expected results generated in the subsystem design process to verify satisfactory performance of the equipment. Following integration and test, we prepare a test report, and the tested subsystem is then ready for further integration, if required.

3.3.3 Use of VHDL in the Hardware Design Process

The hardware design process transforms the architecture component VHDL models into detailed hardware designs. We decompose the abstract, non-evaluated architecture component models into full-functional VHDL models of their constituent entities. We use VHDL extensively

throughout this process. In addition to the full-functional models, we create bus-functional models where needed to efficiently test interface designs. The bus-functional models are timing and functionally correct with respect to interface functionality only, whereas the full-functional models are correct with respect to both interface and internal function and timing.

We develop VHDL structural models for the VHDL behavioral models that were developed during the architecture design process. These structural models show the partitioning and interconnection of architecture component modules. We develop a test bench and a VHDL behavioral model for each module, and each module is, in turn, further decomposed and partitioned into other modules, macro-cells, MCMs, ASICs, or COTS components. We continue this iterative process of decomposition and partitioning for custom components until the hardware functions can be described as RTL logic transformations within leaf-level VHDL behavioral models. We can then automatically synthesize the leaf-level models into gate-level netlists. The VHDL models of programmable units are designed to interpret the data files produced by the software design process to facilitate hardware/software co-simulation and codesign. During the partitioning process, we can identify some of the modules, MCMs, and ASICs as custom parts, while others are selected from COTS parts. For custom-developed parts, we develop VHDL models down to the ASIC level, with a fully functional behavior and bus-functional model describing each ASIC. For COTS parts, we develop or obtain only VHDL behavioral models.

The VHDL behavioral models describe the timing and functionality of the module, macro-cell, MCM, and/or ASIC. Timing data includes:

- Module, macro-cell, MCM, ASIC I/O
  - I/O timing constraints
  - I/O interface structures
  - I/O protocols
  - Signal strengths
  - Message types
- Module, macro-cell, MCM, ASIC processing latency
  - Data acceptance rate
- Module, macro-cell, MCM, ASIC stimuli response

Functionality data includes:

- Control strategies
- Task execution order
- Synchronization primitives
- Inter-process communication (IPC)

The test benches provide test procedures, stimuli, and expected results to verify that the design meets system requirements. The test benches are developed before, and are executed with, the behavioral models of the hardware designs during simulation. The design of test benches before component modeling supports the DFT methodology. The full-functional behavioral model forms the executable specification for a hardware design. We back-annotate the simulation results of the detailed hardware models to the higher level architecture models to make them precise.

3.3.4 Design For Test Tasks in Detailed Design

The focus of detailed design is synthesis and implementation of the selected architecture in hardware and software. From architecture verification, all boards have been broken down into behavioral blocks representing MCMs, ASICs, FPGAs or relatively small logic blocks (i.e. glue logic and/or functions to be implemented in PLDs). Major components such as processors, interconnects and sensor interfaces have been selected, specified and verified at least by simulation. Detailed design elaborates these designs into implementations by schematic capture, synthesis, place, route, autocode and primitive optimization activities.

Physical prototypes are designed, fabricated, tested and integrated with software per the program plan. The extent of the full system can range from a significant sub-assembly (i.e. a MCM or more typically a set of boards where each board type is present) to the complete system. Based upon these results the TSDs are updated with preliminary measurement data on fault population coverage. Design flaw data is collected based upon the extensive simulations of VP3 and the system prototype results. Preliminary results are collected on BIST coverage of manufacturing faults. If the program plan calls for a BIST demonstration, preliminary results are collected on BIST coverage of field support faults.

The generic test insertion process for each design entity is shown in Figure 3-35.

Testable COTS components should be used to the maximum extent practicable. In addition, the "Lead, follow, or get out of the way approach" to dealing with COTS is implemented. This involves adding, or using existing, DFT features to the design to deal with COTS. As illustrated in Figure 3-36, COTS devices possessing BIST features, such as the TI SN74BCT8244 (having a PRPG and PSA mode), are given a "lead" role, since they can lead a test process. COTS devices, such as the four non-boundary-scannable octal flip-flops are given the "follow role," since, while they cannot lead, they at least do not impede the BIST test flow originating at the boundary-scan octals. Finally, the RAM chips, being very complex, and having no test features, coupled with an inability to pass pseudorandom patterns through during RAM tests, must be relegated to the "get out of the way" role which means bypassing them during BIST mode using a output tri-stating approach.

Test domain analysis is performed to stretch the reuse of all DFT/BIST structures. An example of this concept is shown in Figure 3-37. In the example, the source of BIST stimuli (8244) is analyzed to determine how much of the board (and eventually the rest of the system) can be tested using the PRPG and the PSA elements in the 8244s. A minimum of four domains (1-4) are covered along with additional domains in the rest of the system. By adding loopback capabilities, fault isolation is improved; and the number of domains covered is doubled. Pseudorandom patterns emanate from the source 8244 and responses through the loopback are compressed in the parallel signature analyzer in the sink 8244. Test domain analysis can be used to stretch the limits of reuse across packaging levels, as well as across the same package level.

Checking for compliance for COTS sections requires definition of the fault model, such that the coverage is measurable, using non-structural simulation models for the black box COTS elements or by using simplified functional fault models.

See Section 3.2.5 of the DFT Methodology for further details of the Detailed Design Step.

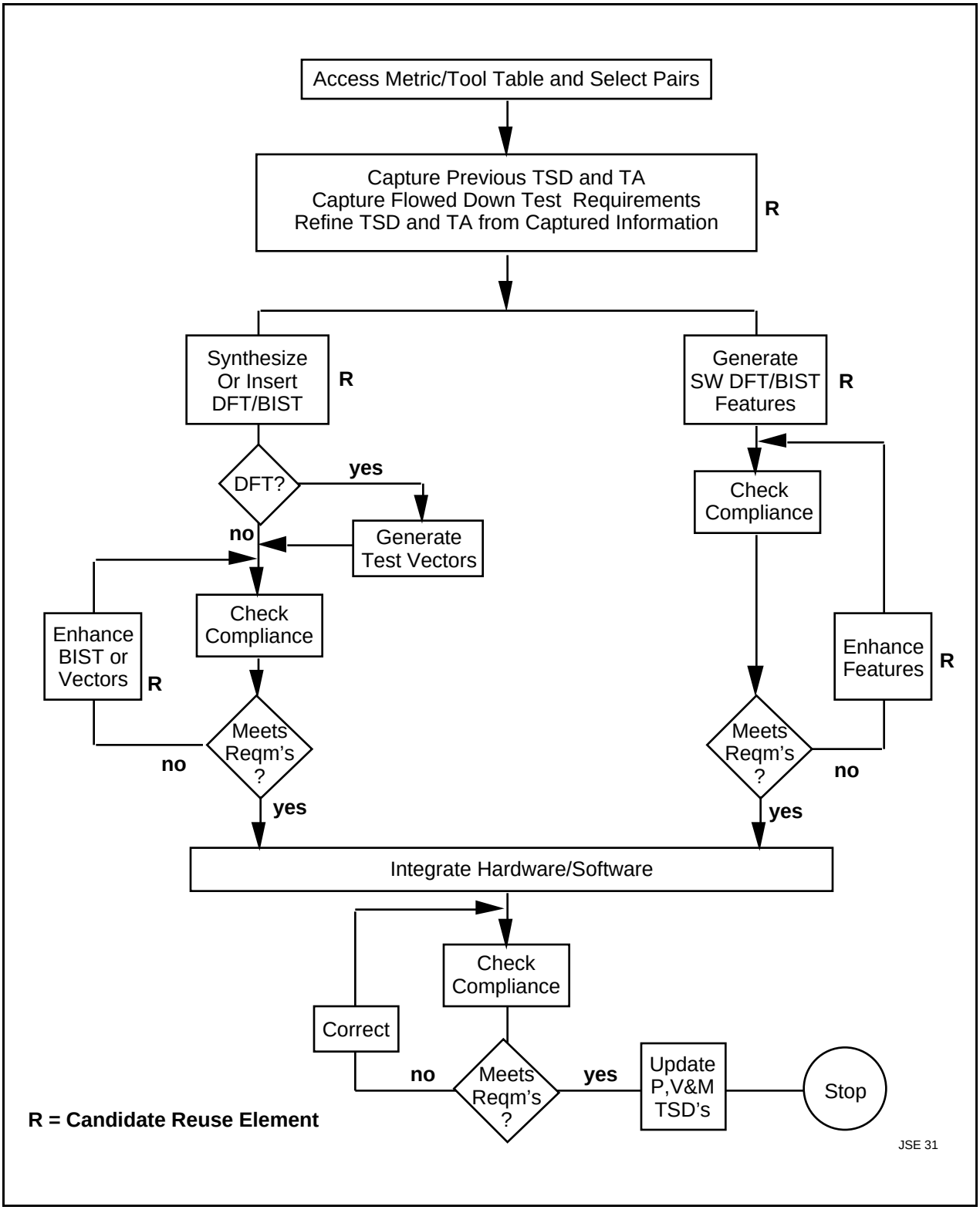


Figure 3-35. Generic flow during detailed design.

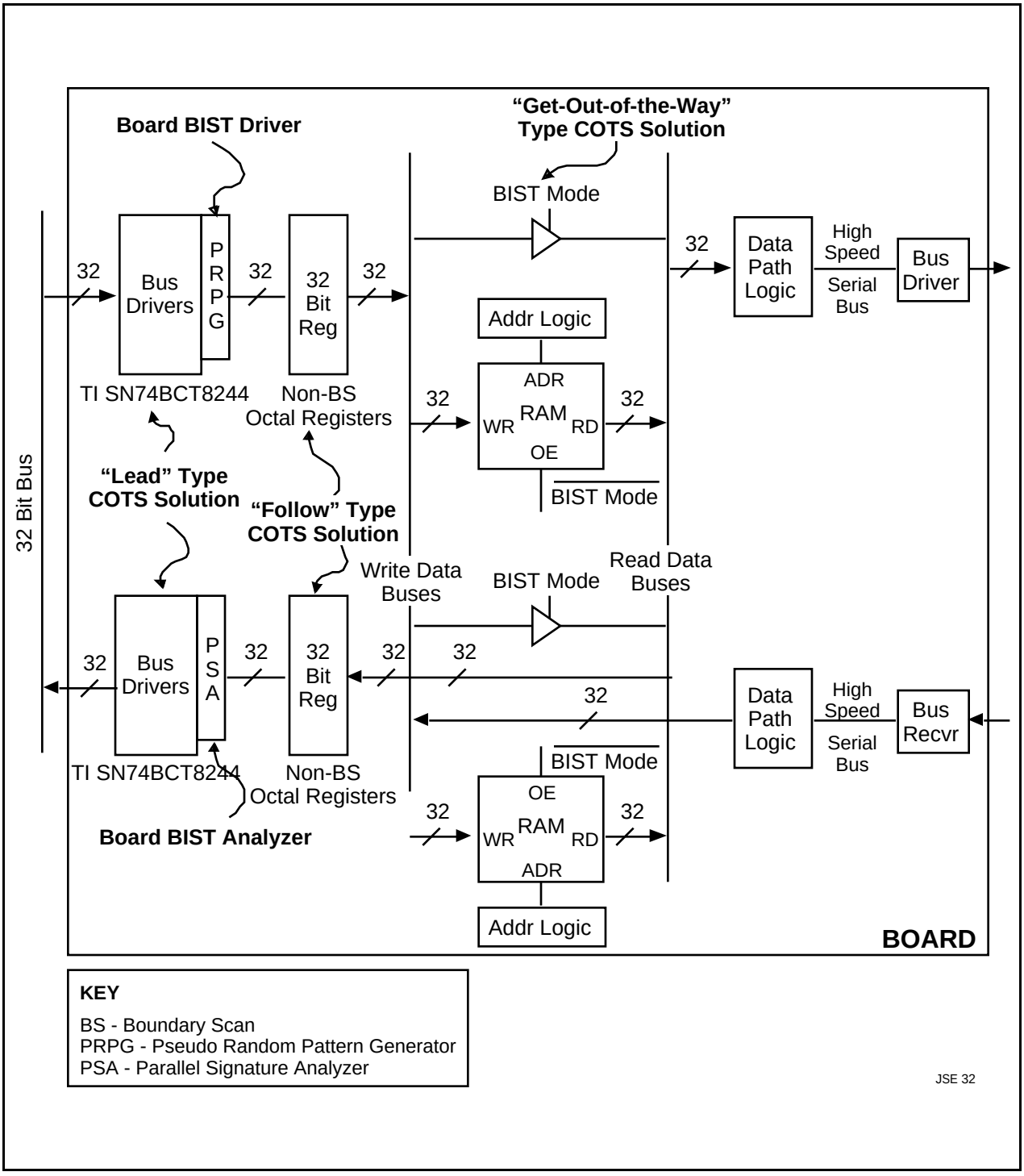


Figure 3-36. One aspect of dealing with COTS in DFT solutions

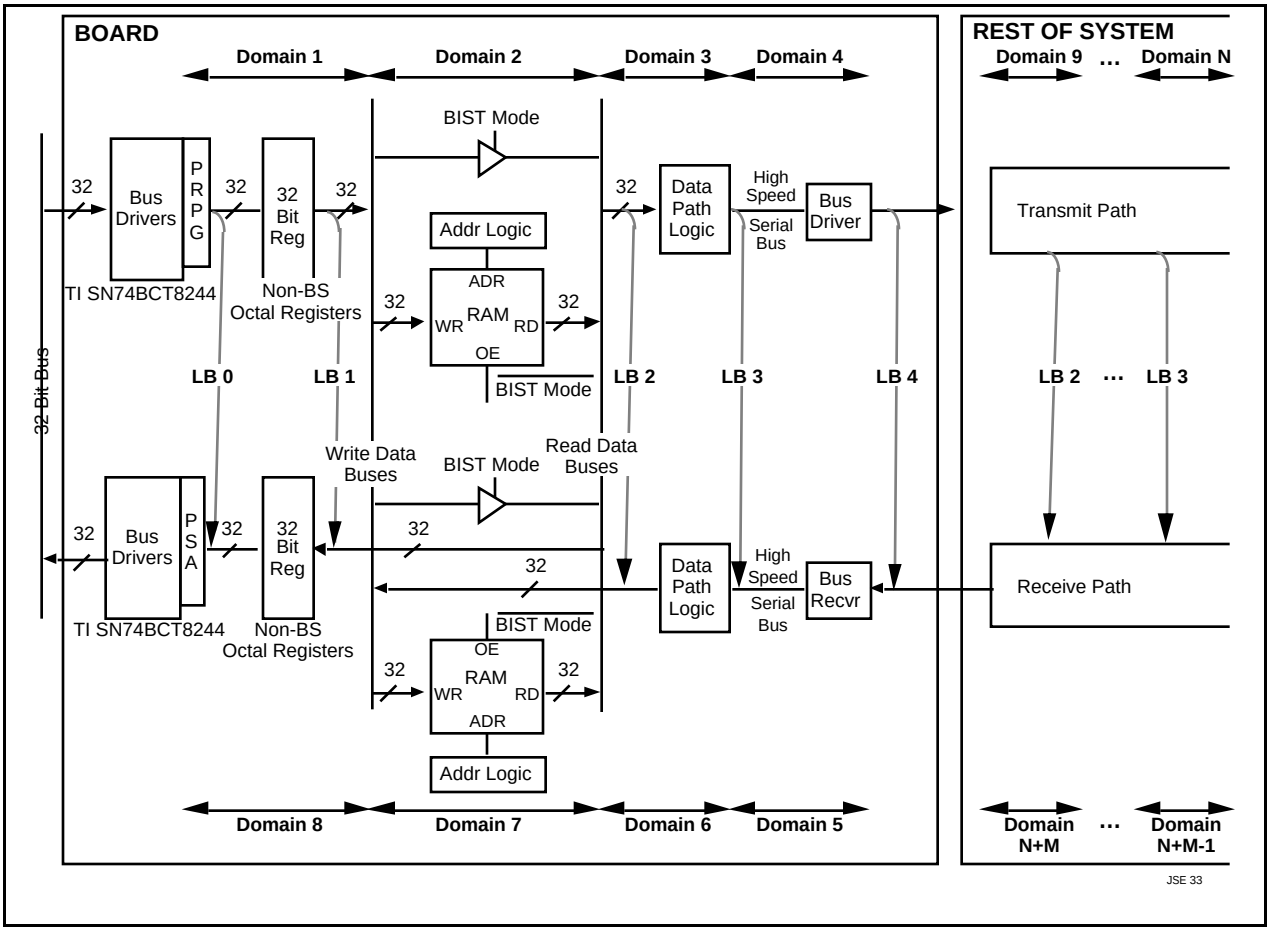


Figure 3-37. An example of test domain analysis.

3.4 Integrated Software View

The overall RASSP process is based upon the philosophy that dramatic decreases in cycle time can only be achieved with maximum reuse of both hardware and software elements. This leads to the notion of a graph-based, correct-by-construction methodology using validated library elements that represent either software fragments and/or hardware instantiation of these elements. This section provides a view of the overall software development that is part of the hardware/software codesign activity distributed throughout the architecture and detailed design processes.

Within RASSP, software development encompasses DFG-based software, control flow software, support software, and library population. We assume that a RASSP-developed signal processor must communicate with a higher-level system, perhaps a command and control system for the platform. Messages must be accepted from the higher-level system that specify actions to be taken by the signal processor, such as data requests, parameter setting, mode switching, etc.. We also assume that the signal processor software must support diagnostics, error handling, startup initialization, downloading etc. In summary, the signal processor software development must result in the generation of all the code that must be downloaded to each processor within the system. If all required signal processing can be constructed from validated elements from the reuse library, the signal processing software represented by the DFGs is automatically generated and documented. When all required processing cannot be constructed from validated library elements to meet the requirements (or when new hardware architectural elements are added to the reuse library), we must generate and validate new library primitives and/or operating system support software elements. This is referred to as library population and is discussed separately in this document. In addition to the signal processing represented by the DFGs, it is also necessary to develop the control software. This, as discussed earlier, is referred to as the command

program. Command program generation is supported by emerging tools, which include autocode generation from state transition diagrams.

3.4.1 Software Development Description

The top-level software architecture is shown in Figure 3-38. The application software consists of an interface to the world outside the signal processor in addition to a command program and data flow graphs. This could either be a user or, more likely, it could be a higher-level system as part of an overall platform. This interface receives messages from the outside which control the processing to be performed, request data from the signal processor, or provide parameters to the signal processor. The DFGs represent the signal processing algorithms that must be applied for each operating mode of the signal processor. The signal processing algorithms are represented as DFGs that drive the autocode generation to produce the executable code for each processor in the system. This process is supported by a reusable primitive library. The command program is the control flow program that provides the overall control, as dictated by the received messages. A run-time support system (represented in the figure by the control and data flow interface) provides the reusable data flow control and graph management for the signal processor. The run-time system is built upon a set of operating system services provided by a real-time microkernel using a standard interface. The run-time system is usable with any operating system microkernel that provides the necessary set of services. The run-time system will be a set of run-time application programs which constitute a 'virtual' machine that performs all graph execution functions and all system interface, control, and monitoring functions.

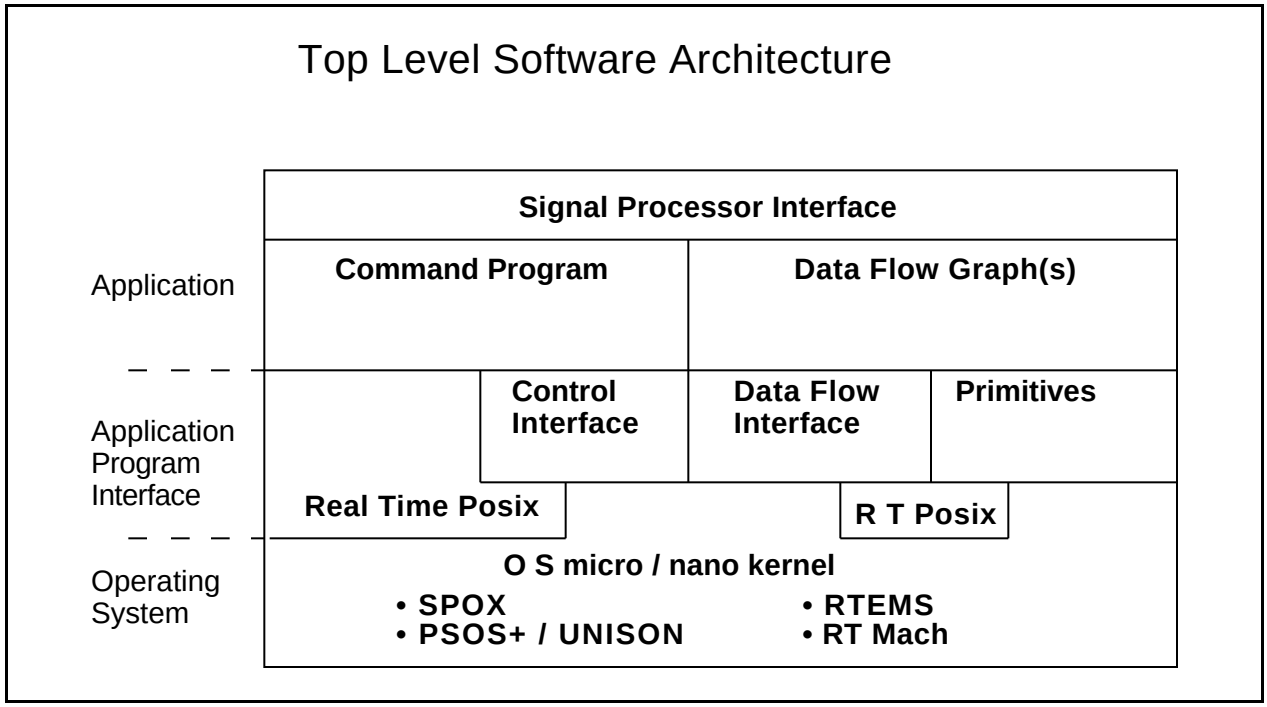


Figure 3-38. Top-level software architecture.

Hardware/Software Codesign

The hardware/software codesign discussion presented here is primarily from a software perspective. As part of the virtual prototyping methodology, RASSP uses the results of simulations tailored to the various levels of the design process, which span system design, architecture definition, and detailed design. Figure 3-39 shows the progression of software generation from the requirements to load image, with emphasis on the graph objects involved and the general RASSP process in which they occur. Also shown is the parallel development co-simulation of the command program. Feedback is not shown in the figure, since this is not intended to be a process diagram, but rather a walkthrough of how we generate and transform the signal processing DFG to downloadable code for a target processor. In reality, many of these steps (along with intermediate steps not shown) occur iteratively as we develop the hardware and software architecture.

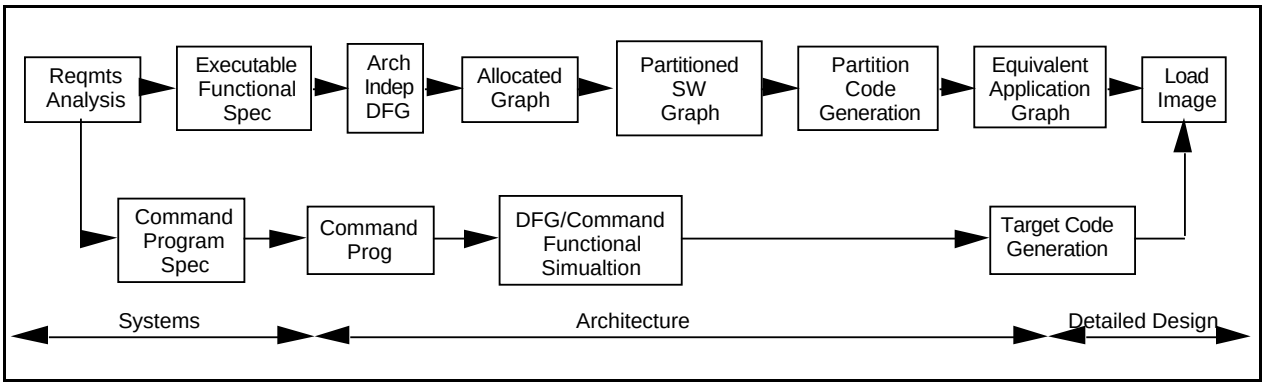


Figure 3-39. RASSP graph-based software development scenario.

During the system definition phase, we develop the initial system requirements. These requirements typically include all external interfaces to the signal processor; throughput and latency constraints; processing flows by operating mode; all mode transitions; and size, weight, power, cost, schedule, etc. We use high-level simulations to generate overall processing strings that are functionally correct and satisfy the application requirements. At this level, no allocation of processing functions to hardware or software is made.

Architecture definition involves creating and refining the DFGs that drive both the architecture design and the software generation for the signal processor. The input to this phase is the executable functional specification for a particular application and the command program specifications. During the architecture definition process, we develop DFG(s) of the signal processing and allocate the elements to either hardware or software. The flow graph(s) are simulated both from a functional and performance standpoint with increasing fidelity until we achieve an acceptable hardware/software partitioning that meets the signal processing system requirements. During the architecture process, we conduct various trade-offs among architecture classes, processor types, specialized hardware elements, and communication mechanisms. We perform high-level, initial performance simulations so that we can quickly evaluate gross architecture trade-offs with size, weight, power, and cost estimations. We also make a first pass at the non-DFG software requirements and tasks definitions.

Once we select a candidate architecture for optimization, the simulations become more detailed and therefore more clearly represent the final system performance. At this level, we perform automated generation of the software partitions to provide executable threads that run on a group(s) of DSP elements. The items to be traded in the detailed software mapping are latency, processor utilization, memory requirements, and computational organization of each of the groups. When the design cycle is completed, the requirements will either be met or the simulation results will show that the candidate architecture is not capable of meeting these requirements. In the latter situation, we use the performance analysis results to indicate where to make architectural changes. The process is iterated until a satisfactory solution is obtained. We further develop non-DFG software code. During architecture verification, we use prototype hardware and behavioral simulations to simultaneously verify the performance and functionality

of the overall signal processing system. The final verification is performed hierarchically, since it is unreasonable to simulate the entire system at the RTL or ISA simulator level because: a) the computation requirements of the signal processing application are beyond the scope of feasibility for simulation at the clock level; and b) the verification must be driven by the types and fidelity of the models available in the component library.

The output of the architecture process is both a description of the detailed hardware architecture and a detailed description of the software (including the computation, scheduling, control, and operating system elements) that must be generated/aggregated for downloading to each of the elements in the architecture.

The final step in the software development occurs during detailed design. The software aspect of the detailed design process is the production of the load image. We must generate and package the final code for downloading and execution on the target hardware. Since we verify most of the software developments during the architecture phase, they are limited at this point to generation of those elements that are target-specific. This may include configuration files, bootstrap and download code, target-specific test code, etc. All the software is compiled and verified (to the extent possible) on the final virtual prototype.

The software load image generation is an automatic build process that is driven by the autocode generation results. The inputs to the process include the architectural description, the detailed DFGs describing the processing, and the partitioning and mapping information produced in the architecture design and verification process. This process is controlled by a software build management function that extracts the necessary information from the library and manages construction of all downloadable code, as directed by the partitioning and mapping data.

The software build management function must coordinate construction of the software to be downloaded to each processing element in the target architecture, the bootup software which initializes the processor, and any control code required for scheduling and sequencing the processors. When complete, the 'build' footprint is preserved in the library under configuration control. In addition to preparing this software, the 'build' management function initiates the documentation process. The documentation function uses documentation elements stored in the library to construct the overall software documentation. All library elements, including both operating system services and application primitives, must have associated documentation elements contained in the library that fully describe the algorithm, its implementation, ranges of applicability, and validation test results.

Since the software is autocode-generated, there are certain assumptions about the validation testing of the resultant software that must be supported by the library process. We assume each primitive has been unit tested. This means that a test suite has been developed for each primitive and this primitive has been instantiated by the software generation process and validated. During the architecture process, we periodically repeat this unit testing as the signal processing graphs are transformed from one form to another. The validation process has also been applied to control program software and we do not have to validate each instantiation of

the control program. Details of the validation of the library elements is given in Section 5.4 of the Draft Software Validation Plan (CDRL A008 10/93).

The software which is constructed must finally be validated on the target hardware. This should be based upon the system test stimulus that has been carried throughout the development process. This is the first time that the software and physical target hardware has come together. Validation of the target software is outlined in Section 4.3.2 of the Draft Software Validation Plan. Based upon the testing, simulation, and virtual prototyping that has been performed throughout the RASSP architecture and detailed design processes, the final validation time should be considerably shorter than the traditional integration and test times.

Library Management

Since the overall RASSP methodology is library-based, managing the reuse library is an extremely important function. The domain analysis and organization of the reuse library is key to minimizing the developer's frustration with the process.

The library must contain, for each primitive, a representation or model that supports both behavioral and performance simulation at various degrees of fidelity. In addition, the library management system must manage the application DFGs, candidate architectures, DFG partitioning files and mapping files.

Information available in the library to support the generation of target software is shown in Table 3-5. This information is comprised of architecture, application, operating system, and support data. It includes the DFG that describes the processing, the software primitives that make up the processing nodes in the flow graph, the communication elements that facilitate the transfer of data between processing nodes, the partitioning and mapping data that describe how the flow graph is mapped to the individual processors, all the operating system kernels and support software the kernels use, any initialization information that may be necessary, and BIT data. The library also contains a mapping of the target-independent primitive to the specific processors supported in the Model Year Architecture. The software run-time system depends upon a given set of operating system kernel operations. These must be available for each specific DSP type and memory, I/O and message passing structure.

The library stores all information required to generate target software including all the elements shown in Table 3-5.

Table 3-5. Library information available for software build management.

Architecture Data	Application Data	Op Sys/Support Data
• Arch description file including processors and interconnections • Partitioning & mapping file describing how the application flow graph is allocated to processors • Documentation elements	• Flow graph(s) describing the required processing • Software primitives representing the processing nodes in the flow graph • Initialization data • Documentation elements	• Operating system kernel(s) • Operating system services elements required to control the transfer of messages or data between nodes • Built in test data • Boot code

The library management system also stores pertinent information relative to the resultant software builds for the target hardware. Version build 'footprints' include all information necessary to reconstruct the build.

Documentation

The documentation we must produce includes all the required documentation delivered with the target hardware. It is important to note that the familiar 2167A waterfall software development and documentation process is no longer applicable to the library-based, correct-by-construction instantiation of deliverable software. The burden of documentation, validation and test generation is more appropriately allocated to the library population function for the individual software elements. However, when the final software is orchestrated for the target hardware, some form of documentation suitable for both users and maintenance must be produced. The exact form of this documentation is TBD.

To support the documentation process, detailed documentation of library elements is a required part of the validation process for inserting elements in the reuse library. As discussed under library population, prototype software elements may be generated during the architecture selection process to support design trade-offs. When we select these prototype elements for permanent inclusion in the reuse library, they must be formally validated. Formal validation includes, among other things, generating detailed documentation for the library element. System documentation requires the assembly of the DFGs, along with the detailed library element documentation, the partitioning and mapping of those elements to the overall architecture, and the associated operating system kernel elements.

3.5 Library Population

When any of the required support components are not present in the library, we cannot proceed with the design without at least defining a prototype element. We call the process of adding any component function to the library ‘library population.’ When the first RASSP example is designed there will exist, by design, a sufficient set of primitives and components in the library to support the example. However, this set must be extended over time to accommodate new technology and applications. The methodology supports the ability to add or modify components of this library. Successful application of the RASSP concept depends upon the ability to perform the library population rapidly and economically so that the project development timelines and cost are not adversely affected. It is important to note that a complete library component is not needed for all the design phases. The methodology allows the design process to proceed in stages, along with the library population effort, to complete the library description. We accomplish this by permitting prototyping of primitives for inclusion in the library as a prototype element. The prototype element may then be used in the hardware/software codesign process to perform high-level architecture trade-off studies during architecture selection. In reality, we can early (based upon the algorithmic processing flows developed during the system design process) that we require new primitives for the application. In fact, application-primitives may be prototyped during system design to simulate the functionality of the required processing. After being confirmed as a desirable primitive, we must validate the prototype element for permanent inclusion in the reuse library. We can perform this validation process in tandem with other portions of the development.

There are three library population software development activities which must be supported:
1) building a signal processing primitive library element, 2) building an operating system primitive library element, and 3) developing hardware models. These three functions are shown in
Figure 3-40 and discussed in the following paragraphs.

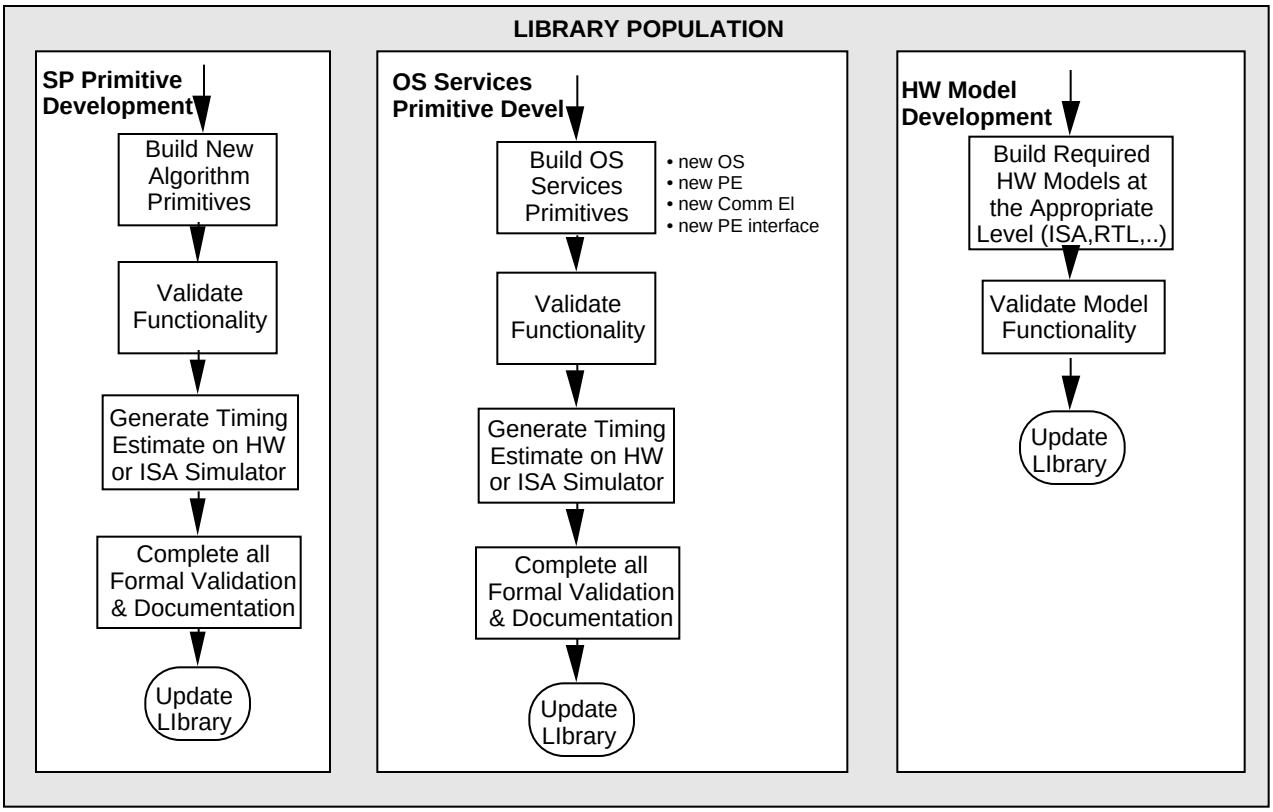


Figure 3-40. Library population top-level process.

Signal Processing Primitive Development

We anticipate instances during the architecture definition process where signal processing functions desired by the designer will not be available in the library. This may be recognized at any time during the design process from early systems analysis on through the architecture process. Since the algorithms required to implement the application drive hardware/software codesign, it is most likely that we will recognize the need for new primitives early. The frequency of these instances will diminish over time for a given application domain as the library is continually extended. There are two ways that a new primitive can be prototyped. First, we can construct the new algorithm hierarchically from more elementary library elements to allow an approximate design to proceed with the understanding that the eventual implementation may have to be optimized for run-time by combining these primitive operations into a new primitive. Second, we can implement the new primitive as a HOL description of the algorithm.

To build a new primitive, the primitive generation tool(s) should interact with the user (i.e., application programmer) to construct the prototype for experimentation in the architecture design process. Templates for prototype primitives are stored in the library to support the developer. These templates provide the encapsulation wrappers to interface with the various tools in the system. Approximate timing information for the primitive (for the processor type(s) of interest) must also be generated or estimated to support the architecture trade-off process. This timing information may be updated as the primitive validation proceeds.

After developing the prototype element, the architecture definition process can proceed. We can use the prototype element in the synthesis, evaluation, and trade-off of candidate architectures during hardware/software codesign.

We continue library population by validating the prototype element for permanent inclusion in the library. There is more to the library validation process than simply validating the prototype code. To be a permanent library element, the code must be generated as a domain-primitive suitable for constructing domain-primitive graphs that are architecture-independent. We must validate it over a broad range of test stimulus, document it as a permanent part of the library, and model it in terms of performance on all processor types in the library. This process is aided by templates maintained in the library.

As we validate the prototype code and add support for the code generation process and validate it, the new library element can be used in the autocode generation. It should be noted that library elements may be optimized, which could result in multiple implementations of the primitive being maintained in the library. For some processors, it may be appropriate to optimize the primitive in assembly language or microcode to maximize performance. Translations of the domain-level primitive to the target processors of interest will be maintained in the library in the form of target processor mappings.

In addition to validating that the code produced is functionally correct, the validation process must produce the necessary documentation elements that are also maintained in the library. This documentation should include the test suites used and the resulting test report.

Operating System Services Primitive Development

There are four instances where we must generate or modify operating system services. These include adding a new operating system, a new processing element, a new communications element, or a new processor interface.

Adding a non-computational architectural element, such as a communication element (e.g., cross bar switch), processor interface, or I/O interface, may require modifications or additions to the application support software maintained in the reuse library. In many cases, the new element requires additions or modifications of operating system kernel support functions, such as drivers or mapping tables, which provide the operating system services to the applications.

The addition of a new operating system requires that all operating system services that are assumed to be available for any processor of interest must be generated for each processor type for which the new operating system is intended for use. We assume that the operating system vendor has provided the operating system for one or more specific processors, and that only the operating system services required to support the application program interface to the run-time system are required.

When we insert a computational element as a new architectural element, we assume that the required HOL compiler(s) is available for the processor. In the RASSP methodology, the atomic computational element may be any general-purpose processor or DSP that is supported by a HOL compiler and/or assembler.

Adding a new computational element requires that we modify other existing library information, or add new library functions. First, the functional performance of all application-primitive elements intended for the new computational element must be verified; that is, they must compile and execute the test suite to produce the correct results. To provide the information required for the hardware/software codesign process, we must update the application primitives and the operating system services elements to incorporate timing estimates for the new computational element. This process must be performed for all primitives in the library available for the new computational element. In addition, to support automated generation of software for the target hardware, an operating system that provides the standard set of services required to support the methodology must be validated for the new computational element. Specialized microcoded functions (such as math libraries) for the new element must also be incorporated into the library so that primitives use the optimum code when translated and compiled for the new

computational element.

Hardware Model Development

When new hardware elements are required in the design process, we must develop VHDL models suitable for use by the various tools operating at different levels of abstraction. This could be a new signal processor or a custom hardware design. Ideally a new signal processor is supported by appropriate models from the manufacturer, but that is not always the case. In the case of a custom hardware element, it may be necessary to develop a high-level performance model for use in the architecture design process, as well as more detailed models suitable for detailed analysis and synthesis. Model generation tools will simplify the process and reduce model generation time.

Section Four
Manufacturing/Integration and Test

The goal of RASSP for manufacturing/integration and test is to support interaction with manufacturing centers and suppliers to enable implementation of concurrent engineering on development projects. Developing automation technologies for manufacturing centers is being addressed concurrently with RASSP on other program and internal development efforts.

For the RASSP program to achieve significant improvements in productivity and design quality, seamless interfaces are required that support the interaction of RASSP design centers with manufacturing centers/suppliers.

This section addresses our approach to implementing those interfaces, leveraging related government and industry-funded efforts, such as the Preamp program, the ASEM program, and PDES/STEP efforts.

4.1 Manufacturability Assessment/Manufacturing Plan

The key role of the manufacturing team member on the Product Development Team (PDT) is to work with design engineering team members to assure a producible design that meets performance, cost, and quality objectives. It is important to involve the manufacturing team member early in the design process. During the conceptual design phases, we formulate general manufacturing capability requirements based on electronic packaging considerations for the anticipated technology mix (ASICs, MCMs, PWBs. etc.), anticipated module sizes, I/O requirements, etc. Figure 4-1 illustrates the PDT manufacturing tasks in relation to the RASSP design tasks.

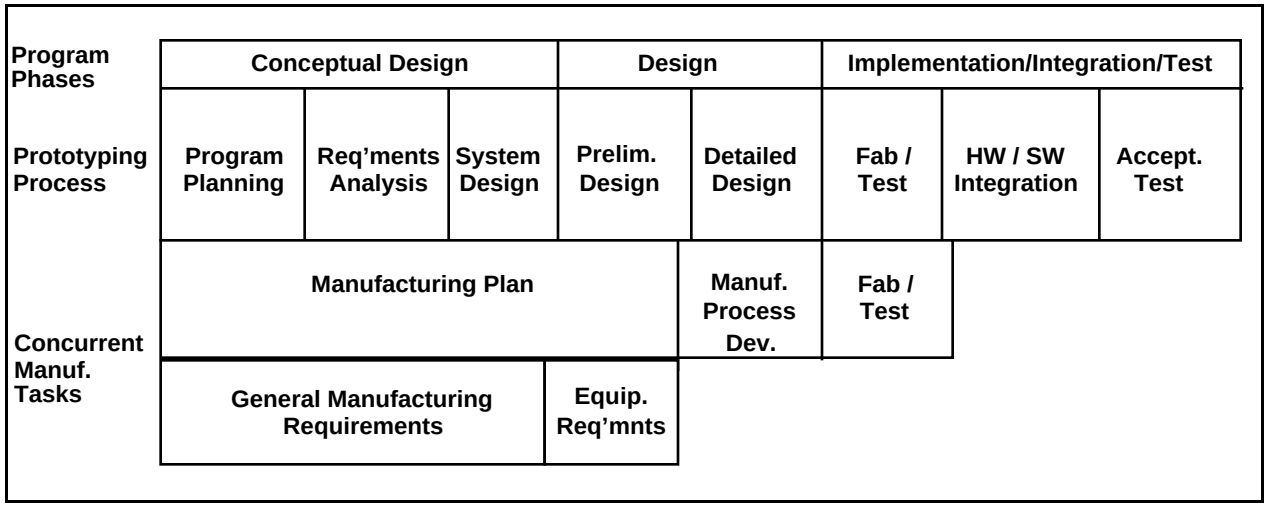


Figure 4-1. Concurrent PDT design and manufacturing task.

During conceptual design, developing the manufacturing plan begins. At this stage, we identify basic capital equipment and decide capital investment and make versus buy issues. For RASSP, these decisions relate to desired processor implementation, dealing with issues such as packaging densities, device lead pitch, fabrication technology, board or substrate materials, and other technology-related considerations. We establish producibility guidelines to define preferred minimum lead spacing, footprint geometries, hole sizes, etc. The production cost model used to project the processing product cost is also updated to reflect more refined manufacturing cost estimates. Ultimately, we will develop a manufacturing cost profile for early prototype units and follow-on production units.

During detailed design, we develop the detailed manufacturing processes with each assembly and sub-

assembly operation sequenced. Examples are the choice of machines for auto-insertion based on device size and densities, the wave soldering process to be used, etc. Specialized manual component mounting processes may be required to optimize heat transfer for power devices. We define processes to be outsourced at this stage, such as bare-board fabrication, cables/harnesses fabrication, chassis fabrication, etc.

An important element of the manufacturing plan is the test plan. We decide the approach for subassembly test, such as in-circuit test versus full -functional test, etc. This dictates test equipment requirements which, in turn, dictate special test equipment and fixtures requirements. Developing the test plan must be an integral part of the electrical design process to ensure adequate test point coverage consistent with testability goals, and adequate test point locations consistent with test equipment probe capabilities.

4.2 Manufacturing Interface

The RASSP system includes a manufacturing interface to support concurrent engineering interaction. The manufacturing interface enables virtual prototyping and physical implementation of designs at multiple qualified manufacturing sites through a standards-based engineering and manufacturing information-sharing automation environment.

To implement the RASSP manufacturing interface we implement specific CAD system capabilities within the design centers and manufacturing centers, and electronic linking with the RASSP design centers.

4.2.1 Manufacturing Interface Requirements

RASSP industrial design centers, in production of military and commercial equipment, must be capable of interacting with multiple external organizations for manufacturing and supply functions. The centers must be highly streamlined and automated to reduce cycle time and cost while improving product quality. Standard information exchange formats and protocols are essential to enable this interface flexibility.

Top-level views of the information flow in typical design/manufacturing interface operations are illustrated in Figure 4-2. The RASSP enterprise framework provides the support necessary to enable this interaction. Engineers at RASSP design centers interact with the enterprise environment to use CAD tools and frameworks in design processes. This enterprise environment also enables the design team to interact electronically with corresponding environments at manufacturing centers and suppliers to share information and services. Interaction is independent of the particular interconnection mechanism (internet, private network, etc.).

Figure 4-3 provides various conceptual user views of the shared information environment (designer view, manufacturing engineer view, etc.). At their workstations, design engineers are provided with manufacturing center data/services and supplier information; manufacturing engineers are provided access to selected designer data/tool, and supplier owned data.
The information flow through the manufacturing interface should support initial capability

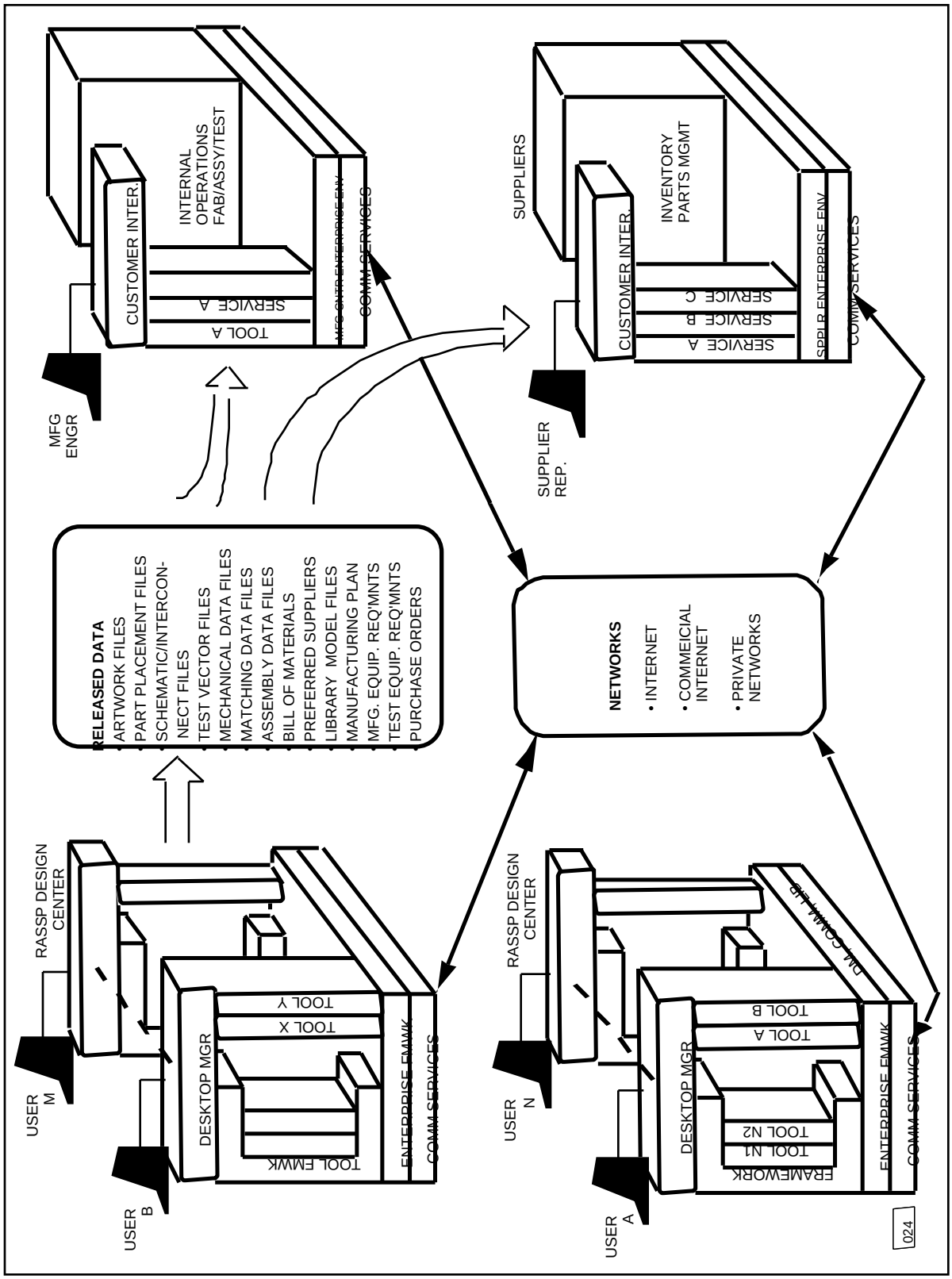


Figure 4-2. Manufacturing/Supplier interaction with RASSP Enterprise environment manufacturing centers.

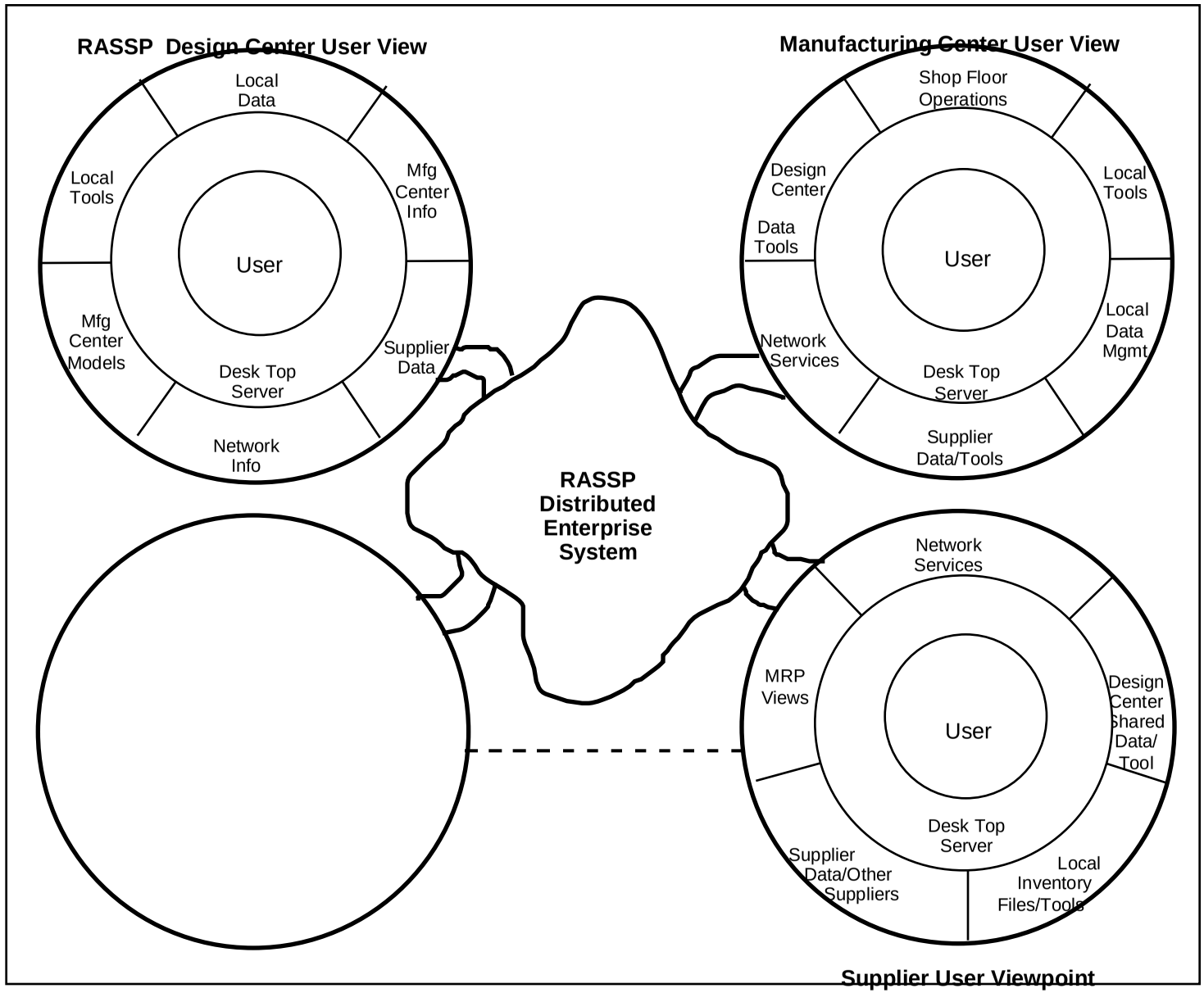


Figure 4-3. RASSP user views.

assessment of manufacturing centers, sharing of relevant library/database information, virtual prototyping interaction (enabling an early assessment of manufacturability, testability, cost, etc.), transfer of design and procurement information in standard formats, and final documentation and product acceptance at the RASSP center. The interface also should support the electronic commerce aspect of handling quotations, orders, competitive bidding, etc.

Examples of specific automated capabilities required are identified in Figure 4-4.

• On-line Capability Assessment	• Electronic Information Server • Access to technology data • Capability, Capacity, Pricing Information
• Electronic Commerce Support	• Receive quotation requests • Receive Orders • MIL Specs
• Information Sharing	• Libraries • Standard packages, test fixtures, sockets • Design rules; rules of thumb
• Virtual Prototyping	• Producibility checks • Design rule checks; prefab signoff steps • Test program generation/installation

Figure 4-4. RASSP information access examples.

4.2.2 Design/Manufacturing Data Interfaces

The RASSP enterprise environment includes a specific manufacturing interface utility that supports interaction of the design centers with manufacturing centers. This function leverages the interface development work being developed on the Preamp program, as well as information-development tasks on related DoD programs, such as PAP-E and ASEM. The architecture for this interface implementation is illustrated in Figure 4-5.

4.2.2.1 RASSP Manufacturing Interface Implementation

The RASSP manufacturing interface, shown in Figure 4-5, includes provisions for converting electronic design information to STEP (Standard for The Exchange of Product Data) compliant formats, assessing manufacturing center capabilities, and negotiation facilities to support concurrent engineering activities associated with resolving technical issues in the engineering to manufacturing interaction.

The architecture of this interface is modeled after the Preamp framework and information sharing services.

Preamp is a three-year research program started in July 1992 and being implemented by a consortium of companies (including Martin Marietta), with matching funding provided by NIST through the Advanced Technology Program. Preamp is also closely tied with the PDES Inc. consortium, which is also on the Martin Marietta RASSP team. The RASSP program will use the technology resulting from Preamp to implement the manufacturing interface, and will demonstrate the interface capability in with the Martin Marietta manufacturing facilities, and the ASEM contractors.

We are designing the interface to support:

- Intelligent information sharing for facilitating concurrent engineering functions

- Manufacturing process plan generation capability for the Integrated Product Development Team (IPDT) based on initial product descriptions and manufacturing capability descriptions

- Mechanisms to support the capture, maintenance, sharing, and use of knowledge to create a decision-support infrastructure - enabling effective use of manufacturing/test expertise by the systems engineering IPDT team.

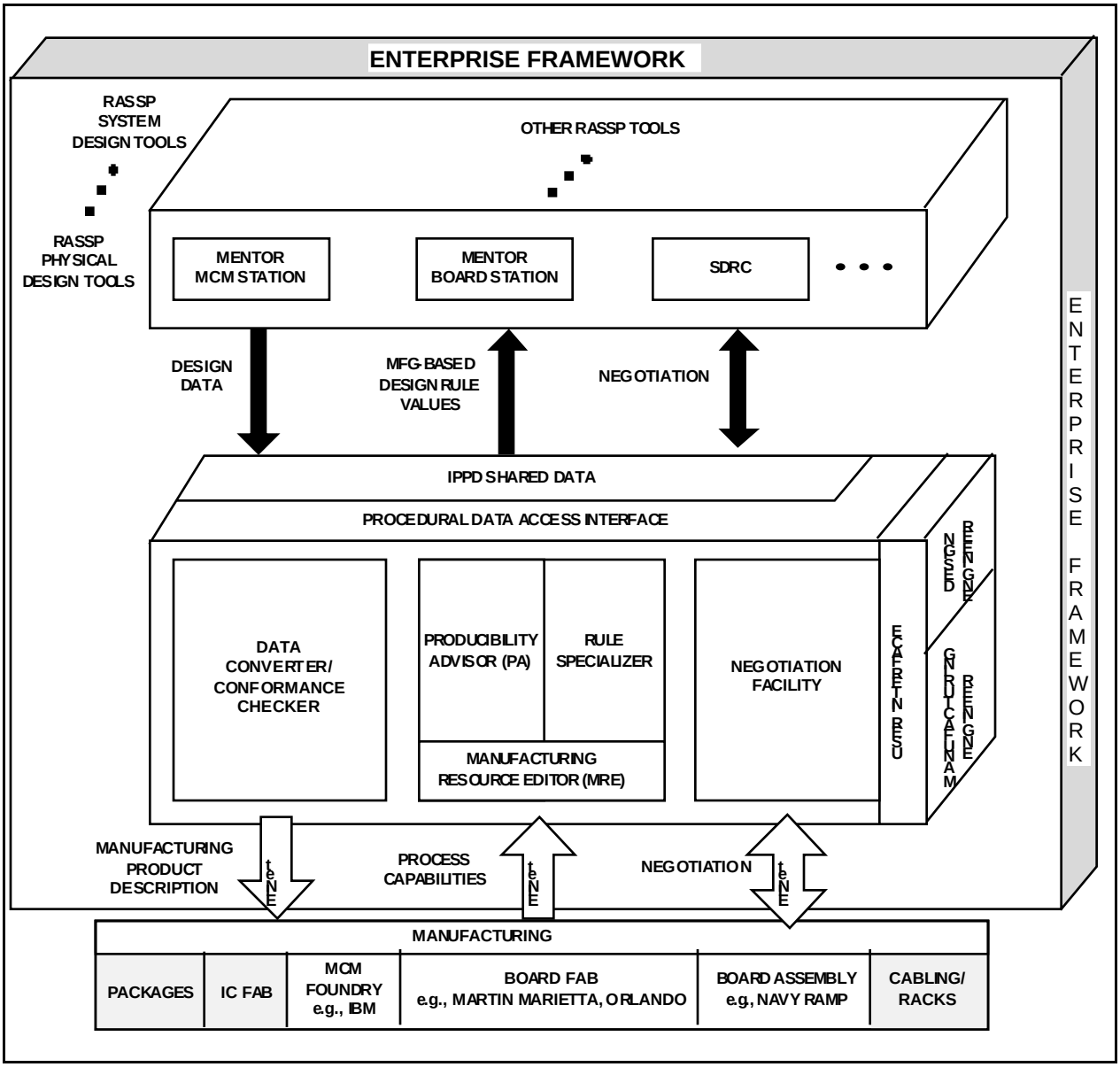


Figure 4-5. Manufacturing interface implementation.

We are designing the interface to allow the product design engineer, manufacturing process engineer, and manufacturing engineer to work together in three contexts:

- Assessing compatibility of product concepts and given/fixed manufacturing processes
- Determining the ability of a manufacturing process to support a mix of products
- Identifying manufacturing process enhancements needed for the product concepts being developed.

A Preamp database is the core function of the manufacturing interface. This database implements the STEP information models and Preamp/RASSP extensions necessary to accomplish concurrent engineering through information sharing. The database information includes the following:

- STEP AP 210 Model of a Printed Circuit Assembly (PCA) - This model contains all the information produced in design necessary to manufacture a PCA (see following section for a description of this information for the Mentor environment).

- Test Information — PIEE model of a manufacturable PCA product being developed on the PAP-E program.

- Information Models of Manufacturing Process Plans — Being developed for the Preamp program.

- Information models of knowledge in the form of rules and models of issues in manufacturing (for support of concurrent design for x methodologies - producibility/cost/etc.).

In addition to the database function of the manufacturing interface, flexible data conversion utilities are included to support interaction with current COTS tools not conforming to STEP standards. An integrated inference engine for utilization of the captured knowledge in the manufacturing interface is also provided.

The STEP Standard Data Access Data Access Interface (SDAI) [ISO-10303-22] is the glue that brings the applications data together with the manufacturing interface database. SDAI is a programming interface defined by the ISO as part of the STEP effort so that it may be automatically generated based upon an EXPRESS information model.

We also integrate the inference engine using the STEP SDAI paradigm. A consistent set of semantics must apply to the information within the manufacturing interface database and its use in applications. To accomplish this, the STEP methodology defines the SDAI programming interface in terms of an information model represented in EXPRESS. Because the internal schema of the Preamp database and the SDAI code are based on the same semantic description, a meaningful exchange of information is ensured.

In a concurrent engineering project, each person contributes to the product according to their discipline, training, and experience. To capture this knowledge, designers and engineers may formulate rules to avoid problems encountered in the past. For example, a test engineer's rules might describe how to assure that a product can be tested, or that testing costs and time are reduced. A manufacturing engineer's rules might address producibility or quality issues. Such rules are termed "Design-For-x" (DFx) rules, where the x may be any of a number of product or process considerations. DFx examples include Design-For-Manufacturing, Design-For-Test, Design-for-Assembly, Design-For-Time-To-Market, Design-for-Serviceability, and others.

Preamp is implementing a knowledge or DFx-based approach to concurrent engineering, which we will use in the manufacturing interface. Two software components support DFx: A Knowledge Capture and Execution Facility (KCEF) and a Negotiation Facility. The KCEF provides user-friendly mechanisms that allow experts to define DFx rules and to apply those rules to a product description in the database. The inference engine is used to chain through a set of appropriate rules to answer any of a number of DFx questions (e.g., does this product violate any design/manufacturing/test criteria, is special handling of any of the components required, are there special processes to follow when soldering/drilling/assembling the product, etc.). The DFx rules are stored in the database, extracted, and reformatted for use with the inference engine. Rule classification mechanisms are provided to enhance chaining performance and aid in rule maintenance.

4.2.2.2 RASSP Manufacturing Interface - Product Release Package

The Martin Marietta Engineering Process Improvement (EPI) program engineering release standard defines the set of electronic files involved in transfer of a PWA design to a manufacturing center. We will use this standard, and the interface models being developed on the Preamp program (discussed previously), as an initial basis to define the information requirements for the RASSP manufacturing interface. This will evolve throughout the program, being driven by additional requirements generated by the RASSP benchmark and demonstration efforts.

An example is shown in Figure 4-6 that illustrates the flow of design data from engineering to manufacturing for PWAs. We typically develop this design data on the Mentor Graphics board design tools

(Board Station) and it is then used by manufacturing operations. The Mentor Graphics tools and the data objects generated by these tools are shown on the left side of the figure. The dashed boxes in the center of the figure describe the part-processing operations on the data, and the shaded boxes on the right side of the figure describe the manufacturing usage of the data.

Figure 4-7 illustrates the data and associated files for a specific PWB fabrication. PWB fabrication data defines specific PWB design information in sufficient detail to be used in industry standard computer-aided-manufacturing (CAM) processes and to drive numerically-controlled (NC) machines. This data is stored in ASCII files and supplied in standard industry formats for transmitting the information between the PWB/CCA design activity and the PWB manufacturing activity. PWB fabrication data consists of PWB artwork data and machining data.

4.3 Integration and Test Plan

The approach to integration and test of the systems and subsystems is often overlooked in the initial phases of development projects, which results in significant cost and schedule problems for programs. It is essential that this program aspect be addressed as an up-front systems engineering task in the development process.

Our goal in this effort is to significantly reduce the effort/time associated with this phase of program activity by leveraging the extensive simulation/virtual prototyping validation efforts performed as part of the RASSP methodology.

The RASSP concurrent engineering methodology addresses this function in the IPDT with a comprehensive integration and test planning task performed in parallel with, and closely coupled to, the design efforts.

The objective of this task is to develop a plan for system/subsystem integration and test covering individual module/subassembly testing through total system test. The plan addresses concurrent

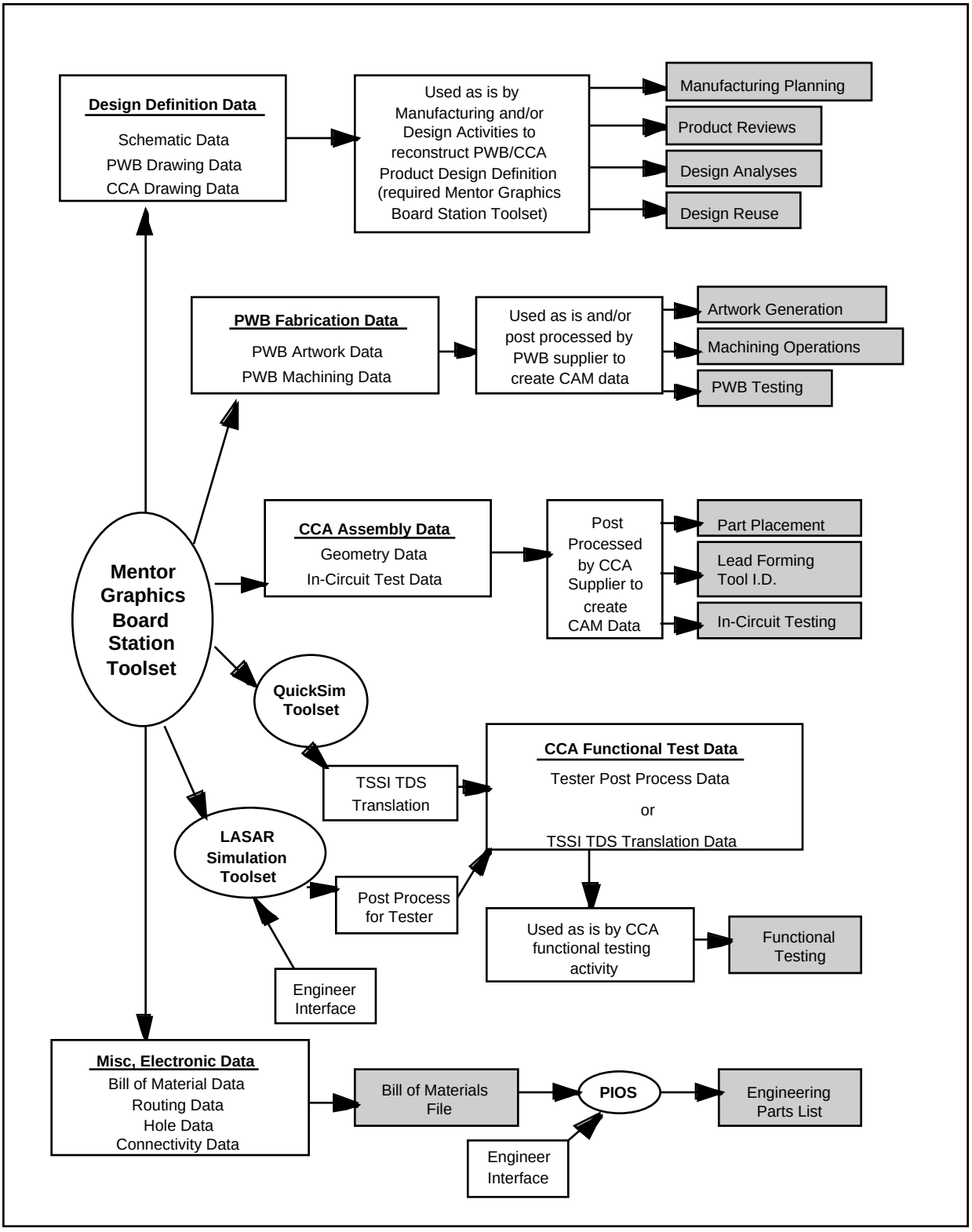


Figure 4-6. Example design and manufacturing data interface.

PWB Fabrication Data	Version 7 Files (see note 1)	Version 8 Files (see note 1)
PWB Artwork Data:		
Format Data:	Photoplot_data_file	artwork_format.Pcb_artworkf.attr artwork_format.artworkf_v
Aperture Data:	Aperture_table_file	Aperture_table.PCB_apertt.attr.aperture_table.apertt_v
Pattern Data:	artwork_#	artwork_#
PWB Machining Data:		
Format Data:	drill_data_file mill_data_file	drill_format.Pcb.drillf.attr. drill_format.drillf_v milling_format.Pcb.millingf.attr milling_format.millingf_v
Tool Data:	drill_table_file mill_table_file	drill_table.Pcb.drillf.attr. drill_table.drillf_v milling_table.Pcb.millingf.attr milling_table.millingf_v
Pattern Data:	drill drill_unplt milling	drill drill_unplt milling

Figure 4-7. PWB fabrication data.

testing of parts of the system and incremental assembly and test of larger functional units. The integration and test plan covers the following:

- Identify Testable Partitions - Select functional partitions of the system that are testable in standalone fashion.
- Develop Unit Test Strategies - Develop unit test strategies for each partition (including equipment/fixturing requirements and test software requirements).
- Define Integration Strategy - Determine the sequence of integration steps and requirements for test/debug support at each phase of the integration. The plan should maximize the capability for concurrent debug of loosely coupled sections of functionality. Incremental capability builds should be specified to enable visibility into progress achieved in the integration/test process.
- Define Test Plans for Incremental Builds - Identify the specific tests, fixturing, and possible special test equipment required to support execution of each of the integration phases.
- Define Test Implementation Plan - Define the plan for ensuring that the test data and special test equipment is developed and implemented to support the integration process. The test should be developed as part of the normal development process, and used to a large extent in the simulation process.

Section Five
… 'ilities and Special Engineering

5.1 Reliability Engineering

5.1.1 Reliability Engineering Objectives

There are three reliability engineering program objectives:

- Influence the system design toward maximum reliability within existing performance and cost parameters.
- Influence manufacturing procedures and processes toward removing as many defects from the hardware as is technically and economically feasible to achieve the designed–in reliability.
- Provide a design that will obtain the highest possible operational availability (A_O) with the lowest life–cycle cost (LCC).

5.1.2 Reliability Engineering Program Responsibilities

We prepare an explicit reliability engineering program plan in accordance with Task 101 of MIL-STD-785 (DI-R-7079) to describe the approach and planning procedures, as well as the organization to implement, execute, and control all activity necessary to satisfy the reliability requirements of the contract statement of work. The activities described in this program plan are the preventive and remedial measures, shown in Figure 5-1, and discussed in the following paragraphs. Upon approval, this program plan becomes the governing contractual document for execution of the reliability program.

- Technical Program Planning — Reliability engineering is a member of the design team at the initial onset of the program. Close coordination of reliability engineering with the design process ensures understanding and implementation of reliability requirements into documents such as the Prime Item and critical item specifications.
- Design Analysis — We perform design analyses, such as reliability modeling, allocation, and predictions; failure modes and effects analysis (FMEA); part stress analysis; thermal survey; and vibration survey to measure capabilities, provide inputs for trade–offs for alternative designs, and identify reliability-critical items. Reliability allocations are issued based on predicted and design goal MTBFs. The component application is based on MIL-P-11268 derating guidelines; design engineering documents for applied component stress levels are used in the reliability predictions.
- Comprehensive Test Planning — As the system progresses, we plan the required testing to assure that the system satisfies all performance requirements. Associated reliability test activities consist of: test, analyze, and fix; reliability growth testing; reliability demonstration/qualification; critical item testing; environmental stress screening; and burn–in. Each reliability-related test is integrated into the Developmental Test Plan to eliminate duplicate effort and maximize cost efficiency and effectiveness.

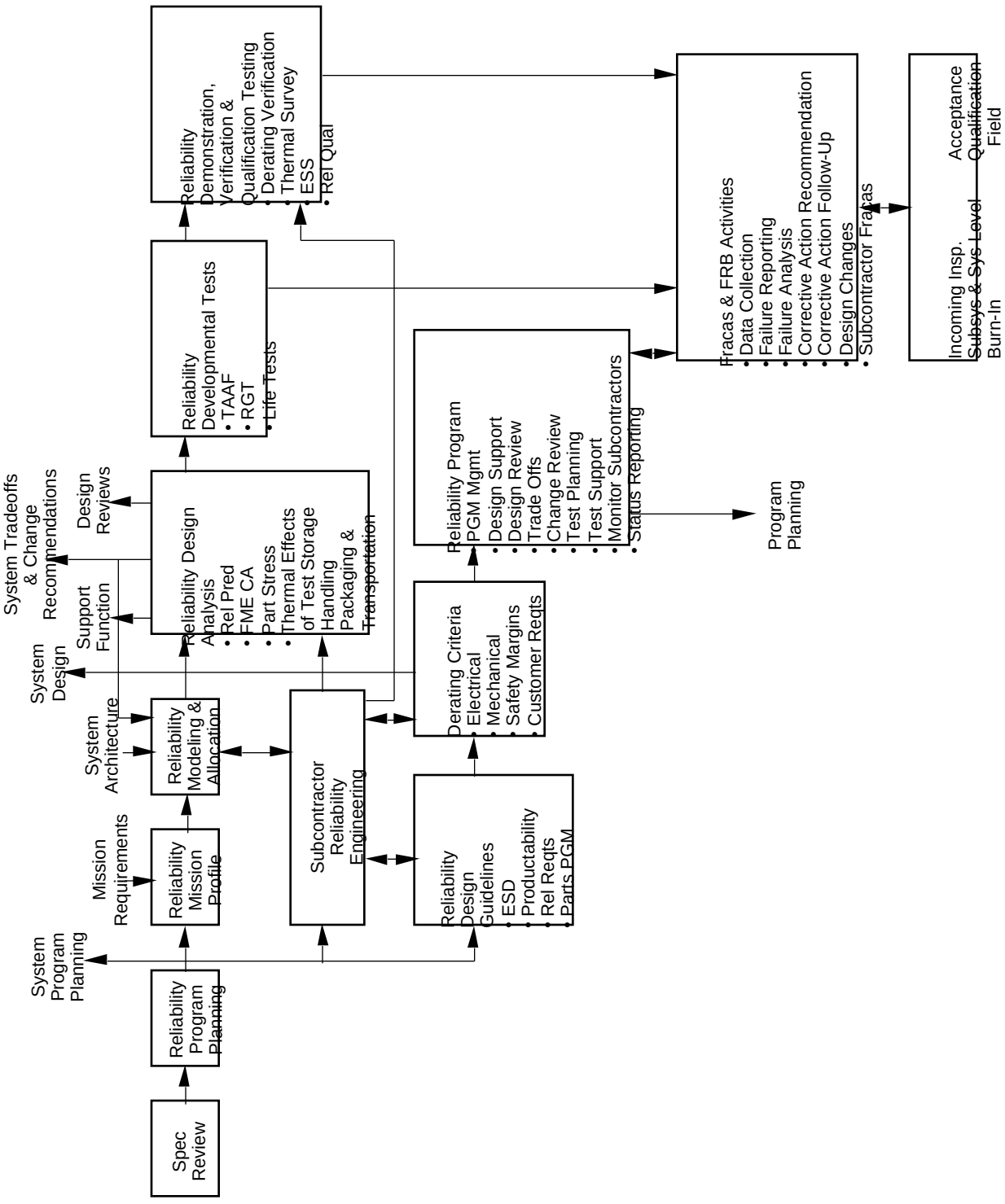


Figure 5-1. Reliability program tasks.

- Design Review — Reliability engineering is an active member of all engineering technical and customer design reviews. In the design reviews, we present the status of the reliability design analysis and provide formal input to the design process to assure that the system can satisfying its performance requirements. We prepare data items, such as PDR and CDR data packages, which allow the customer to evaluate the technical data before the design review.

  Remedial measures are actions associated with manufacturing of the system. These measures provide early problem recognition and effective corrective action. A qualitative description of remedial measures is defined as follows:

- Program Monitoring/Control — A lead reliability engineer is assigned and empowered with the necessary resources to manage and control in–house, subcontracting, and supplier reliability activities. Close monitoring assures a timely, cost-effective reliability program.

- Test Procedures and Implementation — Reliability engineering supports the generation of test procedures and test implementation/performance of the aforementioned reliability tests.

- Failure Monitoring, Reporting and Analysis — Reliability engineering uses an aggressive failure monitoring system, which emphasizes rapid reporting, thorough evaluation, and meaningful corrective action, to assess all test activity. Activities within this system include: periodic inspection of manufacturing processes; review of problem reports and test failure reports to identify failure trends; input failure data into reliability engineering's Failure Reporting, Analysis and Corrective Action System (FRACAS) database; and issuing periodic failure summary reports.

- Failure Review Board (FRB) Activity — Reliability engineering chairs the FRB, which is convened to review failures entered into the FRACAS database, define needed corrective action, and ensure effectiveness of corrective action implementation. The FRB also ensures close coordination of program management and reliability, design, maintainability, safety, manufacturing, components, and quality assurance disciplines during the evaluation and corrective action implementation for failures.

- Configuration Control — Reliability engineering is a permanent member of the Configuration Control Board (CCB) and it reviews all engineering change notices for their impact in reliability.

- Program Reviews — Reliability engineering presents the status of open reliability activities at both informal and formal program reviews. The status of reliability activities are evaluated along with the other program disciplines to establish overall program priorities.

Figure 5-2 lists the specific tasks addressed by the Reliability Program Plan, cross-referenced to the applicable Military Standard.

5.2 Maintainability Engineering

5.2.1 Maintainability Engineering Objectives

The objective of maintainability engineering is to improve operational readiness, reduce maintenance requirements, reduce LCC, and enhance system availability. It is the maintainability engineering responsibility to ensure the product meets both qualitative and quantitative maintainability requirements for required levels of maintenance stated in the program specification. Typical quantitative maintainability requirements are: mean time to repair (MTTR), maximum corrective maintenance time (Mmax), probability of fault detection/isolation, false alarm rate, etc. Maintainability engineering performs analyses using maintainability-related military standards (MIL-STD-470, MIL-STD-471, and MIL-HDBK-472 to determine the maintainability quantitative requirement compliance. If the product fails to meet the quantitative maintainability requirements, maintainability engineering identifies problem areas and coordinates with systems engineering and design engineering to correct the deficiencies.

Reliability Tasks	MIL-STD/Task #
Program Plan	MIL-STD-785B TASK 101
Design Criteria	N/A
Design Review	MIL-STD-785B Task 103
Prediction/Allocation and Modeling	MIL-STD-785B Task 201, Task 202, and Task 203 MIL-STD-756B Task 102, Task 201, and Task 202
Failure Modes, Effects and criticality Analysis Plan	MIL-STD-785B Task 204 MIL-STD-1629A Task 105
Failure Modes, Effects and Criticality Analysis	MIL-STD-785B Task 204 MIL-STD-1629A Task 101
Failure Review Board	MIL-STD-785B Task 105 MIL-STD-785B Task 208
Reliability Critical Items List Failure Reporting and Corrective Action (FRACAS)	MIL-STD-785B Task 104 MIL-STD-785B Task 303
Reliability Test Plans, Procedures and Report	MIL-STD-781D Task 301 MIL-STD-785B Task 301
Environmental Stress Screening Test Plans and Procedures	MIL-STD-785B Task 301
Environmental Stress Screening Report	MIL-STD-785B Task 102
Thermal/Vibration Survey	
Subcontractor Control	MIL-STD-785B Task 207
Parts Control Program	MIL-STD-965 Procedure 1

Figure 5-2. Reliability task elements.

5.2.2 Maintainability Engineering Responsibilities

Maintainability engineering coordinate with systems and design engineering to ensure compliance with the overall maintainability requirements. Maintainability engineering develops the maintainability design concept/criteria/guidelines from the SOW, PIDS, military standards, etc., and distributes it to system and design engineers to assist them in the product development. Maintainability design is verified independently to ensure that the product meets the guideline requirements.

Maintainability engineering interfaces with various engineering disciplines and the program management office. Figure 5-3 illustrates the maintainability interfaces and identifies the respective information flow. The interface scenarios between maintainability engineering and other engineering disciplines are described in the following paragraphs.

Compliance with the quantitative maintainability requirements is essential to the product design; however, maintainability engineering is also responsible for continuing to improve the product design by emphasizing the qualitative aspects of the maintainability design. Maintainability engineering must achieve the following maintainability design objectives by designing the product:

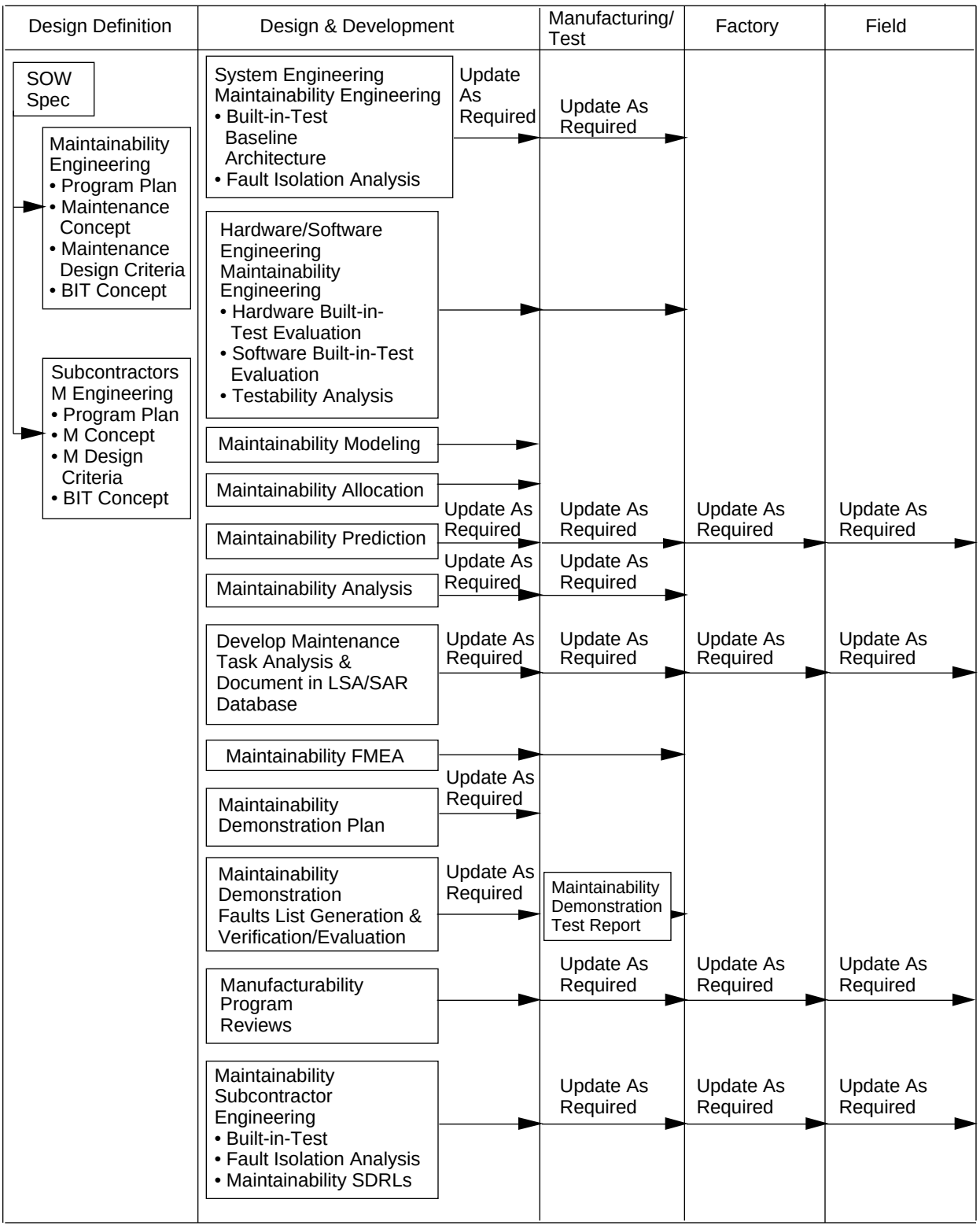


Figure 5-3. Maintainability information flow.

- So that the skill requirements for maintenance personnel are within maintenance skill level required by the procuring agency.
- To minimize support equipment requirements.
- So that all elements are labeled with clearly visible, full identifying information.
- So that no special tools are required for maintenance.
- To minimize calibration and alignment requirements.
- For modularity.
- For easy accessibility to components requiring maintenance, adjustments, inspection, and removal/replacement.
- For commonality and interchangeability.
- With safety guidelines for both equipment and personnel involved in the performance of maintenance and operation.
- With adequate test points for fault isolation.

The key to verifying that the design meets both quantitative and qualitative maintainability requirements is to perform the maintainability demonstration test (MDemo). During the MDemo, maintainability engineering collects the product maintainability data and performs maintainability evaluations to ensure both qualitative and quantitative maintainability requirements are met.

5.2.3 Maintainability Program Plan

Maintainability engineering conducts the maintainability program in accordance with the agreed upon program plan. Upon approval, this program plan becomes the governing contractual document for the execution of the maintainability program. The maintainability task elements are performed by maintainability engineering, which maintains a close interface with hardware, software, and systems engineering during design development to ensure that maintainability factors are designed into the system. Figure 5-4 lists the specific tasks to be addressed by the maintainability program plan, cross-referenced to the applicable Military Standard.

Maintainability Tasks	MIL-STD/Task #
Program Plan	MIL-STD-470B Task 101
Design Criteria	N/A
Design Review	MIL-STD-470B Task 103
Prediction/Allocation and Modeling	
Test Plan/Procedures/Report (Maintainability Demonstration)	MIL-STD-470B Task 301
Subcontractor Control	MIL-STD-470B Task 102

Figure 5-4. Maintainability task elements.

Maintainability engineering is the key contributing discipline for built-in-test (BIT) development. Maintainability engineering assists the systems and architecture processes in defining the BIT concept. In addition, maintainability engineering performs the system fault detection, fault localization, and fault isolation analyses to ensure BIT coverage meets the required levels. If the analysis reveals a deficiency, maintainability engineering identifies the deficiency and recommends necessary test points, BIT test coverage, and test methods to the systems and design engineers so that the product is designed for maximum operability, availability, and supportability.

5.3 EMI Engineering

5.3.1 Objectives

An EMI control program is initiated and maintained through all program phases to ensure that EMI requirements of the equipment specification and SOW are included in equipment design and documented, and that compliance is demonstrated by test.

5.3.2 Responsibilities

The EMI control activity ensures that all EMI effort is planned, incorporated in design, demonstrated by test and documented in accordance with program requirements. This includes ensuring all system and equipment design, and integration and test activities include the required consideration of requirements in basic design features; and that design guidelines and techniques are specified and incorporated in all unit and subassembly designs.

Figure 5-5 illustrates the EMI control organization interaction and responsibilities, and the relationship of EMI control with other program activities.

5.3.3 Documentation

An EMI Control Plan, Test Plan, and Test Report are prepared and submitted for customer review and comment in accordance with SOW, CDRL, and Data Item requirements to ensure visibility of planning effort.

5.3.4 EMI Control Plan Summary

The EMI control plan has design guidelines, design techniques, engineering practices, and program control measures used to ensure that the subassemblies comply with applicable EMI requirements of MIL-STD-461C.

The plan describes product subassemblies, EMI design features incorporated to comply with each of the applicable requirement methods, predictive analysis of anticipated extent of compliance, any expected problem areas, and recommended method of problem resolution.

The plan describes any informal evaluation testing and planned qualification testing.

The EMC control plan is updated and maintained through all program phases.

5.4 Parts Engineering

5.4.1 Parts Engineering Objectives

The objective of the parts engineering activity is to establish and maintain a parts control program that influences the design at the earliest possible stage so that use of standard parts can be maximized; to evaluate nonstandard parts, recommend alternate parts, and ensure that such parts are approved according to MIL-S-965A; and to track part usage and all submittals for approval.

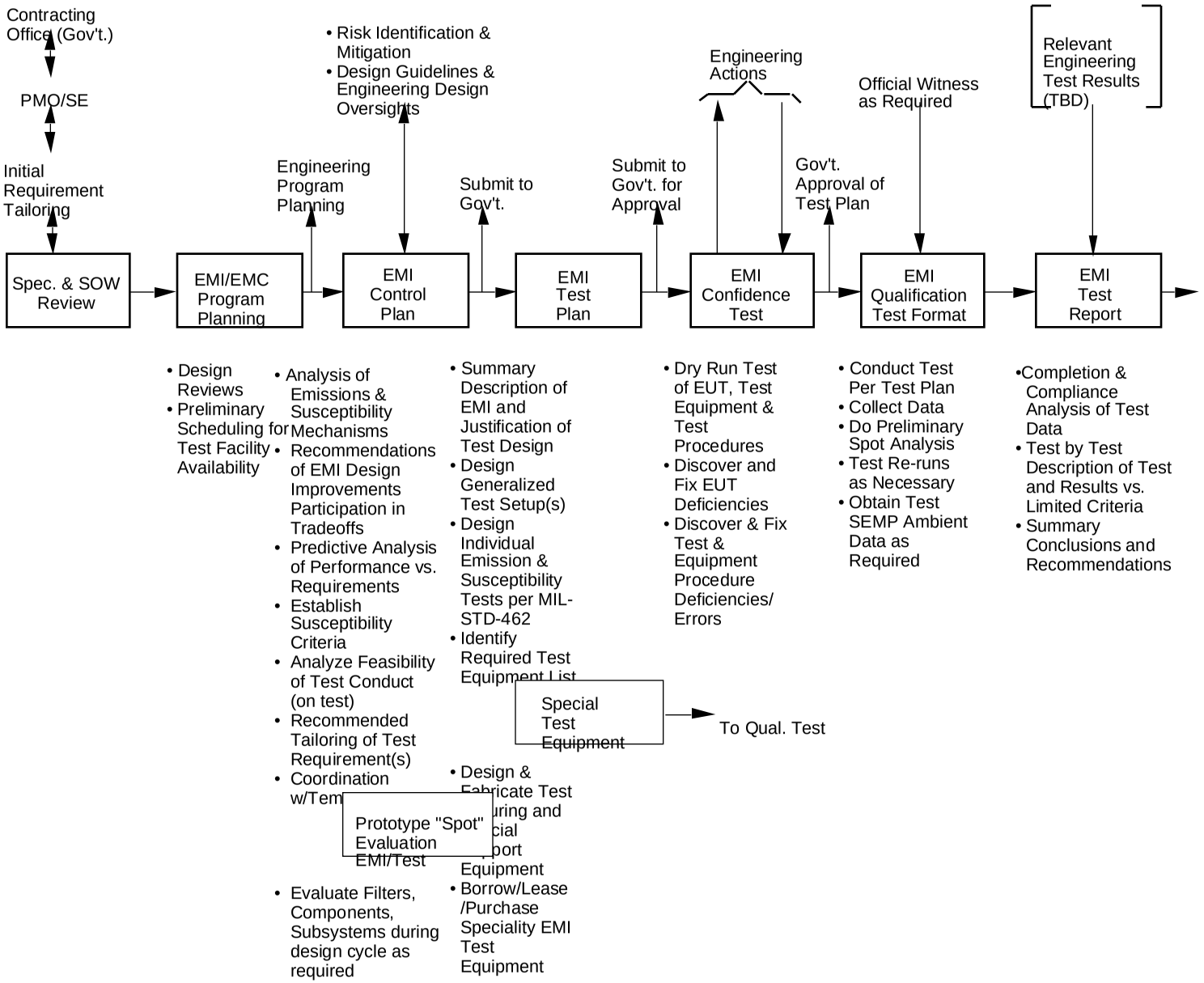


Figure 5-5. EMI control and test effort.

5.4.2 Parts Engineering Responsibilities

Parts engineering provides guidance on part usage and selection to design engineering, and it ensures that the use of standard parts is maximized in the design. Parts engineering recommends alternate parts if the proposed parts are nonstandard, submits nonstandard parts for approval, and submits a Program Parts Selection List (per MIL-S-965A) for each part. Parts engineering determines the qualifications and upgrade screening necessary for nonstandard parts that are not Class B (for microcircuits) or equivalent (for passive devices). The usage of all parts, standard and nonstandard, as well as submittals for approval, are tracked in a database.

5.4.3 Parts Engineering Program Plan

The parts engineering program flow is indicated in Figure 5-6, and each task and corresponding CDRL item is listed in the following paragraphs. The most important aspect of this plan is to maintain a constant and close interface with design engineering to ensure that the use of standard parts is maximized. Selection of standard parts is facilitated through a Component Library Management System (CLMS). Every part considered for use and its pertinent information (vendor name, generic number, static sensitivity, etc.) is tracked in CLMS. Design engineers access CLMS to minimize time spent researching parts.

5.5 ILS Engineering

5.5.1 Objectives

The RASSP approach to ILS is based on the concept of design-for-supportability or the need for early inclusion of the life-cycle support requirements into the development effort. This is accomplished through early supportability assessment via the systems engineering process, followed by the development of integrated documentation, data and plans to support that design in the field.

The four specific ILS engineering objectives are to:

- Influence the system design and operational concept toward maximum supportability within existing performance and cost parameters.
- Encourage modular design of the subsystem LRUs for easy accessibility and quick–connection/disconnection.
- Provide a design that will obtain the highest possible operational availability (Ao) with the lowest LCC.
- Identify the most cost–effective support structure and associated resources to ensure supportability.

LSA, as the integrating mechanism for logistics, impacts designed–in supportability and is the source of data for all ILS support elements by: providing configuration data, operations, and maintenance procedures to support requirements and recommend associated resources; and defining tasks at each maintenance level for each repairable item.

As an integral part of the system engineering team, LSA engineering provides supportability guidance and evaluates the logistics impact of design alternatives and trade studies. Supportability guidance is in the form of recommending supportability design features, defining test capabilities of existing army support equipment, and imposing standardization considerations.

The output of LSA is a set of detailed LSARs that provide in–depth identification of support requirements through depot level.

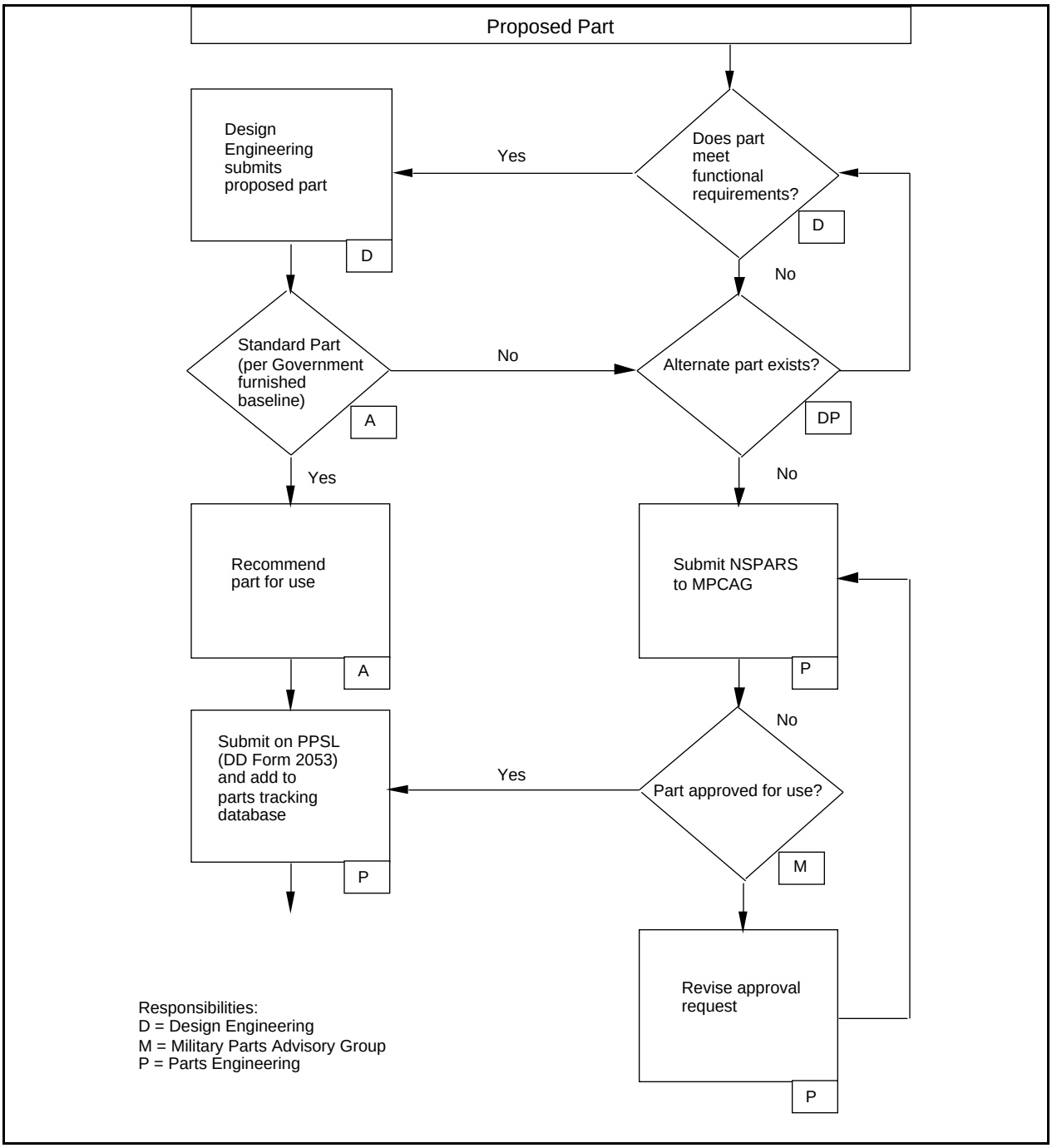


Figure 5-6. Part engineering flow.

5.5.2 Integrated Support Plan (ISP)

This ISP covers the following:

- Describes in detail the procedures, actions, events, and organization to be employed to successfully accomplish the ILS program.

- Assigns responsibilities and establishes milestones for executing the ILS program.

- Describes the processes used in planning, developing, and acquiring the logistics resources for the test support and operational support of the four specified maintenance levels.

- Discusses how design trade–offs and risks are initiated and progressively refined during the product development.

The ISP consists of the following sections:

- Introduction
- A summary of the system characteristics
- Identification of ILS Program Management, Organization and Execution
- ILS Program Tasks
- Milestone Schedules
- Related Plans

Related sub-plans are:

- Logistics Support Analysis Plan (LSAP)
- Verification, Demonstration, and Evaluation Plan
- Manpower and Personnel Integration (MANPRINT) Management Plan
- Level of Repair Analysis Program Plan (LORAP)
- Contract Maintenance Plan (CMP)
- Technical Manual Plan (TMP)
- Training Course Control Document (TCCD)

5.5.3 ILS Tasks

ILS Program tasks are:

- ILS program management
- Logistics Support Analysis (LSA) Tasks/Design Engineering Interface
- LSA Record (LSAR) Tasks/Computer Resources Support
- Logistics Demonstration (LD)
- Maintenance Allocation Chart (MAC)
- Contract Maintenance Plan/Depot Maintenance Study Tasks
- Provisioning Technical Documentation Program/Spare Parts Tasks
- Packaging, Handling, and Storage Tasks
- Technical Manual Program
- Repair Parts and Special Tools List (RPSTL)
- Training Program
- Maintenance and Supply Support
- ILS Data Reporting

5.6 Producibility Engineering

5.6.1 Objectives

The objectives of the producibility engineering specialty are to:

- Reduce production cost
- Reduce cycle time
- Smooth the transition from design to production

5.6.2 Responsibilities

Producibility engineering achieves the objective by considering these areas at a minimum:

- The design must maximize where possible:
  - Simplicity of design
  - Use of economical materials
  - Use of economical manufacturing technology
  - Standardization of materials and components
  - Confirmation of design adequacy prior to production
  - Process repeatability
  - Process inspectability
  - Acceptable materials and processes

- The design must minimize where possible:
  - Procurement lead time
  - Generation of waste
  - Use of critical materials
  - Special Production Testing
  - Use of critical processes
  - Skill level of production personnel
  - Unit costs
  - Design changes in production
  - Use of limited availability items and processes
  - Use of single material or process without alternative

A Producibility Checklist is generated to document the producibility requirements and to measure design conformance.

Producibility engineering periodically calculates the following metrics as a means to measure progress:

- Material Cost
- Labor Cost
- LCC
- Percent of conformance to checklists
- Percent Test Coverage

Producibility engineering uses a spreadsheet program to calculate the following:

- Assemblability
- Theoretical Minimum Number of Parts

- Relative Assembly Time
- Parts Count Design Efficiency

This analysis is used to quantify opportunities for improvement of chassis assembly costs.

Producibility engineering trades off the benefits of greater testability (lower LCC and factory troubleshooting time) verses the increase in parts count, size, weight, power consumption, and cost.

Producibility engineering trades off the benefits of design features that reduce assembly time versus the increase in material costs and engineering costs associated with them.

Producibility engineering coordinates efforts with test engineering to ensure that the equipment is designed so that it can be tested using automated testing techniques, both at the PWA level, LRU level and system level.

Producibility engineering generates a manufacturing plan that contains:

- Facility Layout Requirements
- Capital Equipment Requirements
- Tooling Requirements
- Manufacturing Special Procedures Required
- Preliminary Flowcharts
- Process Plan

5.6.3 Program Interactions

Those interactions required for the producibility engineering program are shown in Figure 5-7.

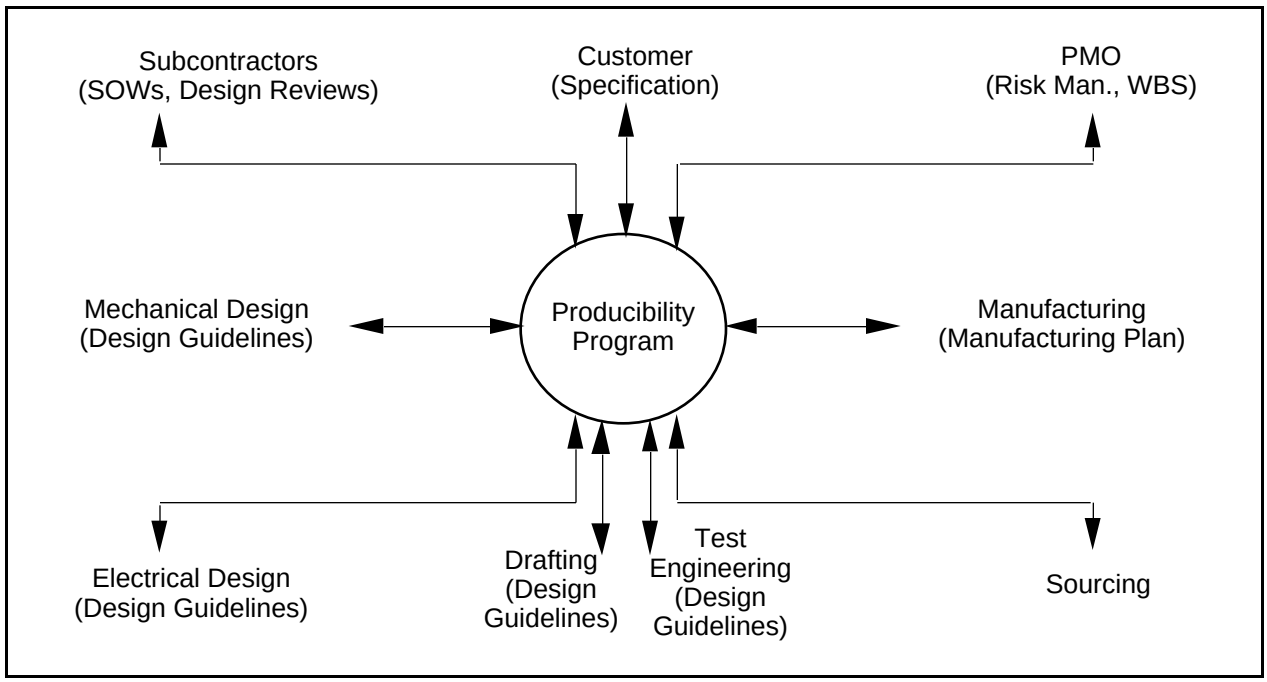


Figure 5-7. Producibility program interfaces.

Appendix A
Design Process and Simulation Figures

Figure A-1 is an end to end diagram of the design process as discussed in this document. The information in the figure is neither new or different from that contained in the document but rather is a consolidation of the top level diagrams from each of the design process discussions into an integrated figure.

Figure A-2 is a picture of the processing from a simulation point of view in the systems and architecture processes. The primary thrust of the figure is the HW/SW Codesign activity during the architecture process but it also shows the concurrent activities which are initiated as the design proceeds. The concurrent activities range from library development(primitives and models) to activities of the Integrated Product Development Team associated with assessing the attributes of a particular candidate architecture.

During the systems process the functional baseline is established through the development of the signal processing flows for all of the processing modes that are required. However, it may be recognized that primitives are required based upon the systems designers knowledge of the reuse library. Although the figure concentrates on the signal processing aspects, part of the systems activity is also devoted to the specification of the non-signal processing software requirements. In particular, the DFG command program must be specified so that both the signal processing and the control of the signal processing can be jointly simulated for correct functionality. The development of the command program can proceed in parallel with the architecture process.

One of the first activities in the architecture process is the development or translation of the processing flows to standard data flow graph notation. The DFG functional simulation provides a validation of the functional baseline expressed in DFG form. Candidate architectures are defined, and in so doing provide the allocation of functionality to either hardware or software. For any candidate architecture, partitioning and mapping allocate the DFG nodes to the elements of the candidate architecture. Performance simulations provide a method of evaluation of the relative performance of the combination of architecture, partitioning and mapping. The performance simulations account for the processing times of the DFG primitives as well as memory and network contention. This is an iterative, interactive simulation cycle in which either the partitioning and/or architecture are modified to achieve some level of optimization. Concurrently with this simulation activity, preliminary assessments of the reliability, testability, size, weight, power, and cost are performed by the Product Development Team.

Given that one or more architectures meet the design requirements, the next level of detail is carried out both for performance simulation and for assessment of implementation attributes. The next level of simulation will incorporate the autocoded software module timing estimates for the partitions, operating system effects and the performance of the run-time system which controls the flow graph execution. Again the simulation cycle is an iterative process in which the partitioning and/or architecture may be further modified to improve performance. It should be noted that each time that a modification is made to either the architecture or the partitioning, the autocode process must also be reiterated. For those architectures which are selected for this level of simulation, it is also likely that updates to the size, weight, power, cost, reliability and testability will be performed. The updates may employ more detailed design advisors and account for implementation technology.

The final portion of the architecture process involves the construction of a hierarchical simulation to evaluate both functionality and performance using hardware testbeds and/or detailed lower level models for the various architecture components. The details of this level of simulation for verification purposes is dependent on the particulars of the architecture and the availability of models. If the entire design is a COTS product design using existing board types with no new interfaces and a well established operating system microkernel, the level of verification that is required is much less stringent than if a new or existing processor requires a new hardware interface to the communication mechanism being used. In the latter case, detailed software will be executed on the low level models of the hardware being built to ensure the proper functionality and timing. The simulation in this instance may result in the modification of either the code or the model. The hierarchical verification plan will exercise all interfaces between architectural elements at the appropriate level of detail to ensure both functionality and performance.

Appendix B
Glossary

Architectural Body	A VHDL construct that is used to define the semantics of a design entity. The semantics are defined in terms of mechanisms that change the values of the signals attached to the output ports of the design entity interface. There may be more than one architecture body for a design entity. All architecture bodies for a design entity share the same interface.
Architecture Selection	Architecture selection is the heart of the RASSP HW/SW codesign which utilizes a library based, DFG driven approach to software development combined with an iterative performance trade-off analysis to support rapid selection/analysis of candidate architectures.
Architecture Verification	Architecture verification is an iterative, hierarchical process whose role is to verify the functionality and detailed performance of a candidate architecture using a combination of testbed hardware, simulator(s), and or emulator(s) prior to detailed hardware implementation.
Back Annotation	The process of assigning values to model attributes as the result of the use of an external assessment tool; especially in assigning precise timing values within higher abstract models from more detailed lower level model simulations.
Behavioral Model	An abstract, high-level VHDL description which expresses the function and timing characteristics of the corresponding physical unit independent of any particular implementation, especially devoid of specific internal structure.
Bus Functional Model	Used to define the operation of a component with respect to its surrounding environment. The interface between the component and its environment are modeled in detail, even though all of the functions internal to the component do not have to be modeled, particularly not at the same level of detail.
Co-Simulation	The term "co-simulation" is used in two contexts: In the context of hardware/software co-simulation, the term refers to the act of simulating the execution of software on target hardware. This is accomplished through a hardware simulation of the target hardware interpreting software instructions. In the context of other domains, besides HW/SW, the term refers to the act of cooperatively running multiple distinct simulators concurrently with inter-process communication between them. Each simulator is simulating a distinct section or aspect of the target system. This can apply to simulations within the same domain, such as Verilog and Quicksim, or to simulators in divergent domains, such as Spice, VHDL, and SPW.
Component	A component is any logically separable hardware unit. They can be combined to form a higher level component by being interconnected. Thus components are directly related to the nodes in the design hierarchy. The VHDL DID requires that VHDL model components correspond to physical components.

Data Flow Graph (DFG)	A directed graph that depicts information flow between signal-processing primitive operations as "arcs" and the transforms or operations that are applied on the data as "nodes".
Design Entity	An entity interface together with an associated architecture body defines a design entity. Different design entities may share the same entity interface but employ different architecture bodies.
DSP (Digital Signal Processor)	A processor system specialized for the computation of signal processing algorithms. It usually consists of many programmable processor elements interconnected via networks to each other and to memory, sensors, displays and other external devices. It is often distinguished from general purpose- or data- processors in that it must operate in real-time, it often has a much higher data input rate, and it usually must perform a higher percentage of mathematical, often floating-point, operations.
Executable Specification (E-Spec)	A description of a component or system that can be executed in a computer simulation to reflect the precise behavior of the intended device. Currently, E-Spec's may often be restricted to describing only specific aspects of the component or system such as timing, performance, or function.
Fault Ambiguity Analysis	An analysis of a system which identifies fault ambiguity groups (groups of components or modules for which a fault cannot be isolated within the group and, hence, repair would cause all items to be replaced).
Functional Analysis	The process of decomposing system requirements into functional blocks which define the behavior of the system.
Functional Model	A model of a hardware system that describes the response of the system to stimuli in a way that is independent of any implementation, and does not provide any information about the timing characteristics of the system being modeled.
Hardware/Software Codesign	The joint development and verification of both hardware and software via simulation/emulation from the hardware/ software partitioning of functionality through design release. HW/SW codesign should result in the development of a virtual prototype (see definition for virtual prototype).
Instruction Set Architecture (ISA)	Describes the externally visible state of a programmable processor and the functions that the processor can perform. An ISA level model of a processor will execute any machine program for that processor and give the same results as the physical machine, as long as all input stimuli are sent to the ISA level model simulation on the same simulated clock cycle as they arrive at the real processor.
Interoperable	Two VHDL models of the same module are interoperable if one model can be substituted for the other without introducing errors into the system. Two VHDL models are also interoperable if they can be connected together as components without introducing errors into the system.
Leaf Module	A design entity that has no associated structural architecture body. Examples of possible leaf modules for a structural VHDL model include power supplies, analog circuit blocks, and digital logic gates.

Methodology	The body of rules, etc. employed by a [engineering] discipline; a particular procedure or set of procedures.
Object-Oriented (OO)	An Object-Oriented design approach consists of Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) techniques. OOA reflects the problem domain and the system's responsibilities within it. OOD reflects an implementation of the requirements.
Performance Model	A model which exhibits the measures of quality of a design that relate to the timeliness of the system in reacting to stimuli. Measures associated with performance include response time, throughput, and utilization.
Process - see Methodology	MMC will use the terms interchangeably.
Processor Element	A programmable device that is one of many that operate cooperatively through a network in a processor system.
Prototype Library Element	A hardware model or software primitive(application or OS service) which is a candidate for inclusion in the reuse library. It has not been fully tested, validated and documented. It is however available for use in the early stages of HW/SW codesign for high level architecture tradeoffs.
Register Transfer Level (RTL) Model	Describes a system in terms of registers, combinational circuitry, low level buses, and control circuits, usually implemented as finite state machines.
Structural Model	Represents a system or component in terms of the interconnection topology of the set of internal components.
Subsystem	A major component of a system. For RASSP, the signal processor is considered as a system. A subsystem represents a major component of the signal processor.
Synthesis	The process of creating a representation of a system at a lower level of abstraction from a higher level of abstraction. The synthesized representation should have the same function as the higher level representation.
System	Depending upon one’s perspective a system could represent a platform, sensor system, signal processor or processing board. For RASSP, a system represents a signal processing system.
System Configuration	The system configuration consists of the major subsystems which makeup the system. The major components of a RASSP system include the exciter, transmitter, antenna, receiver, signal processor and data processor.
System Definition	The process of analyzing customer requirements, performing functional analysis and system synthesis, and performing system level tradeoffs to determine the functional and performance specifications for each subsystem.
System Requirements Analysis	The process of analyzing and interpreting system requirements with the customer to refine the purpose and manner in which the user will operate the system.

System Synthesis	The process of performing top level system tradeoffs to allocate the functional requirements into performance and physical specifications for each subsystem.
Target Hardware	The hardware that is the result of the design process, as distinguished from hardware used in the design process.
Test Bench	A VHDL test bench is a collection of VHDL modules which apply stimuli to a module under test (MUT), compare the MUT's response with an expected output, and report any differences observed and expected responses during simulation.
Testability	An attribute of a design that allows detection and isolation of an apparent malfunction to a single (desired) replaceable item.
Validated Library Element	A prototype library element which has been designed for reuse, tested, validated and documented according to the standards defined for the reuse library.
Virtual Prototype	The set of simulation models that comprises a prototype processor. When exercised, the virtual prototype should behave (function and performance) as closely as possible to its physical counterpart.

Appendix C
Acronyms

A/D	Analog to Digital
ABBET	A Broad Based Environment for Test
ADP	Automatic Document Processing
AEP	Application-Environment Profile
AFAL	Air Force Avionics Laboratory
AIM	Application Interpreted Model
AIRST	Advanced Infrared Search and Track
ALC	Ascent Logic Corporation
AMPLE	Advanced Multi-Purpose Language
ANSI	American National Standards Institute
AP	Application Protocol
API	Application Programming Interface
ARL	Army Research Lab
ARM	Application Reference Model
ARPA	Advanced Research Projects Agency
ASEM	Application-Specific Electronic Module
ASIC	Application-Specific Integrated Circuit
AST	Architecture Selection Toolset
ASW	Anti-Submarine Warfare
AT&T	American Telephone & Telegraph
ATAG	ABBET Technical Advisory Group
ATD	Advanced Technology Demonstration
ATE	Automatic Test Equipment
ATM	Automatic Test Methods
ATPG	Automatic Test Pattern Generation
ATR	Automatic Target Recognition
ATS	Automatic Test Sets
BAA	Broad Agency Announcement
BDE	Block Diagram Editor
BDT	Berkeley Design Technology, Inc.
BFM	Bus Functional Model
BISI	Bi-Directional Schematic Interface
BIST	Built-In-Self-Test

BIT	Built-In-Test
BITE	Built-In Test Equipment
BNF	Backus-Naur form
BONES	Block-Oriented Network Simulator
BTD	Benchmark Technical Description
CAD	Computer-Aided Design
CAE	Computer-Aided Engineering
CALS	Computer-Aided Logistics Support
CAM	Computer-Aided Manufacturing
CAPE	Computer-Aided Parametric Estimating
CASE	Computer-Aided Software Engineering
CAT	Computer-Aided Test
CDE	Common Desktop Environment
CDEM	Customizable Debugging Environment for Multiprocessors
CDG	Cockpit Display Generator
CDMS	Computer-Aided Design Data Management System
CDR	Critical Design Review
CE	Concurrent Engineering
CECOM	Communication-Electronic Command
CEENSS	Continuous Electronics Enhancement Using Simulatable Specifications
CFAR	Constant False Alarm Rate
CFD	Control Flow Diagram
CFI	Computer-Aided Design Framework Initiative
CFI-DR	CAD Framework Initiative — Design Representation
CGS	Code Generation System
CHPC	Center for High-Performance Computing
CIS	Component Information System
CLMS	Component and Library Management System
CM	Communications Model
CMU	Carnegie Mellon University
CND	Cannot Duplicate
CNI	Communication, Navigation, Identification
COSE	Common Open Software Environment
COTS	Commercial Off The Shelf
CP	Command Program

CPP	Command Processing Program
CPU	Central Processing Unit
CSCI	Computer Software Configuration Items
CSV	Comma Separated Value
CUI	Common User Interface
DAG	Directed Acyclic Graph
DB	Database
DDMS	Design Data Management System
DEMVAL	Demonstration/Validation
DFD	Data Flow Diagrams
DFG	Data Flow Graph
DFT	Design-For-Testability
DICE	DARPA Initiative in Concurrent Engineering
DMA	Defense Mapping Agency
DMA	Direct Memory Access
DMM	Design Methodology Manager
DMP	Design Management Port
DoD	Department of Defense
DOM	CFI Standard for Design Object Management
DOTH	Design Object Type Hierarchy
DR	CFI Standard for Design Representation
DR-PI	Design Representation Procedural Interface
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
DSS	Decision Support System
DT&E	Development Test and Evaluation
DVE	Design Viewpoint Editor
ECM	Electronic Countermeasures
EDA	Electronic Design Automation
EDDC	Express-Driven Data Conversion
EDIF	Electronic Data Interchange Format
EDM	Enterprise Desktop Manager
EIF	Enterprise Integration Framework
EINet	Enterprise Integration Network
EPDM	Enterprise Product Data Management
EPI	Engineering Process Improvement
EPM	Enterprise Project Manager
ESPS	Example Signal Processor System
EW	Electronic Warfare

EXPRESS-G	Information Modeling Language - Graphical Representation
FDDI	Fiber Distributed Data Interface
FLIR	Forward-Looking Infrared
FMEA	Failure Modes and Effects Analysis
FMECA	Failure Modes and Effects Criticality Analysis
FO	Fiber Optic
FPGA	Field-Programmable Gate Array
FSED	Full-Scale Engineering Development
FSPS	Fictitious Signal Processor System
GBR	Ground-Based Radar
GDB	GNU Debugger
GEDAE	Graphical Entry Distributed Application Environment
GES	Government Electronic Systems
GFE	Government-Furnished Equipment
GFI	Government-Furnished Information
GFLOPS	Billion Floating-Point Operations per Second
GIPS	Graph Instantiation Parameters
GOTS	Government Off the Shelf
GRAIL	Graph Translator
GRED	Graphical Editor
GUI	Graphical User Interface
GW	Gateway
HCI	Human-Computer Interface
HDI	High-Density Interconnect
HDL	Hardware Description Language
HDS	Hardware Design System
HI-TEA	High-Level Test Strategy and Economics Advisor
HOL	High-Order Language
HSIM	Heterogeneous Simulation Interoperability Mechanism
HTC	Honeywell Technology Center
HTML	Hyper Text Markup Language
HW	Hardware
I/NFM	Intergraph Network File Manager
IC	Integrated Circuit
ICD	Interface Control Document

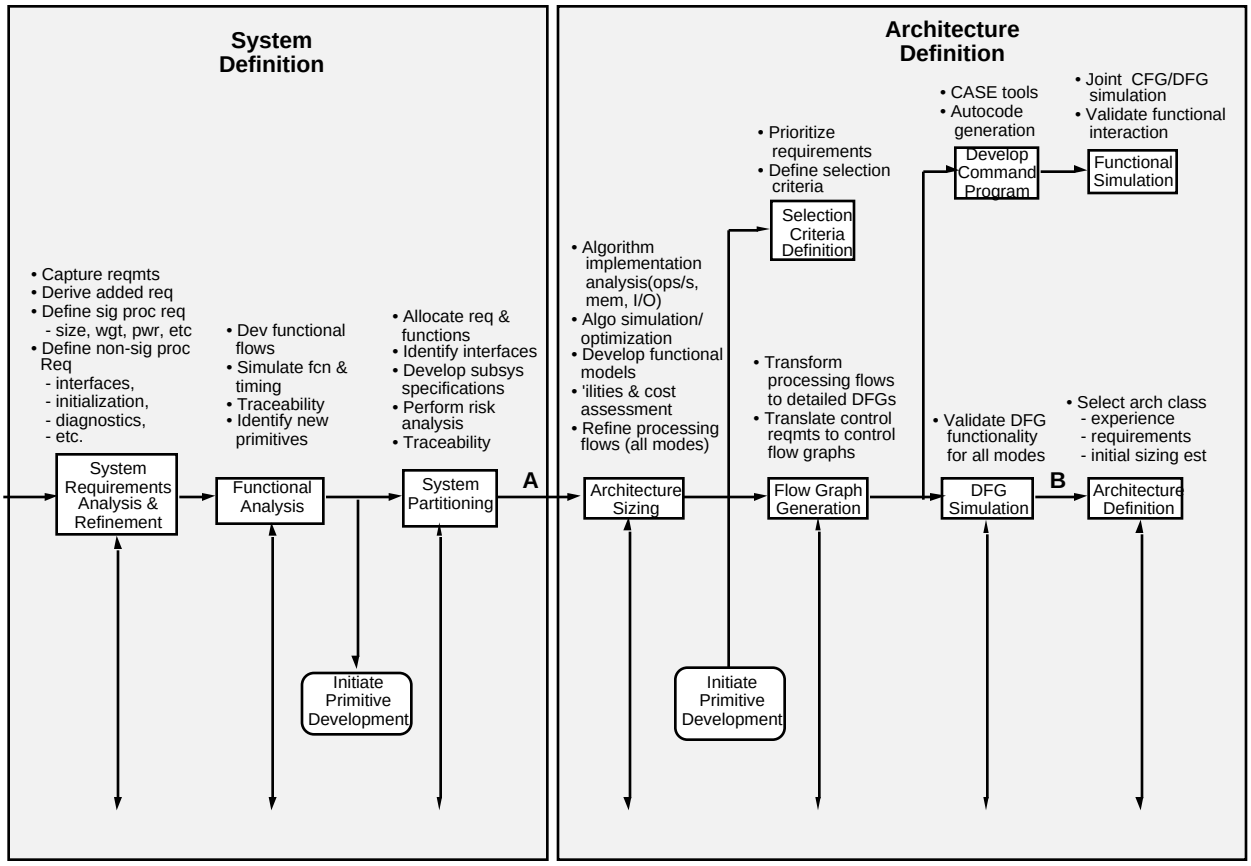
ICNI	Integrated Communication, Navigation Identification
ICOM	Input, Control, Output, Mechanism
IDAS	Integrated Design and Assessment
IDDq	Quiescent Source to Drain Current Test
IDEF	Integrated Computer-Aided Manufacturing DEFinition
IDEF	International Data Exchange Format
IEEE	Institute of Electronics and Electrical Engineers
IGES	Initial Graphics Exchange Specification
ILS	Integrated Logistics Support
IM	Information Model
IPC	Inter-Process Communication
IPDT	Integrated Product Development Team
IPPD	Integrated Product/Process Development
IR	Infrared
IRST	Infrared Search and Track
ISA	Instruction Set Architecture
ISO	International Standards Organization
ITC	CFI Standard for Intertool Communication
JCALS	Joint Computer-Aided Logistics Support
JIAWG	Joint Integrated Avionics Working Group
JRS	JRS Research Laboratories
JTAG	Joint Test Action Group (IEEE 1149)
JTIDS	Joint Tactical Information Distribution System
LAN	Local Area Network
LCC	Life Cycle Cost
LMG	Logic Modeling Group, Synopsys Inc.
LMS	Library Management System
LRM	Line Replaceable Module
LRU	Line Replaceable Unit
LSA	Logistic Support Analysis
LV	Logic Vision Software, Inc.
MANTECH	Manufacturing Technology
MATLAB	Software Tool-Mathematical Analysis of Functions

MCC	Microelectronics and Computer Technology Corporation
MCM	Multi-Chip Modules
MFBIT	Multi-Feature Bayesian Intelligent Tracker
MFLOPS	Million Floating-Point Operations per Second
MGC	Mentor Graphics Corporation
MIMD	Multiple Instruction, Multiple Data
MIT	Massachusetts Institute of Technology
MMC	Martin Marietta Corporation
MMIC	Monolithic Microwave Integrated Circuit
MOE	Measure of Effectiveness
MOU	Memorandum of Understanding
MPID	MIMD Primitive Interface Description
MRE	Manufacturing Resource Editor
MSDA	Multi-Chip Systems Design Advisor
MSI	Management Services, Inc.
MTBF	Mean Time Between Failures
MTI	Moving Target Indicator
MYA	Model Year Architecture
NEP	Node Execution Parameter
NFM	Network File Manager
NGCR	Next-Generation Computing Resources
NIIIP	National Industrial Information Infrastructure Protocols
NRL	Naval Research Laboratory
NetSyn	Network Synthesis System
ONR	Office of Naval Research
OO	Object-Oriented
OOA	Object-Oriented Analysis
OS	Operating System
OSI	Open Systems Interconnect
PCA	Printed Circuit Assembly
PCB	Printed Circuit Board
PCTE	Portable Common Tools Environment
PDCM	Product Data Control Module
PDES	Product Data Exchange using STEP
PDM	Product Data Manager
PDP	Product Data Package
PDR	Preliminary Design Review
PDT	Product Development Team

PE	Processing Element
PGM	Processing Graph Method
PGM/DFL	Processing Graph Method/ Data Flow Language
PGSE	Processing Graph Simulation Environment
PID	Primitive Interface Description
PIDGen	Primitive Interface Description Generation
PIE	Performance Instrumentation Environment
PLD	Programmable Logic Device
PM	Process Model
PMB	Performance Measurement Baseline
PML	Process Modeling Language
PMO	Program Management Office
PreAMP	Pre-Competitive Advanced Manufacturing Process
PRICE	Parametric Review of Information for Costing and Evaluation
PRR	Production Readiness Review
PWA	Printed Wiring Assembly
PWB	Printed Wiring Board
QA	Quality Assurance
R&M	Reliability and Maintainability
RAM/ILS	Reliability, Availability, Maintainability/Integrated Logistics Support
RASSP	Rapid Prototyping of Application-Specific Signal Processors
RDBMS	Relational Database Management System
RDD-100	System Design CAD tool - Ascent Logic
REDM	RASSP Enterprise Data Module
REMR	RASSP Enterprise Model Repository
RIC	Rockwell International Corporation
RL	Rome Laboratory
RMWG	RASSP Methodology Working Group
RNI	Reconfigurable Network Interface
RRDM	RASSP Reuse Data Manager
RSS	Reusable Software System
RTL	Register Transfer Level
RTM	Requirements Traceability Matrix

RTOK	Retest OK
RTS	Run-Time System
SAL	Signal Processing Algorithm Library
SCRA	South Carolina Research Authority
SDD	Software Development Document
SDR	System Design Review
SE	Switching Element
SEM	Standard Electronic Module
SEMP	System Engineering Management Plan
SEMS	System Engineering Management Schedule
SES	Systems Engineering Station
SIMD	Single Instruction, Multiple Data
SM	State Model
SNR	Signal-to-Noise Ratio
SOW	Statement of Work
SP	Signal Processor
SPEAR	Signal Processing Environment Aiding debugging for RASSP
SPGN	Signal Processing Graph Notation
SPW	Signal Processing Workstation
SQL	Standard Query Language
SRAM	Static Random Access Memory
SRR	System Requirements Review
SRS	System Requirements Specification
STA	Science and Technology Associates
STEP	Standard for The Exchange of Product Data
STS	Self Test Services
SVI	Standard Virtual Interface
SW	Software
SWAP	Size, Weight, and Power
T&E	Test and Evaluation
T&M Bus	Test and Maintenance Bus
T/R	Transmit/Receive
TAP	Test Access Port
TBD	To Be Determined
TBR	To Be Resolved
TCL/TK	Tool Command Language/Tool Kit
TCP/IP	Transmission Control Protocol/Internet Protocol
TES	CFI Standard for Tool Encapsulation

TI	Texas Instruments
TIGER	Testability Insertion and Guidance Expert for RASSP
TIS	Test Information Sheet
TK	Toolkit for X11 Window System
TMC	Test and Maintenance Controller
TPM	Target Primitive Map
TPS	Test Program Set
TRD	Test Requirements Document
TRP	Technology Reinvestment Program
TRSL	Test Requirements Specification Language
UUT	Unit Under Test
VARs	graph VARiables
VHDL	VHSIC Hardware Description Language
VHF	Very-High Frequency
VHSIC	Very-High-Speed Integrated Circuits
VITA	VFEA International Trade Association
VLSI	Very-Large-Scale Integration
VPS	Virtual Prototyping System
VTEST	Virtual Test
WAN	Wide Area Network
WAVES	Waveform & Vector Exchange
WBS	Work Breakdown Structure
WEC	Westinghouse Electric Corporation
WL	Wright-Patterson Laboratory



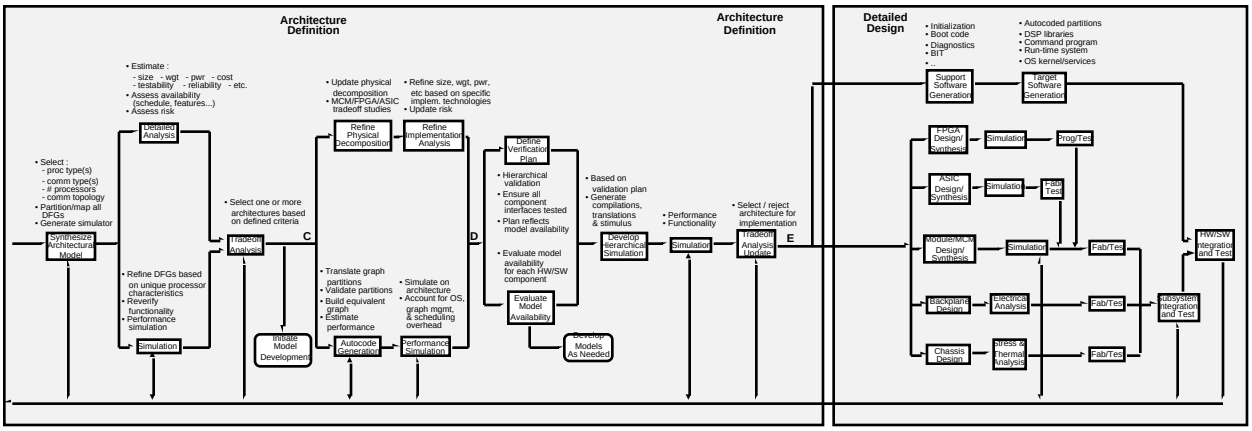
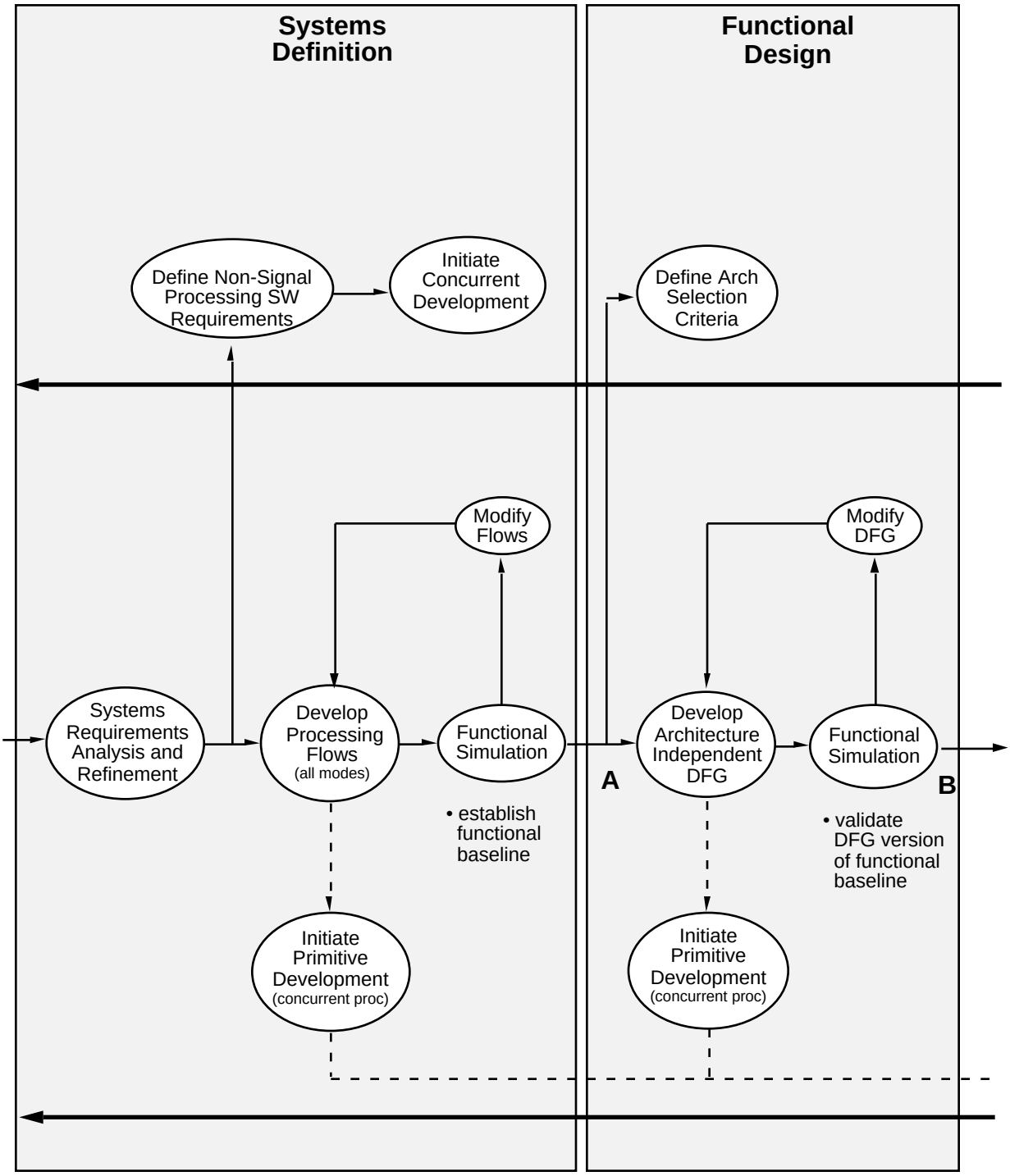


Figure A-1. Design process flow from systems to detailed design.


Systems Definition
Functional Design
Define Non-Signal Processing SW Requirements
Initiate Concurrent Development
Define Arch Selection Criteria
Modify Flows
Modify DFG
Systems Requirements Analysis and Refinement
Develop Processing Flows (all modes)
Functional Simulation
Develop Architecture Independent DFG
Functional Simulation
A
B
• establish functional baseline
• validate DFG version of functional baseline
Initiate Primitive Development (concurrent proc)
Initiate Primitive Development (concurrent proc)

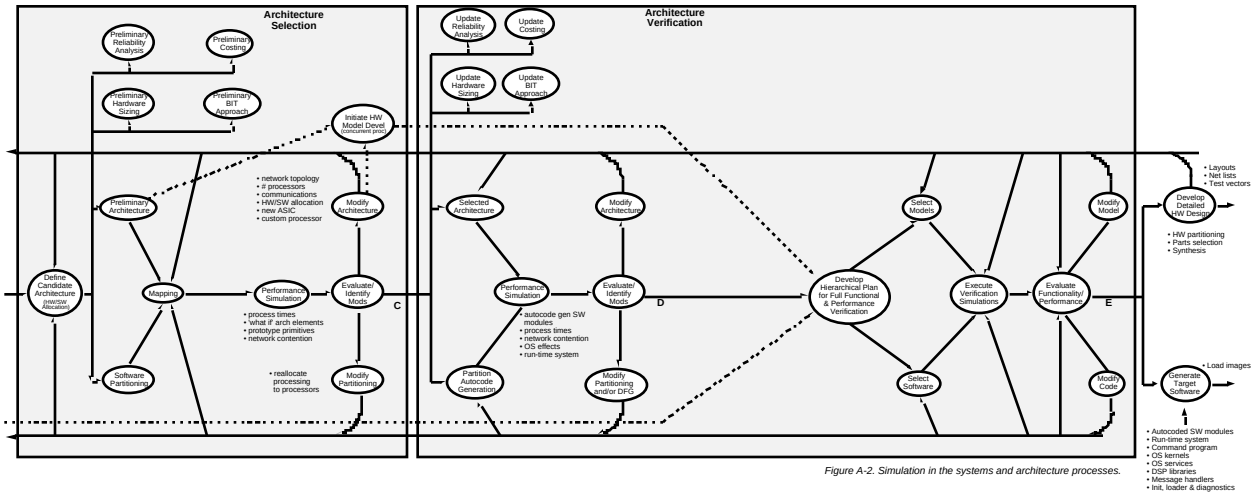


Figure A-2. Simulation in the systems and architecture processes.