Chapter 9

Embedded Systems Design and Development – Hardware – Software Co-Design

THINGS TO LOOK FOR...

- Things to consider in a design.
- The traditional product development life cycle
- The Co-Design methodology and life cycle
- An overview and motivation for the Co-Design methodology in the embedded world.
- The goals and steps to design in the Co-Design methodology.
- The need to understand the environment and the system being designed.
- The difference between requirements definition and specification.
- Formulating a Design Specification.
- Techniques and major phases of hardware / software Co-Design.
- Motivation for and objective when partitioning a system.
- Coupling and cohesion and why they are important.
- The differences between functional and architectural models of a system.
- Modeling under Co-Design.
- Modeling tools and methodologies.
- Motivation for and timing of static and dynamic analysis of a design.
- Co-Simulation, Co-Synthesis, and Co-Verification.
- Capitalization and reuse of designs.
- Archiving the project
- Requirements traceability.

9.0 Introduction

In this chapter, we will study the major phases of the Co-Design development process for embedded systems. The more detailed aspects of that process will be explored in conjunction with the design and test of the specific hardware and software elements of the system.

We will learn that design is the process of translating customer requirements into a working system and that the complexity of contemporary systems demands a formal approach and formal methods. Working from a formal specification of a problem, we will look at ways of partitioning the system as a prelude to developing a functional design. We will then examine the process of mapping functional model on to an architectural structure and ultimately to a working prototype. To help to ensure the robustness of the ultimate product, we will illustrate how to critically analyze the design both during and after development.

We will also look at several other important considerations in the design lifecycle including intellectual property, component/module reuse, and requirements management and the archival process.
As we begin to think about a new product or adding new features to an existing one, we must look at things from many different points of view. The most important of these perspectives is the customer’s since he or she finances the development of the product either directly through an agreed upon contract or indirectly through a purchase. The best design of little value if no one is willing to buy. So, we pose the question: What kinds of things should be considered?

If we look at products, we must know how to measure costs and features. We must be able to identify and distinguish between real and perceived needs. Too often when we talk with customers about new products, the essential “requirement” in the next generation product is that which was missing when a problem arose this morning. Another important consideration is the environment in which the system will be operating.

It is important to learn to make market and technology trade-offs. Several years ago the following very simple table in Figure 9.0 was proposed. Taking old technology into and old markets is a reasonable and safe strategy. These are the niche markets and often provide support and evolutionary growth for products that are no longer in a vendor’s mainstream offering. Taking new technology into new markets is difficult and risky. At the same time, the rewards can be very high. The personal computer is a very good example. Xerox and Apple both had limited success with their early offerings. The people and the full technology were simply not ready. Taking new technology in to an existing area or existing technology in to a new area is easier. At least one portion of the problem - the market or the technology - is well understood and well developed.

We must understand the importance of deadlines and costs. Product development is based upon a (directly or indirectly) negotiated contract between us and the customer(s). Failure to respect development and delivery costs or schedules leads to loss of sales, market share, and credibility.

We also must always consider security, reliability, safety, and quality in the products we design. We learned about these in Chapter 8. Beyond obvious need to work properly, the product must be robust. Simply put, “Does it do what it is supposed to?” and “How does it behave with unexpected inputs?” Robust means much more than this, however. Robust also implies that the system performs even if it is partially damaged, or under extreme temperature conditions, or if it is dropped. If a product does what it is supposed to do but is fragile and buggy, the product is not robust.
The documentation we produce to accompany the product must be clear and understandable. The product must be easy to use - intuitive rather than counter-intuitive. Post sales support, including the correction of bugs, is very important. Lack of quality has two costs. The first cost is obvious and immediate, the cost to repair, which is often small. The second a hidden cost, the loss of customer confidence and sales – and it can be very large. Once confidence lost very difficult to regain.

9.1 SYSTEM DESIGN AND DEVELOPMENT

System design and development is a challenging problem. What makes it fun and exciting is that there is a very large creative component to it. There are no rules, no steps to follow to make one creative. There are, however, a large collection of rules to ensure the opposite. Consider a new child. Each comes into this world, eyes wide open with a million questions. Why is the sky blue? Why is the sun yellow? Why can not we see the air? Where does air come from anyway? We put them in school. We teach them the rules. Walk into any group of little ones and ask, how many of you can sing? How many of you can draw? Almost every tiny hand leaps up. Go into any similar group of adults and ask the same questions. Everyone is suddenly fascinated with their shoes. One hand may come slowly up. Why? We place too many restrictions on our thinking. Sure, we may need 10 million dollars worth of electronic equipment to give our voice perfect pitch, but, so what. We need to remove artificial restrictions that we impose on our thinking.

Look at the little ones drawing or coloring. What do we tell them? No, people aren’t purple. Cows can’t fly. Fish don’t have legs - anymore. Oh, and by the way, always color in the lines….and Let us also learn how to be creative.
9.1.1 Getting Ready – Start Thinking

Ok, Let us start. Driving is always a good place to begin. The rules are easy. Keep the yellow line on your left and the white on your right – except in Britain and several other places. Now the chance to be creative. In the autumn in the northern parts of the world, the days are warm, but the nights start getting colder. Often there is a bit of fog that makes an appearance as well. By the morning, the fog and chill have combined to give a very fine glaze of ice on the road. We call this black ice; it gives us the opportunity to be creative. Hop in the car and race out onto the road. What’s this nonsense about staying in the lines?

Now perhaps we have decided that maybe we can be just a little creative, let us begin to explore. As we begin thinking about a new design, we will discover that there are a lot of things to be considered. The problem may not always be what it seems at first blush. Roger van Oech in *A Whack on the Side of the Head*, Warner Brooks (1983), says "Always look for the second right answer." He’s right. As we begin, it is important to understand the problem to make sure that we are solving the right problem. Consider the illustration in Figure 9.1. Which one is the correct image? Is it the old lady or the young one?

When we begin trying to solve a problem, it is important to talk with everyone involved; to listen to different opinions; to see how the design might affect the people who have to work with it. We have to take the time to look at different views of the problem; to look at it from both the inside and the outside. Based upon our view, we can have a couple of different interpretations of the following problem presented in Figure 9.2. Are we building a goblet or are we building two statues?

There will always be occasions in which we have too much information, too many opinions, or too many details. Remember the old expression of not being able to see the forest for the trees. The same holds true as we begin trying to understand a problem during the early stages...
of a design. Look at this next drawing in Figure 9.3. What do you see? This interesting design; it looks perhaps like a snowflake. This is a case in which we have too much information; now remove some of the information as in Figure 9.4. If we take a more abstract view of the problem, the solution is easier to see.

Now that we have a start, Let us look at the design problem. Let us look at each design as a chance to explore.

9.1.2 Getting Started

Designing and developing embedded systems does raise some interesting challenges and does require a large number of decisions. Some of those decisions require knowledge about the problem, others about the tools and techniques that may be available, and still others choose methods for approaching the solution. There will often be still more things to think about that are not related to the technical part of the problem at all. The collection of these things that we do as we move from requirements to application is often called the product development life cycle.

Like so many other things in life, there are probably as many different product development life cycle models as there are people designing these systems. Who said there isn’t any creativity? Each of these models has its supporters and each also has it group of detractors. The goal in the next few pages is to introduce some of the more important things that one should think about when executing a design, present several of the more common life cycle models, and to present some guidelines for things that have worked on successful projects. Despite what they tell you, there are no hard and fast rules – well, perhaps there are a few: learn a lot with each project, have fun, and do the job right, to the best of your ability. Let us get started.

9.2 LIFE CYCLE MODELS

The product development life cycle of an embedded application is purely a descriptive representation. It breaks the development process into a series of interrelated activities. Each activity plays a role of transforming its input (specification) into an output (a selected solution). The steps are organized is done according to a design process model – the life cycle model. Formality in design provides the structure for a development that lets us creatively explore the design while using the tools to manage some of the more mechanical issues. We use the structure as an aid rather than something that encumbers design.

As we have commented already, the related literature presents a variety of different approaches and models. At the end of the day, all have the same basic goal, however; they all have similar phases. Perhaps, we could more accurately say that they all have similar needs or goals or objectives. These needs are very simple as shown in Figure 9.5. Several of the historically more common models or approaches are listed in Figure 9.6.

- Waterfall
- V Cycle
- Spiral
- Rapid Prototype

Figure 9.6 Common Life-Cycle Models
Today, we are continually developing new ones. But whichever model we choose, the most important point is to understand the meaning and intent or objective of each of the phases or steps in the process. Understand the deliverables for each step as well as the necessary outputs and inputs that are required to move, conclude, or to enter each phase in the selected model. Then follow those and don’t take shortcuts. We will look briefly at each of these four models momentarily. Before we do so, Let us look at another model that fits just about any phase of engineering; it looks something like that in the accompanying Figure 9.7.

This is called the hockey stick model or curve; its shape is strongly suggestive of where the name originated. We have talked about how important it is to address reliability and safety early in the requirements and design specification phases of the life cycle. The hockey stick curve, shown in Figure 9.7, provides an intuitive feel as to why. If we label the horizontal axis as time and the vertical one as cost, and apply it here, we see that the longer we delay in addressing those issues, the greater the cost will be. Cost is not limited to be money alone.

Let us begin with the Waterfall model. Use your artistic creativity here. Its name evokes its sound, which evokes the philosophy and approach engendered in the model.

9.2.1 The Waterfall Model

The Waterfall model represents a cycle – specifically, a series of steps appearing much like a waterfall, sequentially, one below the next as we see in Figure 9.8.
The steps are:

- Specification
- Preliminary Design
- Design review
- Detailed Design
- Design review
- Implementation
- Review
Together, these capture each of the needs that we identified earlier. Successive steps are linked in a chained manner. Such a linking tends to say: *Complete this phase and go on to the next.*

Observe that each phase is also connected back to the previous phase. That reverse connection provides an essential verification link backwards to ensure that the solution (in its current form) agrees with and follows from the specification. With the Waterfall model, the recognition of problems can be delayed until later states of development where the cost of repair is higher (the hockey stick curve). The Waterfall model is limited in the sense that it does not consider the typically iterative nature of real-world design.

**9.2.2 The V Cycle Model**

The *V Cycle* is similar to the Waterfall model except that it places greater emphasizes on the importance of addressing testing activities up front instead of later in the life cycle. Each stage associates the development activity for that phase with a test or validation at the same level. Each test phase is identified with its matching development phase as we see in Figure 9.9.

In the diagram, we have

- Requirements ↔ System/Functional Testing
- High-Level Design ↔ Integration Testing
- Detailed Design ↔ Unit Testing

We identify the major phases of a project life cycle across the top of the drawing. These phases extend from specification to customer delivery and postdelivery support. If one follows the sequence down the left hand side of the drawing, one can see that the specification and design procedure utilizes a top-down model whereas implementation and test proceed from a bottom-up model as is reflected on the right hand side of the drawing.

![Figure 9.9 The V Life-Cycle Model](image-url)
It is evident that each development activity builds a more detailed model of the system and that each verification step tests a more complete implementation of the system against the requirements at that phase. The development concludes the design and design-related test portion of the development cycle of the system with both a verification and a validation test against the original specification.

9.2.3 The Spiral Model

The Spiral model was proposed and developed by Barry Boehm, *A Spiral Model of Software Development and Enhancement*, (Computer, May 1988). A simplified version of that model is presented in Figure 9.10.

![Figure 9.10 The Spiral Life-Cycle Model](image)

The model takes a risk-oriented view of the development life cycle. Each spiral addresses the major risks that have been identified. After all the risks have been addressed, the Spiral Model terminates as did the Waterfall and V models, in the release of a product.

Like the earlier models, the Spiral Model begins with good specification of the requirements. It then iteratively completes a little of each phase. Its philosophy is to start small, explore the risks, develop a plan to deal with the risks, and commit to an approach for the next iteration. The cycle continues until the product is complete. Boehm’s model contains a lot more detail that the one presented in Figure 9.10. In both cases, each iteration of the spiral involves six steps,

The Spiral model is an improvement on the Waterfall and V models because it provides for multiple builds as well as several opportunities for risk assessment and for customer
involvement. On the negative side, it is elaborate, difficult to manage, and does not keep all developers occupied during all of the phases.
9.2.4 Rapid Prototyping - Incremental

The Rapid Prototyping model is intended to provide a rapid implementation (hence the name) of high-level portions of both the software and the hardware early in the project. The approach allows developers to construct working portions of the hardware and software in incremental stages. Each stage consists of design, code and unit test, integration test, and delivery. At each stage through the cycle, one incorporates a little more of the intended functionality.

The prototype is useful for both the designers and for the customer. For the designer, it enables the early development of major pieces of the intended functionality of system. By doing so, it helps to establish and verify the structural architecture as well as the data and control flow through the system. Such an approach permits one to identify major problems early (the hockey stick curve again).

The customer benefits by having the opportunity to work with a functional unit much earlier in the development cycle than with any of the three previous models. The customer can use the prototype in the intended context to provide feedback to the designers about any problems with the design.

Such feedback is a critical aspect of the approach because it encourages backwards or reverse flow through the process. It can be used to refine or change the prototype in order to correct the identified problems and to ensure that the design meets the real needs of the customer.

The prototype can be either evolutionary or throw away. It has the advantage of having a working system early in the development process. As noted, problems can be identified earlier and it provides tangible measures of progress. To be effective, however, the rapid prototyping approach requires careful planning both at both the project management level and the designer’s level.

Be careful how the prototype is used,

Caution: The prototype should never turn into the final produce

Let us now move into the design process. Design begins with the real world where we are trying to solve problems in order to make our life easier.

9.3 PROBLEM SOLVING

As we have learned, embedded systems touch almost every aspect of our daily lives. Such applications can comprise thousands of lines code, large heterogeneous collections of microprocessors, VLSI components, and array logics. Such collections may also include a variety of legacy components; we can not afford to redesign each new system from ground up. They may utilize a mixture of different forms of control such as event driven, reactive, time based, or data flow constructs to orchestrate a mixture of different technologies to solve a customer’s problem.

Components or IP may come from variety of different sources for which we may not have access to internals. Those components may be distributed around an office or around the world. Systems on a chip (SOC) are becoming increasingly more common and we are
accelerating towards networks on a chip (NOC). Artificial Intelligence (AI) and the Internet of Things (IOT), with all their pluses and minuses are also becoming a reality. Yesterday, embedded system design drew primarily from the field of digital logic augmented by some support from a new field called software engineering. Today, the boundaries of embedded design grow increasingly fuzzy as knowledge and skills from virtually all fields of engineering and computer science are drawn upon and have become essential.

Our designs are often subject to contradictory constraints on cost, size, power, speed, safety, and reliability. At same time, they are becoming increasingly sophisticated and powerful as well as larger yet smaller. The ubiquitous microprocessor – or multicore processor, coupled with our advances in the design and development of large-scale integrated circuits supported by our progress and developments in the software fields are creating opportunities to explore designs that push the edges our current science and technology. Today the successful deployment of embedded applications requires new approaches and new tools that we can utilize to address and manage these complexities. Yet the typical developer often continues to use methodologies and principles that can be quickly overwhelmed by the complexities and demands of modern systems. The problem is growing worse, particularly with multicore devices, increasing speeds, and (widely) distributed systems. As sizes decrease and speeds surge, the effects on the quality of signals from parasitic devices rooted in fundamental physics are becoming an increasingly challenging issue. Yesterday, we could design then build our system on our lab bench. Today with designs approaching or exceeding the million transistor level; traditional methods are no longer viable.

The modern creative design and development process begins with an abstracted notion of the system to be built. Hopefully, we learned years ago that the first step to design is not to grab the nearest keyboard or processor and start hacking out code or wiring parts together. With today’s complex systems, planning and thought before starting are essential to any successful, reliable, and safe design. Computer based tools and methods have become an essential part of that process.

As a first step, we begin with a set of requirements usually stated in text form. The goal is to map those requirements - the real world - through a series of transformations into a solution - the abstract world. During the design process, we move from the concrete, real world into the abstract. These steps comprise what we describe as good design engineering practices. If one takes the central elements from each of the potpourri or life cycle models, one finds that good system designers and successful projects generally proceed using a minimum of six steps listed in Figure 9.12.

- Requirements definition
- System specification
- Functional design
- Architectural design
- Prototyping
- Test

Figure 9.12   Six Steps to a Successful Design
The formality of each step depends upon the complexity of the end product. If one is working alone or with several others in your own company on a smaller project, a white board in the center of the garage can often suffice. If one is orchestrating a project that includes developers, manufacturers, and regulations in several countries around the world, (which is becoming increasingly common today) the need for formality increases. When working with each of these phases of a product life cycle, one must remember that they are guidelines; collective best practices. They are not a checklist to a successful project; and they are not exhaustive.

Today the contemporary design process must also enforce IP (intellectual property), capitalization and reuse at every design stage. The glory days of Bob Widler (the father of the op amp) lecturing about the fundamentals of integrated circuit design in the bars of Silicon Valley are long gone. One must also consider traceability in both the forward and reverse directions. Traceability captures the relationships between requirements and all subsequent design data and helps in managing requirements changes.

We have taken our first steps into the embedded systems development cycle. Often our view on the journey focuses on the hardware. Working with the six steps, Let us now bring the in software side and integrate the concurrent development of both the hardware and the software into the methodology called Co-Design. We begin with an overview of such a process.

9.4 HARDWARE – SOFTWARE CO-DESIGN

In the earlier sections of this chapter, as a foundation for our next steps, we introduced and examined several of the more commonly used development cycles. For many years, the traditional design approach has followed these classical models. Underlying most such models has been the philosophy in which we design the hardware components, then design the software components, and then bring the two together. Often, the hardware components significantly lagged the software components and yet the software had to fit the hardware and correct any errors in the hardware design that may be discovered late in development and test process. The hockey stick curve makes another appearance.

Today’s designs are continually increasing in complexity, decreasing in size, operating at higher frequencies, and utilizing a growing breadth of exciting new technologies. Integrated circuit mask costs are in the millions of dollars; the cost of an error in time and dollars can be significant. Spending time testing and debugging later in the development cycle where cost of change is higher (hockey stick curve again) can add to the problem. The problem is only partially mitigated with the introduction of programmable logic devices (PLDs). Today, the evolving world of design is demanding tools and support that can match the challenges of new and advanced product features that can meet customer expectations.

9.4.1 The First Steps

Today, the Co-Design methodology, which builds on ideas underlying earlier models, is bringing a fresh view to the world of embedded design. The approach supports the iterative combined and ‘simultaneous’ design, development, and test of the hardware and software components with the objective of meeting and optimizing system-level requirements through well-reasoned trade-offs between these components. Following the six steps to successful
design, the key points in the process are to (iteratively) specify, design, develop, and test both the hardware and software aspects of system concurrently to meet required performance and functional objectives. The goals, as with any design or design process, are to increase productivity – reduce the design cycle time, improve product quality, security, reliability, and safety, and for tomorrow’s designs, of one pass silicon on ASICs or custom ICs.

### 9.4.2 Traditional Embedded Systems Development

Before we look at the Co-Design methodology in depth, let us start with a quick overview of the traditional embedded systems development process. As we know, most embedded systems share a common structure and common development cycle. Through such a cycle, the embedded design is developed. Figure 9.13 gives the high-level flow and identifies the major elements of such a cycle. Specifically, the hardware design involves the design of the components, the printed circuit boards (which are becoming an increasing challenge), and the system. The software design may entail the design of the high level, assembly language, and machine code. Work at the assembly language level requires detailed knowledge of the

![Figure 9.13 Traditional Embedded Development Cycle](image-url)
microprocessor architecture and its register structure. The hardware or software constituents may also include legacy components.

Immediately following formulation of the high-level architecture, the contemporary approach has been to break the system into hardware and software components. Each component would be developed separately; often by separate teams. Generally, hardware development lagged the software development. Lead times on parts and the fabrication of custom circuitry slowed the process but had to be finished first, before software was introduced and integrated.

Such an approach limits hardware-software trade-offs. Interactions between them are determined or discovered later in development. Late integration led to the philosophy of ‘let the software fix the hardware problems…we can send out updates later’, potentially poor quality designs, last second modifications, slipped schedules, costly design changes, the need for routine updates to correct problems over the product’s lifetime, and possibly cancelled projects.

The development cycle illustrated in Figure 9.13 reflects the potential for a high degree of concurrent hardware and software development following the specification of the system architecture. Such concurrency demands greater co-operation between all designers. Shorter times to market mitigate against an extended trial and error approach although such an approach actually worked quite successfully during the Soviet space program. The traditional development cycle puts an increased demand that errors be identified and corrected early, that requirements be very well established and understood prior to the start of the design and that a predictable schedule is essential. Under such constraints, the increased reuse of legacy components could help to reduce the design and development time. Such observations naturally lead to a Co-Design approach.

Let us now look at the various elements of the Co-Design process in a bit more detail.

9.5 HISTORY

As we begin to explore the Co-Design process, we will start with a look at where we came from. Early computers were known as complex instruction set computers; the CISC architecture. Work at IBM plus other places led to simpler architectures and a significant reduction in the numbers and complexity of instructions. These machines because known as reduced instruction set computers or the RISC architecture. Such a change simplified the design of both the hardware and software thereby increasing the speed of computation and simplifying and improving performance and flow of control through the machine. In particular, the flow associated with context switching. Today’s ARM (Advanced Risc Machine) processor evolved from this early work.

Simultaneously, work continued on compilers for such machines. Developed ‘independently’ or serially, such a process led to two designs – hardware followed some time later by the software, neither of which was optimal. The thought occurred that if the development and optimizations could be executed simultaneously it would be possible to improve both.

The fundamental concepts behind Co-Design that have been around for approximately 20 to 30 years started to gain traction around 10 – 15 years ago with the first international
workshop. The approach evolved from thoughts and research in several areas. On the hardware side, the recognition that microprocessor based systems were becoming a growing and increasingly important area for traditional IC and system designers. On the software side, the development and growth of software engineering and the recognition that software was becoming an integrated and essential component in future chip and system designs. The development of both the hardware and the software were supported by work in synthesizing designs from behavioral models. Models, and modeling have become a significant and integral component of the Co-Design process.

The early objectives of the Co-Design methodology were to gain control of the design of the hardware and software based systems and to ensure greater predictability in meeting initial goals and requirements. Supporting such objectives was the desire to give designers tools to assess and verify that the delivered system met the specified speed, power, cost, reliability, performance, and complexity goals while enabling them to explore and evaluate alternative designs at the detailed level without having the cost of a full implementation. As noted, essential to the Co-Design methodology was the use of computer-based tools and models. –

The Unified Modeling Language (UML) and the Structure analysis and Design methodology have become very good tools contributing to and supporting that process.

9.5.1 Advantages of the Co-Design Methodology

Co-Design permits both the hardware and the software to influence the design during early stages of development and supports a continual verification of the design and design alternatives throughout the development cycle. Co-Design supports and encourages models and the interoperability of hardware and software design tools throughout process, particularly during architecture definition. It enables greater exploration of design trade-offs and more interesting architectures. Reuse is integral part of the process specifically of component’s whose behaviors have been previously verified and co-verified as part of existing design. Such reuse enables one to reduce integration and test times leading to quicker time to market and to approach the boundary of one pass silicon. Be careful here, however. We may be moving a design flaw in an original design where it was not a problem forward to where it may become one. The European Space Agency’s Arianne 5 rocket is one such example.

9.6 CO-DESIGN PROCESS OVERVIEW

The hardware / software Co-Design approach to embedded systems design suggests tools and approaches / methodologies that integrate the design and development of both the hardware and the software components. Co-Design focuses on the following major areas of the design process:

- Ensuring a sound hardware and software specification as input to the process.
- Decomposing the system into functional modules.
- (Iteratively) partitioning and mapping the functional modules onto the hardware and software.
- Based upon the hardware-software partition, formulating the architecture for the system to be designed.
• An iterative process of hardware-software synthesis, simulation, and verification.

Following the six steps to successful design, the Co-Design process comprises number of subprocesses. The Unified Modeling Language (UML) echoes many of these same ideas. Included among these are:

• **System Specifications – Requirements and Design**
  Develop and validate the specification for both sets of components. This is an essential and core aspect of the approach.

• **Functional Decomposition**
  Partition the system into major functional blocks.

• **Partitioning – Co-Design**
  Partition and map the functional blocks onto hardware and software components; define and refine the inter process communication.

• **Modeling – Architecture**
  Develop a system architecture from the hardware and software models that takes into consideration both aspects and works to model the complete system then validate those models. Synthesize hardware – software interfaces.

• **Co-Synthesis – Prototype**
  Synthesize both the hardware and software components from higher-level models. Software synthesis will target specific tasks to designated hardware components and hardware synthesis will decompose computation steps into clock cycles then bring two pieces together under co-simulation.

• **Co-Simulation – Prototype**
  Simulate the architecture using the modeled hardware running real software. Later we migrate towards real hardware. The goal is to keep both simulations – hardware and software – synchronized to ensure proper performance in the ultimate target platform.

• **Co-Verification – Test**
  Verify that the architecture meets the specification.

• **Repeat the Process**
  Explore alternate partitions as necessary or appropriate.

Such processes require both coarse and fine-grained computer based tools and support. In particular, the processes of co-synthesis, co-simulation, and co-verification are limited by the processing power of our tools. What advantages and disadvantages do multicore processors bring to this need for tools?
9.7 THE CO-DESIGN PROCESS

We now move from the traditional embedded development cycle given in Figure 9.13 to that in Figure 9.14. The latter figure presents the high-level flow through the Co-Design development cycle and identifies the major elements of the process. The focus of the methodology is on the design and synthesis aspects. Its domain is indicated in the boxed portion. It is important to note, that as with the design of any kind of well-conceived and developed system today, we must begin by identifying the stated requirements and capturing these in a formal specification. Such recognition holds, independent of any approach used.

Examining the Co-Design methodology a little closer, we can list the essential concepts that underlie and guide the process:

**Ideals**

- Transparent methods of modeling hardware and software.
- Transparent design and analysis techniques.
- Interoperability of tools and methods.
- Seamless migration of pieces of functionality between hardware and software.
- An iterative design approach.
- The ability to quickly evaluate a number of different hardware-software partitions and tradeoffs.
- The ability to evaluate system performance in a unified design environment.
- The ability to incrementally segue real hardware and software into models and simulations.
- The ability to seamlessly move from high level models into real hardware and software.
- The ability to work at multiple levels of abstraction.
As we study and practice the design and development of any serious large-scale systems, we must consider both the *system* to be designed and the *environment* in which it must operate. The abstract view given in Figure 9.15 captures this. We will refine the model as our discussion evolves and we iteratively develop ideas.
As our design progresses, we decompose the top-level system block into modules and subsystems. Some of those modules will be hardware, some software, and some in a middle area that may be either.

Components that must or should be either hardware or software are generally clear. We can say: this part must be hardware or this part must be software. For example, the power supply, display, communications port are necessarily hardware. We can agree that the operating system and associated communications drivers are necessarily software.

Graphically we have expressed the situation in Figure 9.16. We have a gray area between the hardware and software where the implementation approach not precisely defined. Co-Design focuses on this area; those components that may be either hardware, software, or both. In such cases, we are making engineering decisions or trade-offs related to speed, cost, size, weight, or other factors.

Co-Design emphasizes models – working from the abstract to the concrete. It is inherently a top-down process that flows as illustrated in high-level blocks in Figure 9.17. Recall that the major elements of the Co-Design process include: specification, functional decomposition, partitioning, co-design, modeling, architecture, co-synthesis, co-simulation, and co-

- 20 of 105 -
verification. In our discussions, we will go through each process step and examine the critical points.

![Diagram of the Co-Design Process]

**9.8 LAYING THE FOUNDATION**

As we begin to explore the product development cycle, we will walk through each of these six steps. Rather than focus on how one particular model approaches the interpretation of these steps, we will try to identify the essential elements of each. The approach that we will present is top down and iterative.

The first two steps focus on capturing and formalizing the external behavior of the system. The remaining four move inside the system and repeat, as desired or necessary, the process for the development of the internal implementation that gives rise to the desired and specified behavior. As we will do from the outside, we will move from the general to the specific, capturing and specifying each aspect of the design on the inside.

A major task, once we start to move inside the system, will be that of decomposing and refining the design from a nebulous entity that someone needs into the product that meets that need. We will first decompose (organize) the collection of customer’s wishes into functional blocks that are then partitioned and mapped into an architecture. That architecture provides the aggregate of hardware and software modules that will make up the ultimate system. The final step in the design cycle is that of bringing the design together into a prototype, testing, and ultimately into production.

Because there is not one right answer, the problem represents a challenge and an opportunity to be creative. A colleague who worked on numerous designs of a particular piece of measurement technology once said, ‘although each design performs exactly the same function, each also represents an opportunity to explore a new approach that is better than the old.’ That colleague built a career around doing what everyone else said could not be
done...including some of the top names in the industry. One of the best ways to learn how to do something it is simply to do it; so, Let us get started. As we walk through each of the steps in the Co-Design process, we will see how they apply to the following design. We begin with a textual description.

**EXAMPLE 9.0**

*Designing a Counter*

As a senior development engineer at *Your Time is Our Frequency, Ltd.com*. You’ve just finished one project and are now getting ready to head off to the next. As part of the early planning of that project, you and one of the marketing folks are traveling around the country talking with people from a number of different engineering firms. You are trying to determine what features your customers would like to see in the next generation product.

You’ve been on the road with this guy for a couple of weeks now and are anxious to get home. All the cities are beginning to look exactly alike. Tuesday, this must be Cleveland…hmmm, looks just like the last three cities, oh well. This is the last customer for this trip. This morning, you’re talking with *High Flying Avionics, Inc.* They’re interested in a new counter that can be used on several of their avionics production lines.

Following several hours of discussion with one of the manufacturing managers, you identify most of their requirements. Your discussion with them follows.

Business is a little slow right now and money is tight, so we don’t have a large budget to purchase a lot of different new instruments. In fact, ideally, we’d like to be able to use the same instrument on several of our lines.

Today, we have our technicians running most of the tests manually, but, in future, we’d like to be able to automate as many of these tests as we can. As we upgrade our systems, we’d like to be able to operate several of these counters remotely from a single PC. Here are some of the other things that we’d like to be able to do.

As part of our ongoing efforts to improve production and flow through our lines, we monitor the rate at which units arrive into each of the major assembly areas. To do that, we need to be able to track how many of our navigation radios come down a production line each hour. Because we support small quantity builds of different kinds of radios, the rate at which the units come past the monitoring points is not constant. As each radio arrives at an entry point, it breaks an IR beam. On most of the lines, breaking the beam generates a 1 μsec. wide, negative going 5.0 V pulse. However, we do have several older lines that we must still support. On these, the pulse is positive going.

On several of the newer lines, we have to measure frequency up to 150.000 MHz. We also have several tests for which we must measure frequencies in the range of 50KHz ± 0.001 KHz and 100 Hz with 0.001 Hz resolution. On another line, we have several instruments with output signals that have a duration up to 1.0000 ± 0.0001ms and others that have a duration of up to 9.999 to 10.000 ms and up to 1.000 ± 0.001sec. These signals are not periodic. Finally, we have several periodic signals on those same units that we must be able to measure with the same accuracy and resolution.
9.9 IDENTIFYING THE REQUIREMENTS

The development of any kind of well-conceived and well-designed system must begin with a requirements definition. Such a need holds, independent of the life cycle model that one chooses to work with. Unlike the people in Figure 9.18, paraphrased from an unknown author, we cannot begin a design until we know what we are supposed to be designing.

The goal of the requirements identification process is to understand the needs of all interested parties, capture a formal description of the complete system from the user’s point of view, then to document these needs as written definitions and descriptions. The focus is on what problem the system has to solve and what needs to be done. Such documentation forms the subsequent basis for the formal design specification.

Very often, we use the natural language of the customer and of the application context. We do so because such a formal expression of the requirements forces the early discussion and resolution of many complex problems, involving a variety of people with expertise in many different areas, particularly those who are knowledgeable in the application domain. We express the role that the requirements definition plays between the customer and those who execute the design with the accompanying simple graphic in Figure 9.19.

At this early stage in the product’s life, the primary step is focusing on the world in which the system will operate, not immediately on the system itself. The concern is how the world is affecting that system. We follow with an examination of the combination. Thence, our goal is to capture and to express purely external views of the environment, the system, and their interaction. With respect to the system, one refers to such a view as its public interface. The objective is to identify what needs to be done (and how well it needs to be done) starting with the user’s needs and requirements.

The requirements definition provides the interface between the customer and the engineering process. It is the first step in transforming the customer’s wishes into the final product. One can see, then, that the requirements definition is a description of something that is wanted or needed. It identifies and captures a set of required capabilities or operations. Once again, as
we begin to identify all the requirements, we must consider both the system to be designed and the environment in which it is to operate.

The first abstract model of the environment given in Figure 9.20 captures this. We will refine the model as our discussion evolves and we iteratively develop ideas.

From the diagram, it is evident that the environment surrounds the system. The inputs to and outputs from the system can come from or go to anywhere in the environment. As one begins, one should make no assumptions about the extent of either.

The first step is to abstract and consolidate that view such that both appear as black boxes as given in Figure 9.21.

The initial focus must be on the world or environment (the application context) in which the system is to operate. Next, one follows with an increasingly detailed description of the role played by the system in that environment, at each step adds to, and refines the requirements.

From the perspective of the environment, one can see that the requirements definition must include a specification for the containing environment, a description / definition of the inputs and outputs to and from that environment, a description of necessary behavior of the system, and a description of how the system is to be used.

From the system’s point of view, one starts at a high level of abstraction with an outside view and develops the definition(s) that are appropriate for that level. As was done when specifying the environment, through progressive refinement, one moves to lower levels of abstraction and a more detailed understanding and definition.

At this stage in the development life cycle, as the definition of the requirements solidifies and is ultimately formalized into a specification, one should not be encumbered by plans for implementation, rather the focus should be on the high-level behavior of system.
complete, accurate, and internally consistent specification must be available before one can start formal design. Ideally, it should be executable and thereby able to work in conjunction with a modeling tool suite. Such an executable specification ultimately serves as basis for verification and validation of the system.

Although an executable specification is a laudable goal, achieving that goal can become difficult when one must include support for non-functional constraints, integrate legacy components into an abstract model, and potentially combine different domain specific languages and semantics.

9.10 FORMULATING THE REQUIREMENTS SPECIFICATION

The objective of specification process is to capture the description of both the complete system and the environment. Such a description should be structured, understandable, and verifiable. Our first focus must be on the world or environment in which system is to operate. We follow with an increasingly detailed description of the role played by the system in the application. At each step, we add to the specification.

From Figure 19.21, and as stated earlier, our design must include a model of the containing environment, a description / definition of the inputs and outputs, a description of necessary behavior, and a description of how system is to be used. We start at a high level of abstraction with an outside view of the system. We then develop the model(s) appropriate for that level. Through progressive refinement, we move to lower levels of abstraction, to a more detailed model.

Let us examine some of the things that one should think about when starting to identify and capture the requirements then trying to define them in a formal specification. The form, extent, and formality of such a specification depends upon the project on which one is working, the target audience, and the company for which one is working. Remember, too, that it is a product that is being delivered, not a pile of paper. As a rule of thumb, the specification should be the absolute minimum necessary to capture and clearly identify all of the necessary requirements.

In capturing requirements, one strives to be very specific about the details from the user’s point of view. Bear in mind that one is identifying and formalizing the requirements. One still cannot begin to design until the specification has been completed and the customer has agreed to it. Remember, too, one should not be discussing microprocessors, memory, peripheral chips, or software modules at this point in the development process.

As one begins a design, one usually has some general ideas, had casual discussions, thoughts, but nothing firm. One can use these as a guide in directing the steps, but one cannot design from them. It is important to be careful, however, not to rely too heavily on preconceived ideas. One should always be open to alternative approaches. Starting to code or draw logic diagrams at this point is inviting major problems as the project proceeds. In all likelihood, the project will fail.

For the environment component of the specification, we must identify and establish a detailed picture that includes all inputs, outputs, and characterization of the functional behavior for each of the relevant entities that make up the target environment. We must know
and understand how the environment is interacting with and affecting our system as well as the effect(s) on the environment of the system’s output(s). For the system, we require a description of all inputs and outputs as well as a complete description of the functional and operational behaviors and the technological constraints.

At this juncture, we can naturally ask: how can one get such information about (let alone model) the system and the environment without describing or knowing the implementation of the system? The internals are inherently unknown at this point. How does one capture the desired behaviors?

9.10.1 The Environment

A reasonable first step begins with defining and describing the environment; the world in which the system must operate. The environment is a temporal world; it is a heterogeneous collection of entities of one form or another. It comprises the collection of physical devices to which the system is interconnected as well as any physical world attributes that the system is intending to measure or control or that can have an effect on the system. The initial goals in understanding the environment are to identify all relevant entities then characterize their effects on the system and vice versa. When the requirements specification has been completed, one should have all the necessary information about such entities with sufficient detail to support moving forward to begin developing the solution.

9.10.1.1 Characterizing External Entities

Each of the entities that make up the environment can be characterized using method similar to CRC cards: Class – Responsibilities – Collaborators. We can view the entities as characterized by a name, an abstracted public interface comprising the inputs and outputs, and the functional behavior. We are thinking in an abstract sense at this stage. The specification of the external environment should contain the following for each entity,

We now have:

- **Name and description of the entity**
  The name should be suggestive of what the entity is or does. The description should present the nature of the entity. Is it data, an event, a state variable, a message etc.? As an example, an entity may be something that is to be controlled such as the rudder on an aircraft or the clear air turbulence that must be accounted for in such a control.

- **Responsibilities – Activities**
  What are the activities or actions the environment expected to perform. The hydraulic system moving the rudder is part of the environment. Its action or responsibility is to move the rudder in response to the signal coming from the system being designed.

- **Relationships**
  What are the relationships between the entity and its responsibilities or activities?

- **Safety and Reliability**
  Safety and reliability issues must be included early in the specification process. With respect to the environment, at the requirements stage, the focus is primarily on safety. The goal is to identify all safety critical issues and hazards so that they can be
addressed in detail in the system design specification. One should also identify any regulatory agencies under whose auspices the system will operate.

9.10.2 The System

Next, focus shifts to the system’s point of view. The same questions posed for the environment are now asked about the system. As with the characterization of the environment, the initial goals are to identify all of the aspects of the public interface of the system then characterize their effects on the environment and vice versa. For the system component, we must have a description of all inputs and outputs, a description of the functional and operational behavior, and an identification of all technological constraints.

Let us use the requirements description and definition as a starting point. Such a definition describes the customer’s need; it is something that is desired. We talk with the customer. The analysis of the system leads to a synthesis of reality in the form of a model.

For the system, we formulate the design from three perspectives:

- Functional view – defines the system’s internal functions and the relationships between and among those functions.
- Operational view – captures and express the behavior of those functions.
- Technological view – formulates the hardware solution to the problem, identifies the components comprising solution, the implementation of the functional behavior on the hardware that is consistent with the identified constraints.

At this stage, for our system and based upon these perspectives, we develop three types of specification the functional specification, the operational specification and the technical specification.

Functional Specification

The functional specification describes functions or operations to be performed by system on / in the environment. The specification enumerates the system functions for the application. These are the external functions in contrast to internal implementations. They give a description of the behavior of the environment under the operation of the system for these functions. The specification poses and answers the question: ‘How does the system affect the environment?’

We view the system from the point of view of the user’s needs and requirements. We must observe or hypothesize what the system must do in its environment. This can be done in UML through Use Cases. We must observe how the system interacts with objects in its environment; such a view is purely external. Knowing the environment means modeling the objects without the system and understanding and describing the relationships between them. The functional specification gives us the bulk of our high order requirements.

Operational Specification
We are now capturing the detailed requirements and constraints. The operational specification focuses on the behavior, performance, information details, methods and approaches to be used in system. It leads ultimately to the *Design Specification*.

**Technological Specification**

The technological specification includes high-level timing and timing constraints, geographic distribution constraints, characteristics of the interface, and implementation constraints.
Our high-level model now takes on the form of the diagram in Figure 9.22.

![Diagram](image)

**Figure 9.22 Refining the System Specification**

### 9.10.2.1 Characterizing the System

We move next to characterizing the system. Once again, that process begins with identifying the inputs and outputs.

- **System Inputs and Outputs**

  The System interacts with the real world through entities described and defined in the environmental characterization. Inputs to our system are outputs from environmental entities and outputs from our system are inputs to environmental entities. We can thus see that the system I/O has already been characterized in the environmental entity specification.

  For each such I/O variable, we know its name, use as input or output, and its nature as an event, data, state variable, etc.

  We have a functional definition for each that identifies its structure, domain of validity, physical characteristics and technical constraints.

- **Functional View**

  As was done with the specification of the environment, focus now turns to the function that the system is intended to perform. Before it is designed, the system appears as a black box. It can only be viewed from external point of view. A section addressing functional behavior is now included in the specification.

  We have a *Functional Specification* that is (semi) static showing functional relations and data and control flow. Again, referring back to objects and UML Use Cases; before the system is designed, it appears as a black box. It can only be viewed from an external point of view.

  The functional description defines the external behavior of the system. It gives a characterization of the effects of the system outputs on the environmental entities and the system’s intended response to inputs from the environmental entities and the system’s response to inputs from such entities.
It is also our first cut at an internal implementation. It elaborates how the system is used and to be used by user. Such a specification is equivalent to developing a high-level model of the system.

The functional description can be captured in a variety of ways. One effective approach is to use the UML tools discussed earlier. One such model can begin with a UML State Chart and Activity diagram. Another view can be gained through data and control flow diagrams commonly used in structured design methodologies. We will look at others shortly.

As one formulates these diagrams and the specification, care must be taken to ensure that:

✓ The specified (and ultimately modeled) states are relevant and appropriate to the application.
✓ The actions associated with system I/O that are captured in the specification are necessary to express functional specification and accurately reflect the desired (external) behavior of the system as perceived and intended by the customer.
✓ The conditions or constraints on its behavior are only a function of the system inputs, specification states, internal events, and the appropriate time demarcation (relative or absolute).

**Operational View**

With respect to the operational specification, operational means the manner in which a function must operate, what conditions are imposed on the operation, and the range of operation. This specification is dynamic.

Such a specification must consider concrete numbers (precisions and tolerances) quantifying all variables in the functional specification, all operating conditions, and all ordinary and extraordinary operating modes. The known information may contribute to producing and evaluating a design and may include domain specific knowledge and proprietary or heuristically known to customer.

**Technological View**

The technological specification includes all specifications relevant to the hardware and software design. We can easily identify 6 areas that should be considered:

1. **Geographic constraints**
   
   For distributed applications, we must consider items such as topologies and communications methods. We must also identify restrictions on usage and environmental contamination.

2. **Electrical considerations for interface signals**
   
   Such considerations include characteristics and constraints on any electrical I/O signals. These are driven by the external environment and may be beyond control of designer.
3. **User interface requirements**

A system such as a medical or instrumentation device may have an interface to external world. We must consider presentation method(s) and protocols.

4. **Temporal considerations**

The system may have hard or soft real-time constraints imposed. The constraints may specify delays on signals originating from external entities, responses to system outputs by external entities, and internal system delays.

5. **Maintenance, Reliability, Safety, Security**

The system may have requirements for diagnostic tests, remote maintenance, or remote upgrade. We must have concrete numbers for MTTF and MTBF (Mean Time To or Before Failure) and environmental and safety issues. We must address performance under partial or full failure. We must identify security vulnerabilities.

6. **Electrical Considerations**

Electrical characterization of internal signals and behavior include power consumption, supplies, tolerance to degraded power, characterization of signal levels, times, frequencies, etc.

- **Safety and Reliability**

  In formulating the safety and reliability requirements for the system, the focus is on the high-level objectives of each and on the strategy for achieving those goals. Relevant information can be taken from the exceptions component of the UML Use Cases.

  The safety considerations should address:
  ✓ Safety guidelines, rules, or regulations under the governing agencies identified under the environment portion of the specification.

  With respect to reliability, one can specify:
  ✓ The system uptime goals
  ✓ Potential risks, failures, and failure modes
  ✓ Failure management strategy

**Identifying the Requirements**

Starting with the trip report from *High Flying Avionics, Inc.*, which discussed their needs for a new counter.

As a first step in the development process, one extracts and summarizes the essential information from the trip report. By doing so, one can begin to focus on what should be included in the requirements specification. From the discussions with the customer, a high-
level sketch of the system and the environment captures the essential parts of the problem. The next step is to begin to formalize the model of the system and the environment as illustrated in Figure 9.23. Let us put the Requirements Specification together.

In its initial configuration, the environment contains:
- A set of navigation radios which are to be tested
- The user who is doing the testing
- The factory

Signals flow from the navigation radio to the counter, but not the reverse. The factory has inputs to the counter as well; these include the power system and the ambient environment in the factory. The user’s interaction is bidirectional. The user must select and configure the measurement to be made and then view the results once the measurement is complete. For the computer, the signal interchange with the counter similarly occurs in both directions.

In the developing model, the factory can be viewed as an aggregation of test lines and the radios to be tested. Later, the remote computer is to be added. The system to be designed, that is, the counter, interacts with all three entities. Such interaction is reflected in Figure 9.24. Now let us move to the next level of detail.
The customer has stated that the counter is to operate in a factory environment on any of several production lines. Based upon such an understanding, one can make certain assumptions about temperature, power, and ambient lighting.

- Time intervals and frequencies on the navigation radios and events from equipment monitoring the production line are to be measured.
- The time intervals may be either periodic or aperiodic but cannot be both.
- The polarity of the event signal to be counted can be either positive or negative going.
- The data display and the annunciation for mode and range are the only outputs expected from the counter.
- The assumption is made that the signals to be measured are independent of one another.
- In future, commands will be sent from a computer to the counter to direct its operation. Data will be sent from the counter to the computer.

**THE COUNTER**

- The counter must have the ability to measure time intervals, frequencies, and to count events.
- The frequencies are fixed but span a range of values.
- The time intervals span a range of values and may be either periodic or aperiodic but cannot be both.
- The counter will support the ability for the user to manually select mode and measurement range for all input signals.
- The counter will continue to make and display the selected attribute of the signal until power to the system is turned off or until the user makes another selection.
- The counter will measure only one signal at a time.
- An event can be modeled as an aperiodic time signal.

The design will be sufficiently flexible to allow future inclusion of the ability to send commands from a computer to the counter to direct its operation. The response of the counter to remote commands will be the same as its response to front panel selections with the exception that measured data will be sent from the counter to the computer as well as to the front panel display.

- The response of the counter to remote commands will be the same as its response to front panel selections with the exception that measured data will be sent from the counter to the computer as well as to the front panel display.
Requirements Specification for a Digital Counter

System Description
This specification describes and defines the basic requirements for a digital counter. The counter is to be able to measure frequency, period, time interval and events. The system supports three measurement ranges for each signal and two for events. The counter is to be manually operated with the ability to support remote operation in future. The counter is to be low cost and flexible so that it may be utilized in a variety of applications.

Specification of External Environment
The counter is to operate in an industrial environment in a commercial grade temperature and lighting environment. The unit will support either line power and battery operation.

System Input and Output Specification

System Inputs
The system shall be able to measure the following signals

Frequency in three ranges
- High range up to 150.000 MHz
- Midrange up to 50.000 KHz
- Low range up to 100.000 Hz

Period in three ranges
- High resolution up to 1.0000 ms
- Mid resolution up to 10.000 ms
- Low resolution up to 1.000 sec

Time interval in three ranges
- High resolution up to 1.0000 ms
- Mid resolution up to 10.00 ms
- Low resolution up to 1.00 sec

Events – up to 99 events in 1 minute
All signal inputs will be
- Digital data
- Voltage range 0.0 to 4.5 VDC
**System Outputs**

The system shall measure and display the following signals using a 6 digit display:

**Frequency in three ranges:**
- High range up to 200.000 ± 0.001 MHz.
- Mid range up to 200.000 ± 0.001 KHz.
- Low range up to 200.000 ± 0.001 Hz

**Period in three ranges**
- High resolution up to 2.000 ± 0.0001 ms
- Mid resolution up to 20.00 ± 0.01 ms
- Low resolution up to 2.000 ± 0.001 sec

**Time interval in three ranges**
- High resolution up to 2.0000 ± 0.0001 ms
- Mid resolution up to 20.00 ± 0.01 ms
- Low resolution up to 2.000 ± 0.001 sec

**Events in two ranges**
- Fast up to 200 events in 1 minute
- Slow up to 2000 events in 1 hour

**User Interface**

The user shall be able to select the following using buttons and switches on the front panel of the instrument:

- **Mode:** Frequency, Period, Time Interval, Events
- **Range:** Frequency, Period, Time Interval – High, Mid, Low
  Events – Fast, Slow
- **Trigger Edge:** Frequency, Period, and Events
  Rising or Falling edge
- **Time Interval:**
  Rising to rising edge
  Falling to falling edge
  Rising to falling edge
  Falling to rising edge
- **Reset**
- **Power ON/OFF**
The measurement results shall be presented on a 6 digit display; leading zeros will be suppressed. The display shall be readable in direct sunlight and from any angle.

The front panel will appear as follows,

```
Freq  Period  Intvl  Events  Range
MHz   kHz     Hz     ms   sec   min   hr
```

**Use Cases**

The use cases for the counter are given in the following two diagrams. The first indicates manual operation through the front panel and the second through a remote connection to a computer.

The remote option will not be included in the initial model, but will be incorporated in a later release. The time of that release is to be determined.

Execution of the selected measurement function will not depend upon how (local or remote) that function was selected.

At power ON, the default mode is **Measure Frequency.** All ranges will default to their highest value.

**Measure Frequency** The counter will continuously measure and display the frequency of the input signal on the currently selected range as long as the **Frequency** mode is selected.

If the frequency of the input signal exceeds the maximum allowable value on the selected range, the display will present the full scale reading and will flash.
If the frequency of the input signal is below the minimum allowable value on the selected range, the display will present a zero reading. If the input signal returns to a value within the bounds of the range, the value of the frequency will be displayed. The range may be changed at anytime by depressing the range select pushbutton. The use may elect to measure frequency starting on the positive or negative edge of the signal by depressing the start trigger edge pushbutton.

Measure Period  The counter will continuously measure and display the period of the input signal on the currently selected range as long as the Period mode is selected. If the period of the input signal exceeds the maximum allowable value on the selected range, the display will present the full scale reading and will flash. If the period of the input signal is below the minimum allowable value on the selected range, the display will present a zero reading. If the input signal returns to a value within the bounds of the range, the value of the period will be displayed. The range may be changed at anytime by depressing the range select pushbutton. The use may elect to measure period starting on the positive or negative edge of the signal by depressing the start trigger edge pushbutton.

Measure Interval  The counter will continuously measure and display the duration of the selected portion of the input signal on the currently selected range as long as the Interval mode is selected. If the duration of the selected portion of the input signal exceeds the maximum allowable value on the selected range, the display will present the full scale reading and will flash. If the duration of the selected portion of the input signal is below the minimum allowable value on the selected range, the display will display zero. If the input signal returns to a value within the bounds of the range, the value of the duration of the selected portion of the input signal will be displayed. The range may be changed at anytime by depressing the range select pushbutton. The user may elect to commence measuring the interval on the positive or negative edge of the signal by depressing the start trigger edge pushbutton. The user may elect to terminate the measurement interval on the positive or negative edge of the signal by depressing the stop trigger edge pushbutton. Note that the signal duration from positive edge to positive edge or negative edge to negative edge is the same as the period of the signal.
**Events** The counter will continuously count and display the number of occurrences of the input signal on the currently selected range. The accumulated count will be reset to 0 at the end of the select count duration.

The range may be changed at anytime by depressing the *range select* pushbutton.

The user may elect to increment the count on the positive or negative edge of the input signal by depressing the *start trigger edge* pushbutton.

If the number of accrued counts exceeds the maximum allowable value on the selected range, the display will present the full scale reading and will flash.

**System Functional Specification**

The system is intended to make 4 different kinds of digital measurement in the time and frequency domains comprising frequency, period, time interval and events. The activities associated with the *Measure Frequency* mode are shown in the following diagram.
9.11 THE SYSTEM DESIGN SPECIFICATION

The System Design Specification formalizes the qualitative view of the system given by the System Requirements Specification to present a more quantitative view. Thus, the purpose of the Design Specification step is to capture, express, and formalize the purely external view of the system identified during requirements definition. We have identified WHAT needs to be done starting from needs and the user’s requirements; we now quantify those WHATs. The step requires a solid understanding of the system behavior, the environment, and the system in the environment.

The time and frequency measurements will be implemented to provide three user selectable resolution ranges, high frequency range / shorter duration signals, a second for midrange frequency / midrange duration signals, and a third for low frequency / longer duration signals. The events measurement capability will support two selectable counting durations, shorter and longer.

For frequency, period, and events measurements, the user will be able to select either a positive or negative edge trigger. For interval measurements, the user will be able to select the polarity of the start and stop signals independently.

Operating Specifications

The system shall operate in a standard commercial / industrial environment

- Temperature Range 0-85°C
- Humidity up to 90% RH non-condensing
- Power 120 – 240 VAC 50 Hz, 60 Hz, 400 Hz, 15 VDC

The system shall operate for a minimum of 8 hours on a fully charged battery

The system time base shall meet the following specifications

- Temperature stability 0-50°C
  - < 6 x 10^{-6}
- Aging Rate
  - 90 day < 3 x 10^{-8}
  - 6 month < 6 x 10^{-7}
  - 1 year < 25 x 10^{-6}

Reliability and Safety Specification

The counter shall comply with the appropriate standards

- Safety: UL-3111-1, IEC-1010, CSA 1010.1
- EMC: CISPR-11, IEC 801-2, -3, -4, EN50082-1
- MTBF: Minimum of 10,000 hours
The formal design specification must be written in precise language stating specific requirements of the system. It can include,

- Tables
- Equations or algorithms
- State or flow diagrams
- Formal design language
- A pseudo language.

Unless there are exceptional and limited circumstances, it does not include:

- Schematics
- Code
- Parts lists

Non-functional specifications have to be added; we use these to explain constraints such as

- Performance and timing constraints
- Dependability constraints
- Cost, implementation and manufacturing constraints

The *Requirements Specification* provides a view from the outside of the system looking in. The *Design Specification* formalizes and extends that view by with a view from the inside looking out as well. It is written in the designer’s language and from the designer’s perspective. It serves as bridge between the customer and the designer as we see in Figure 9.25.

Notice also that the *Design Specification* has two masters,

- It must specify the system’s public interface from inside the system.
- It must specify how the requirements defined for and by the public interface are to be met by the internal functions of the system.
We have seen that the Requirements Specification is written in less formal terms with the intent of capturing the customer’s view of the product. The Design Specification must formalize those requirements in precise, unambiguous language. It should be sufficiently clear, robust, and complete that a group of engineers could develop the product without ever talking to the author of the specification.

9.11.1 The System

As a part of formalizing and quantifying the system’s requirements, one must attach concrete numbers, tolerances, ranges, and constraints to all of system’s input and output signals. All timing relationships must be clearly defined. The system’s functional and operational behaviors are described in detail.

Design Note,

A good litmus test of the viability of a design specification is the question, ‘If I send this to my colleague (who is working for one of our subcontractors), will he or she understand this?’ If answer is no, the specification should be re-examined.

9.11.2 Quantifying the System

The quantification of the system’s characteristics begins with the inputs and outputs, based upon the specified requirements. The necessary detailed technical descriptions, specific details, and constraints are added to enable the engineer to accurately and faithfully execute the design.

- **System Inputs and Outputs**
  For each I/O variable, the following are specified:
  - The Name of the signal.
  - The use of the signal as an input or output.
  - The nature of the signal as an event, data, state variable, etc.
  - The complete specification of the signal including nominal value, range, level tolerances, timing, timing tolerances.
  - The interrelationships with other signals including any constraints on those relationships.

- **Responsibilities – Activities**
  For system responsibilities, the following are specified:
  - Functional and Operational Specifications
    The functional and operational specifications that will quantify the dynamic behavior of the system are now formulated. The functional requirements specification identifies the major functions that the system must perform from a high-level view. The operational specification seeks to capture specific details of how those functions behave within the context of the operating environment.
The manner in which a particular function must operate, the conditions imposed on the operation, and the range of that operation are now captured. The specification must consider concrete numbers, precisions, and tolerances. All variables in the functional specification, all operating conditions, and all ordinary and extraordinary operating modes must be quantified. The specification may include domain specific knowledge that is proprietary or heuristically known to customer. Such knowledge can be very important to the design.

In stating the specific design requirements for the system, one can use tables, equations or algorithms, formal design language, or pseudo code, flow diagrams, or detailed UML diagrams such as state charts, sequence diagrams, and timelines. Schematics, code, or parts lists are not included, except in limited circumstances.

- Technological (and Other) Specifications

The technological portion includes all detailed and concrete specifications that are relevant to the design of the system hardware and software. Six areas that should be considered can easily be identified.

1. Geographical constraints

   Distributed applications can span a single room, expand to include a complete factory, or encompass several countries. Consequently, one must address both the technical items such as interconnection topologies, communications methods, restrictions on usage, and environmental contamination as well as non-technical matters such as costs associated with the physical medium and its installation.

2. Characterization of and constraints on interface signals

   The assumption is made that signals between the system and the external world are electrical, optical, or wireless or they can be converted to or from such a form. The necessary physical characterization of each is obviously going to depend upon the type of signal. That is, an electrical signal is specified differently from an optical signal.

   Since many of the interface signals may be driven by the external environment, potentially they are beyond the designer’s control. Therefore, it is important to gain as much information about them as possible.

3. User interface requirements

   If the system interfaces to such external world devices as medical or instrumentation equipment, how information is presented and whether there are any relevant and associated protocols must be considered. There may also be standards that govern how such information must be presented.
Consider the significant risk that would arise if each avionics vendor presented critical flight information and controls to the aircraft pilot in a different way. The near disaster at Three Mile Island arose, in part, because of the confusion caused by too much information.
4. Temporal constraints

The system may have to perform under hard or soft real-time constraints. Such constraints may specify delays on signals originating from external entities, responses to system outputs by external entities, and/or internal system delays.

5. Electrical Infrastructure considerations

There must be a specification for and electrical characteristics of any of the electrical infrastructure. Included in this portion of the specification are power consumption, necessary power supplies, tolerances and capacities of such supplies, tolerance to degraded power, and power management schemes.

- Safety and Reliability

In formulating the design requirements for the safety and reliability of the system, the focus shifts to the detailed objectives of each and on the strategy for achieving those goals.

Safety considerations should address:

- Understanding and specifying any environmental and safety issues.

The reliability specification should include:

- Requirements for diagnostic tests, remote maintenance, remote upgrade and their details.
- Concrete numbers for MTTF and MTBF of any built-in self-test circuitry.
- Concrete numbers for MTTF and MTBF of the system itself.
- Consideration of the system performance under partial or full failure.

Let us now bring everything together as we continue with the development of the counter. The system Design Specification will follow, but extend, what has been captured in the Requirements Specification that will serve as input to the Design Specification. The focus will now be on providing specific numbers, ranges, and tolerances for signals that are within the system.

Once again, we will put together any thoughts about the environment and the system prior to writing the specification.

Environment

Specifications relating to the environment have been discussed earlier. There are no changes.

Counter

- When specifying measurement and stimulus equipment, the specifications for that equipment are generally 10 times (one order of magnitude) better than for the signals that must be measured or generated.
• The required margin is provided when specifying the range and tolerances on the counter's measurement capabilities.

• Specifications on counting events are based upon the granularity of the timing of the interval during which the events are counted.

• The values to be displayed at the measurement boundaries are now defined.

The next step is to provide any additional details that may be needed and to fully quantify the counter specifications.
EXAMPLE 9.0

Designing a Counter (Cont.)

Design Specification
Requirements Specification

Quantifying the specification

<table>
<thead>
<tr>
<th>Design Specification for a Digital Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System Description</strong></td>
</tr>
<tr>
<td>This specification describes and defines the detailed design requirements for a digital counter. The counter is to be able to measure frequency, period, time interval and events. The system supports three measurement ranges for each signal and two for events. The counter is to be manually operated with the ability to support remote operation in future. The counter is to be low cost and flexible so that it may be utilized in a variety of applications.</td>
</tr>
<tr>
<td><strong>Specification of External Environment</strong></td>
</tr>
<tr>
<td>The counter is to operate in an industrial environment in a commercial grade temperature and lighting environment. The unit will support either line power and battery operation. Specific details are included under Operating Specifications.</td>
</tr>
<tr>
<td><strong>System Input and Output Specification</strong></td>
</tr>
<tr>
<td><strong>System Inputs</strong></td>
</tr>
<tr>
<td>The system shall be able to measure the following signals</td>
</tr>
<tr>
<td><strong>Frequency in three ranges:</strong></td>
</tr>
<tr>
<td>• High range up to 150.000 MHz</td>
</tr>
<tr>
<td>• Mid range up to 50.000 KHz</td>
</tr>
<tr>
<td>• Low range up to 100.000 Hz</td>
</tr>
<tr>
<td><strong>Period in three ranges</strong></td>
</tr>
<tr>
<td>• High resolution up to 1.0000 ms</td>
</tr>
<tr>
<td>• Mid resolution up to 10.000 ms</td>
</tr>
<tr>
<td>• Low resolution up to 1.000 sec</td>
</tr>
<tr>
<td><strong>Time interval in three ranges</strong></td>
</tr>
<tr>
<td>• High resolution up to 1.0000 ms</td>
</tr>
<tr>
<td>• Mid resolution up to 10.00 ms</td>
</tr>
<tr>
<td>• Low resolution up to 1.000 sec</td>
</tr>
</tbody>
</table>
Events
- Events to 99 per minute
- Signal level 0-4.0 V ± 0.5 V
- Transition time 10 ns ≤ τ_{rise}, τ_{fall} ≤ 50 ns

Voltage Sensitivity
- 50 mV RMS to ± 5.0 V ac signal + dc signal

All signal inputs will be
- Digital data
- Voltage range 0.0 to 4.5 VDC

System Outputs
The system shall measure and display the following signals using a 6 digit display

Frequency in three ranges
- High range
  Measure: 0 – 200 ± 0.0001 MHz
  Display: 0 – 200.000 MHz.
- Mid range up to 200.000 KHz.
  Measure: 0 – 200 ± 0.0001 KHz
  Display: 0 – 200.000 KHz.
- Low range up to 200.000 Hz
  Measure: 0 – 200 ± 0.0001 Hz
  Display: 0 – 200.000 Hz.

Period in three ranges
- High resolution up to 2.0000 ms
  Measure: 0 – 2.00000 ± 0.00001 ms
  Display: 0 – 2.0000 ± 0.0001 ms
- Mid resolution up to 20.00 ms
  Measure: 0 – 20.0000 ± 0.00001 ms
  Display: 0 – 20.000 ± 0.001 ms
- Low resolution up to 2.000 sec
  Measure: 0 – 2.0000 ± 0.0001 sec
  Display: 0 – 2.000 ± 0.001 sec
**Time interval in three ranges**

- High resolution up to 2.0000 ms
  
  Measure: $0 - 2.00000 \pm 0.00001$ ms
  
  Display: $0 - 2.0000 \pm 0.0001$ ms

- Mid resolution up to 20.00 ms
  
  Measure: $0 - 20.0000 \pm 0.0001$ ms
  
  Display: $0 - 20.00 \pm 0.001$ ms

- Low resolution up to 2.00 sec
  
  Measure: $0 - 2.0000 \pm 0.0001$ sec
  
  Display: $0 - 2.00 \pm 0.001$ sec

**Events in two ranges**

- Fast up to 200 events in 1 minute
  
  Measure: $0 - 200 \pm 1$ event
  
  Display: $0 - 200 \pm 1$ event

- Slow up to 2000 events in 1 hour
  
  Measure: $0 - 2000 \pm 1$ event
  
  Display: $0 - 2000 \pm 1$ event

**User Interface**

The user shall be able to select the following using buttons and switches on the front panel of the instrument.

The mode select push buttons are interlocked to ensure that only one mode at a time can be selected.

Mode:

- Frequency, Period, Time Interval, Events

Range

- Frequency, Period, Time Interval – High, Mid, Low

  Events – Fast, Slow
Trigger Edge

Frequency and Period
  Rising or Falling edge

Time Interval
  Rising to rising edge
  Falling to falling edge
  Rising to falling edge
  Falling to rising edge

Reset

The reset button will clear the display to all 0’s and reset the internal timing/counting chain.

The counter will be placed in the frequency mode with the range set to KHz, and the trigger edge set to rising.

Power ON/OFF

The measurement results shall be presented on a 6 digit LED display; leading zeros will be suppressed.

The decimal point will move to reflect the proper value for the range selected as the range push button is pressed.

The display shall be readable in direct sunlight and from any angle.

The front panel will appear as follows,
Use Cases

The use cases for the counter are given in the following two diagrams.

The first indicated manual operation through the front panel and the second through a remote connection to a computer.

The remote option will not be included in the initial model, but will be incorporated in a later release. The time of that release is to be determined.

Execution of the selected measurement function will not depend upon how (local or remote) that function was selected.

At power ON, the default mode is Measure Frequency. All ranges will default to their highest value.

**Measure Frequency** The counter will continuously measure and display the frequency of the input signal on the currently selected range as long as the Frequency mode is selected. The following use cases are defined for the Frequency mode.
If the frequency of the input signal exceeds the maximum allowable value on the selected range, the display will flash and will present one of the following values based upon the selected range,

- 200.000 MHz
- 200.000 KHz.
- 200.000 Hz

If the frequency of the input signal is below the minimum allowable value on the selected range, the display will present a zero reading.

If the input signal returns to a value within the bounds of the range, the value of the frequency will be displayed.

The range may be changed at anytime by depressing the range select pushbutton.

The use may elect to measure frequency starting on the positive or negative edge of the signal by depressing the start trigger edge pushbutton.

**Measure Period** The counter will continuously measure and display the period of the input signal on the currently selected range as long as the Period mode is selected. The following use cases are defined for the Period mode.

If the period of the input signal exceeds the maximum allowable value on the selected range, the display will flash and will present one of the following values based upon the selected range,

- 2.0000 ms
- 20.000 ms
- 2.000 sec

If the period of the input signal is below the minimum allowable value on the selected range, the display will present a zero reading.

If the input signal returns to a value within the bounds of the range, the value of the period will be displayed.
The range may be changed at anytime by depressing the *range select* pushbutton.
The use may elect to measure period starting on the positive or negative edge of the signal by depressing the *start trigger edge* pushbutton.

**Measure Interval** The counter will continuously measure and display the duration of the selected portion of the input signal on the currently selected range as long as the *Interval* mode is selected. The following use cases are defined for the *Interval* mode.

If the duration of the selected portion of the input signal exceeds the maximum allowable value on the selected range, the display will flash and will present one of the following values based upon the selected range,

- 2.0000 ms
- 20.000 ms
- 2.000 sec

If the duration of the selected portion of the input signal is below the minimum allowable value on the selected range, the display will display zero.

If the input signal returns to a value within the bounds of the range, the value of the duration of the selected portion of the input signal will be displayed.

The range may be changed at anytime by depressing the *range select* pushbutton.

The user may elect to commence measuring the interval on the positive or negative edge of the signal by depressing the *start trigger edge* pushbutton.

The user may elect to terminate the measurement interval on the positive or negative edge of the signal by depressing the *stop trigger edge* pushbutton.

Note that the signal duration from positive edge to positive edge or negative edge to negative edge is the same as the period of the signal.
**Events** The counter will continuously count and display the number of occurrences of the input signal on the currently selected range. The accumulated count will be reset to 0 at the end of the select count duration. The following use cases are defined for the Events mode.

If the number of accrued counts exceeds the maximum allowable value on the selected range, the display will flash and will present one of the following values based upon the selected range,

- 200 min
- 2000 hour

The range may be changed at anytime by depressing the range select pushbutton.

The user may elect to increment the count on the on the positive or negative edge of the input signal by depressing the start trigger edge pushbutton.

**System Functional Specification**

The system is intended to make 4 different kinds of digital measurements comprising frequency, period, time interval and events.

The time and frequency measurements will be implemented to provide three user selectable resolution ranges, high frequency range / shorter duration signals, a second for midrange frequency / midrange duration signals, and a third for low frequency / longer duration signals. The events measurement capability will support two selectable counting durations, shorter and longer.

For frequency, period, and events measurements, the user will be able to select either a positive or negative edge trigger. For interval measurements, the user will be able to select the polarity of the start and stop signals independently.

The system will be designed so as not to preclude the incorporation of a remote access option in future.
The system comprises 6 major blocks as given in the following block diagram.

**Input Subsystem** – the input subsystem shall provide the ability for the user to select any of the measurement functions, ranges, and triggering polarities. The subsystem also selects and routes the input signal to the appropriate portion of the measurement subsystem.

**Output Subsystem** – the output subsystem implements the range, edge selection, control information, and data formatting for proper presentation on the front panel display.

**Time Base** – the time base subsystem is a phase locked loop and divider chain driven from a 100 MHz crystal oscillator. This subsystem will provide two clock phases to drive the internal control and decision logic. Each phase will be $200.0000 \pm 0.0001$ MHz.

The time base will also provide the following frequencies that are used to define the measurement windows for the events and frequency measurements and provide the counting frequencies for the time interval and period measurements.

- Frequency – $200.0000 \pm 0.0001$ MHz
- Period – $100.0000 \pm 0.0001$ MHz
- Time Interval – $100.0000 \pm 0.0001$ MHz
- Events – $10.00 \pm 0.01$ Hz

**Measurement Subsystem** – the measurement subsystem provides the logic and control to execute the measurements of time and frequency.

- The frequency measurement will be implemented by opening a window for $1.00 \pm 0.01$ seconds. During the time the window is open, the measurement subsystem will gate the unknown input frequency into a 7 stage binary coded decimal (BCD) counter. When the window closes, the counter will contain the value of the unknown frequency.
The activities that are necessary to execute a frequency measurement as given in the following diagram.

- The period and time interval measurements will be made by opening a window on the specified signal edge. While the window is open, a frequency of 100.0000 ± 0.0001 MHz will be gated into a 7 stage BCD counter. When the window closes, the counter will contain the values of the unknown time interval.

- The counter will contain the number of events that occurred during the measurement interval.

- The events measurement will be made by opening a window for 1.00 ± 0.01 seconds for the fast mode and 3600.0 ± 0.1 seconds for the slow mode. During the time the window is open, the measurement subsystem will gate the unknown input to a 4 stage BCD counter. When the window closes, the counter will contain a measure of the number of events that occurred during the time interval.
Power Supply Subsystem – the power supply subsystem will provide the following voltages at the specified current levels to the internal logic.

+5.0 ± 0.01 VDC @ 10 A
+15.0 ± 0.01 VDC @ 500 mA
-15.0 ± 0.01 VDC @ 500 mA

At power on, there shall be a negative going reset signal. That signal shall remain in the low state for a minimum of 10 ms and shall have the ability to sink up to 1A.

Display – the instrument display shall display the results of the selected measurement on a 6-digit, 7-segment red LED display. The layout of the major features and functions is given in the earlier diagram.

Operating Specifications

The system shall operate in a standard commercial / industrial environment

Temperature Range 0-85C
Humidity up to 90% RH noncondensing
Power Automatic line voltage selection

• 100 – 120 VAC ± 10% 50, 60, 400 Hz ± 10%
• 220 – 240 VAC ± 10% 50, 60 Hz ± 10%

The system shall operate for a minimum of 8 hours on a fully charged battery
Net weight / size 2.75 kg, H: 90 mm x W: 200mm x D: 300 mm

The system time base shall meet the following specifications

Temperature stability 0-50 C
< 6 x 10⁻⁶

Aging Rate

90 day < 3 x 10⁻⁸
6 month < 6 x 10⁻⁷
1 year < 25 x 10⁻⁶

Reliability and Safety Specification

The counter shall comply with the appropriate standards

Safety: UL-3111-1, IEC-1010, CSA 1010.1
EMC: CISPR-11, IEC 801-2, -3, -4, EN50082-1

MTBF: Minimum of 10,000 hours
9.12 SYSTEM REQUIREMENTS VERSUS SYSTEM DESIGN SPECIFICATIONS

Examining the different steps that have been outlined up to this point, we seem to find a lot of duplication. It would appear that the System Requirements Specification and System Design Specification are just different names for the same thing. However, they are not; requirements and specifications are fundamentally different types of descriptions.

**Requirements** - Give a description of something wanted or needed. They are a set of needed properties.

Generally requirements come from the marketing, product planning, or the sales department and they represent the customer’s needs. The requirements definition and specification is not concerned with the internal organization or implementation of the system. Rather, it is an outside view intended to describe what a system must do and how well it has to do it, not how it does it.

The System Design Specification is generated by engineering as an answer to and a formal description of how to implement the requirements. Then the two groups negotiate and iterate until the requirements and specifications are consistent and achievable.

**Design Specification** - Is a description of some entity that has or implements those properties.

The system design specification is a means of translating the description of needs into a more formal and detailed structure and model. None-the-less, it seems that every part of the design needs another specification. Specifications can and do exist at various levels as the design is refined and elaborated. Different things must be quantified and at different levels of detail during different phases of the product development. The System Requirements Specification may specify a serial intersystem communication channel. In the System Design Specification, such a requirement may be written as: a serial intersystem communication channel with transfer data at the rate of 10,000 bytes per second at a specific bit error rate. The detailed Hardware and Software Specifications then establish the requirements and constraints on their respective components to be able to meet those specifications.

A Design Specification is a precise description of the system that meets stated requirements. Ideally such a document should be,

- Complete
- Precise
- Consistent
- Comprehensible
- Traceable to the requirements
- Unambiguous
- Modifiable
- Able to be written
The Design Specification should be expressed in as formal yet readable a language or notation as possible. Ideally, it should also be executable. It should focus precisely on the system itself and should provide a complete description of its externally visible characteristics, that is, its public interface. External visibility clearly separates those aspects that are functionally visible to the environment in which the system operates from those aspects of the system that reflect its internal structure.

9.13 EXECUTING THE HARDWARE/SOFTWARE CO-DESIGN PROCESS

We have looked at the first several steps in the Co-Design process: formulating a requirements specification and a design specification. At this point in the design cycle, all of the system requirements have been identified, captured, and formalized.

Based upon these documents, Let us now examine how we execute the design itself. Our focus moves inside the system and changes from what to how as we begin the process of specifying and designing the functionality that gives rise to the external behavior. The design process proceeds by:

- Identifying the major functions in the system.
- Partitioning the functions into hardware and software components.
- Refining interprocess communication.
- Modeling the hardware and software functions.
- Formulating the architecture.
- Co-synthesis
  - Synthesizing hardware – software interfaces.
  - Software synthesis – Target specific tasks to hardware components.
  - Hardware synthesis – Decompose computation steps into clock cycles.
- Co-simulation.
- Co-verification.

9.14 FUNCTIONAL DECOMPOSITION

As we move from formal specification to detailed design and implementation, we will start by identifying and decomposing the major functions that give rise to and support the system’s requested behavior. We want to ensure that we take a disciplined approach to the design of system hardware and software.

Goals:

1. Utilize strategies for developing the design solution from well-defined statement of problem.
2. Use a variety of tools to aide in rendering the system’s complexity understandable and tractable.
3. To attack the complexity of by partitioning into modules and organizing the modules into hierarchies.
4. Begin to establish criteria for evaluating the quality, reliability, and safety of the design.
Thus, the purpose at this stage of the Co-Design development cycle is to begin the task of finding an appropriate internal \textit{functional} architecture for the system. We are beginning to formulate \textit{how} the requirements that have been identified can be implemented. The current focus is on analyzing and thoroughly understanding the problem. Though such an analysis, a somewhat loose understanding of the design can be transformed into a precise description. The result should be a detailed textual or graphical description of the system that is a complete, consistent, functional definition of the required behavior.

To establish an appreciation for a functional model of a system, consider an aircraft. If an aircraft is the system to be designed, the top-level functional model should probably not consist of more than 3 major functions: \textit{take-off}, \textit{fly}, and \textit{land}. With such a view, we make no statements about such issues as the support structure for the aircraft (wheels, skies, pontoons), the propulsion system (jet, rocket, propeller), or the method of lift (wings - conventional aircraft or blade - helicopter). Early on, these are not important; such decisions can be postponed until later. The advantage of such an approach is early flexibility – time to explore before beginning to constrain the system. A functional description simply formalizes the intended behavior of the design.

The functional description should be written to be understood by those knowledgeable in the application domain and by those who will do the hardware and software development. The specification must also be such that it can be reviewed by the many diverse and interested parties and tested against reality. If it is too complex to read and understand, no one will read it. When the completed project is delivered, it is too late to discover that the customer’s view and developer’s view of reality are very different.

The first step in managing the complexity is partitioning or decomposing the initial high-level view into modules. We set the goals of such a process:

- The system should be partitioned so that the function of each module is easy to understand.
- Each module should solve one well defined piece of the problem.
- Partitioning should be done so that connections between modules are only introduced because of connections between pieces of problem.
- Partitioning should assure that connections between modules are as independent as possible.

A first functional decomposition is carried out based upon a search of essential internal variables and events in the system. The design process then consists of successive refinements or decompositions for each function (using exactly the same process) until elementary or leaf functions are obtained. Such decomposition forms a \textit{functional} or \textit{behavioral model} of the system. The model expressed, by the collection of such functions, should be sufficient to verify the design quality and to assess and evaluate system safety, behavior, and performance.

During modeling and verification, the system’s operations and associated performance requirements are allocated to the internal functions and the relations between such functions are defined. Such a process also allows one to estimate the expected performance of the system.
As with the *Requirements and Design Specifications*, ideally, the functional model should be executable so as to permit verification with respect to the specification. There are tools today that will allow us to do this. One such tool is a behavioral Verilog model. UML is also beginning to make executable models a reality.

The functional model is different from the specification and also from the physical architecture that will be developed next. The specification describes the *external* behavior of the system; the functional model is the first step in targeting the *internal* behavior that will lead that the external. Eventually, the architectural model will comprise the physical hardware and software components onto which the functions are mapped.

In Figure 9.26, we illustrate a first level functional decomposition of a simple input / output task. It is important to recognize that the functional decomposition is just that, a hierarchical decomposition; it is not a flow chart. The illustrated decomposition captures and expresses the requirement that the system must support receiving data from and transmitting data to the outside world. Associated with the task, is a required code conversion to ASCII.

![Figure 9.26 First Level I/O Task Decomposition](image)

Each of these functions may be further decomposed as necessary. If required, the second level functions may also be successively refined to give the level of detail needed to understand and to execute the design. The next step in the analysis is to identify the messages that flow between the user or other active external objects and the system as well as the internal signals that flow between the major functional blocks. We identify how the user will interact with the system to make it do what it is intended to do.

Let us now apply our understanding of partitioning to the functional design of counter system. First and foremost, we must continue to postpone the idea of working with the specific data structures, bits, bytes, microprocessors or array logics for a while longer. Though important later in the process, at the moment, they limit explorations and can bias the functional decomposition of the system.

**EXAMPLE 9.0**

*Designing a Counter (Cont.)*

**Identifying the Functions**
The first high-level diagram in Figure 9.27 presents an aggregation of the objects in the system. That aggregation includes both the environment and the counter being designed.
The model of the measurement system is expressed as a collection comprising

- The user,
- The factory,
- The future remote computer,
- The counter.

The factory is an aggregation of test lines and numbers of navigation radios that must be tested. Note that we are using the looser term *aggregation* rather than *composition* here.

**EXAMPLE 9.0**

*Designing a Counter (Cont.)*

**Functional Decomposition**

The *Design Specification* provided a high-level block diagram of the system. For this problem, such a diagram provides good starting place for the initial hierarchical decomposition of the system. Figure 9.28 elaborates on the counter components and gives one possible functional decomposition for the system.

The interface to the outside world is segregated into two functional blocks. The first is associated with the *presentation of information* to the user. The second is charged with *bringing in information* from the user and other tasks necessary to support the measurement. Both functional blocks are further decomposed into local operations versus remote operations.
Such a choice is made in the first case because the display is considered an output function and control an input function. In the second case, two different sets of functionality and different grammars for expressing the user’s commands are anticipated. Front panel operations tend to be rather straightforward; remote operations can be a bit more involved. Certainly, these are not the only choices.

The next drawing, in Figure 9.29, captures the interface between the counter and the surrounding environment.
Based upon the decomposition in Figures 9.28 and 9.29, Figure 9.30 now expresses a high-level functional partitioning and the signal flow between the major functional blocks.

Next, the system architecture is formulated and functions are then mapped onto the hardware and software blocks comprising the system.

### 9.15 PARTITIONING AND MAPPING TO AN ARCHITECTURE

The purpose at this stage in the development cycle is to find an appropriate internal functional architecture for the system. Throughout all of the previous discussions, modularity and encapsulation have been repeatedly stressed. We will look first at why such an approach is recommended and then at what should be considered as the process of decomposing and ultimately as partitioning the system into hardware and software modules proceeds.

#### 9.15.1 Initial Thoughts

So, to the first question, ‘Why do we do this?’ Reuse is one important reason. With each new design, one should always look to the previous project as well as the next one. What can be used from the last project to expedite the development of this one? How can the current design be implemented to support a future feature? Are there parts of this design that can be used in future projects?

Second, many compilers generate object code in segments, one for each module or translation unit. Such actions may place size restrictions on the individual modules. Poor module builds can significantly affect memory accesses, increase cache misses, promote thrashing and significantly reduce performance.
Third, work assignments are often made on a module by module basis. Module boundaries should be defined to minimize interfaces amongst different parts of the system. Such a practice simplifies the process of subcontracting some of the work as well. Security issues also must play a role when subcontracting is considered. Whether working for a toy company or on a sensitive government project, one needs to consider what information to make available to outside vendors. By properly decomposing a system, the portions that can be outsourced and those over which control should be retained can be more easily identified.

Fourth, the modules should be packaged with the goal of stabilizing the module interfaces during the early part of the design.

Fifth, partitioning the system into well-defined loosely coupled modules helps to ensure a safe, secure, and robust design. Such an approach helps to prevent a failure or attack in one part of the system from propagating into and affecting another. (See chapter xyz).

The importance of partitioning a new design should be evident; the next step is to examine the process for doing so. The process starts with the top-level system then progressively refines that model into smaller and more manageable pieces that can more easily be designed and built.

Initially, the focus is on a functional view of the system rather than specific pieces of hardware and software. It is important to understand and to capture the behavior at a high level first. The next step will then be to map those functions, that functionality, onto the hardware and software as necessary to satisfy the constraints identified during the initial phases of the design. Partitioning is important during the early stages of the development of the system to aid in attacking the complexities of a large system and then later as a guide in arriving at a sound physical architecture.

As we begin to think about organizing the system into the collection of pieces that will ultimately implement the customer’s requirements, one should continually look at the problem from both a high-level view and from a more detailed view. It is important to remember that developing a partition is not a one time process; it is not necessary to be perfect the first time. The partitioning process will probably need to be repeated several times before a satisfactory and workable decomposition is successfully achieved.

Prior to beginning the system partition, there are some general thoughts.

1. Remember that with every rule or guideline, there must always be room for exceptions.
2. Each module should solve one well-defined piece of the problem.
3. Mixing functionality across modules makes all aspects of the development and support process much more difficult. By doing so, one can easily create noodle hardware and spaghetti code. Future changes to such modules will be very difficult to implement and can easily lead to unexpected side effects and unrelated pieces of the system suddenly not working.
4. The system should be partitioned so that the intended functionality of each module is easy to describe and understand.
Although it is desirable to have well-defined modules, with simple interfaces, that solve nicely encapsulated pieces of the problem, sometimes in embedded applications one does not have such a luxury because of performance or economic constraints.

If the design can be described to and understood by other parties, then they will be able to maintain it and to extend it as necessary throughout the product’s lifetime. Remember, over half of the engineers who are involved in embedded systems design do not do new designs; they maintain and enhance existing designs.

During development, easy to understand designs will lead to fewer surprises as the design nears completion. All interested parties should be able to follow the design and comment as the process unwinds. A design that is too complex quickly discourages early criticism. Typically, people will not take the time to learn what the system is to do. Unfortunately, such early acceptance often is replaced by later rejection and potentially major redesign efforts. Although it is important to be proud of one’s work, one should practice what is called egoless design and always seek out others constructive ideas.

5. Partitioning should be done so that connections between modules are only introduced because of connections between pieces of problem.

   One should not put a piece of functionality into a module just because there is nowhere else for it to go.

6. Partitioning should assure that connections between modules are as independent as possible.

7. Once again, keep like things together. Such a practice helps to reduce errors.

   Partitioning is also done to help meet the economic goals of the design.

When forming partitions, the process must be considered from a number of viewpoints. Taking only a single point of view or neglecting any one can have significant long-term effects. At the end of the day, the system may meet neither the customer’s expectations nor the performance specifications.

As the decomposition process proceeds, the design should first be considered from a functional point of view. The outcome from the decomposition steps is a functional model that can be used to define the system architecture. Among the many things that should be considered, two that should appear early in the process are the coupling and the cohesiveness of the modules into which the system is being decomposed. The goal is to develop loosely coupled, highly cohesive modules. Let us see what these mean.

9.15.2 Coupling

Coupling is a heuristic that provides an estimate of how interdependent the modules are. Tightly coupled modules will generally utilize shared data or interchange control information. As module interdependence increases, so does the complexity of managing those modules, and the more challenges that one will have in…

   • Debugging the design during development
   • Troubleshooting the system in the event of field failures,
   • Maintaining the modules and system,
   • Modifying the design to add features or capabilities.
The major goal is to minimize coupling, i.e. to make the system’s modules as independent as possible. Reducing coupling means reducing the complexity of module interconnections. Low coupling between modules indicates a well-partitioned system.

**Design Heuristic:** The lower the coupling, the better the partitioning job that has been done.

During the early stages of the design, think about the following to help reduce coupling:

1. Eliminate all nonessential interaction between modules.
   If a particular piece of functionality or shared parameter is not a part the intended task of two modules, then eliminate it.
2. Minimize the amount of essential interaction between modules.
   While this sounds the same as the previous point, it is not. If an early analysis establishes that some interaction with another module is necessary, effort should be made to reduce the complexity of that required interface. The goal is to keep things simple.
3. Loosen the essential interaction between modules, if possible. Unless the environment demands a high degree of coordination between several modules to accomplish a task or to ensure error free communication, simply pass the module the information necessary to get the job done. Thereafter, wait for an indication that the task has completed. Execute some other part of the task.

Some of the ways to help to reduce complexity include:

- Reduce the number of interconnections between modules and thereby reduce the number of pieces of data that must flow between modules.
- Try to take the most direct route to a signal or piece of data as appropriate. There are some cases, security for example, for which the best implementation is to use a proxy as an interface to a signal or piece of data. In general, however, it is best to reduce the number of modules involved.
- In general, avoid using shared global variables. A better method is to pass data into a module via its parameter list or calling interface. With embedded applications, however, there are times when such sharing is critical to meeting time constraints.
- Avoid arcane interconnections between or amongst modules. A guiding principle underlying all design, once again, is to keep things simple.
- Don’t hard code values into a module’s parameter list or calling interface unless absolutely necessary. We must do so on occasion when an interface module or port must be at a specific address location; don’t make this a general practice.

9.15.3 Cohesion

An idea related to coupling is cohesion. The notion of coupling addresses the partitioning of a system; cohesion addresses bringing the pieces together. Cohesion is measure of strength of the functional relatedness of elements within a module. The goal is to create strong, highly cohesive modules whose elements are genuinely and tightly related to one another. Conversely, elements in one module should not be strongly related to those in another. We want to maximize cohesion and minimize coupling.
The use of cohesion as a reliability and quality metric has been around since the mid 1960s. A number of years of refinement and integration of the ideas of many people studying various designs and design approaches led Constantine and Yordon (Y and C 1979 Page-Jones) to formulate a cohesion scale based upon an ease of maintenance metric.

Let us look at several different kinds of cohesion.

**Functional Cohesion** – The module implements a single task and all comprising elements contribute to the execution of that one task.

**Sequential Cohesion** – The module implements a task as a sequential set of procedures. The output data of each procedure becomes the input data to the next. All of the comprising elements are involved in one of those procedures.

**Communicational Cohesion** – The module implements a task that has a number of procedures working on the same set of input data such as an image processing task.

**Procedural Cohesion** – The module implements a number of procedures that may or may not be related to a common activity. Control, rather than data, flows from one procedure to the next.

**Temporal Cohesion** – The module implements a number of unrelated procedures or activities that are sequentially ordered in time.

**Logical Cohesion** – The module implements a number of procedures that are possible alternative methods for accomplishing a task. A subset of those alternatives is selected by an outside user to actually execute the task.

**Co-incidental Cohesion** – The module aggregates a number of unrelated procedures. Such cohesion - or lack thereof should not be used.

We compare the different kinds of cohesion and coupling from several different perspectives in Table 9.0. The ranking is Excellent/Easy = 5…Poor/Difficult = 1.

<table>
<thead>
<tr>
<th>Cohesion</th>
<th>Coupling</th>
<th>Ease of Modification</th>
<th>Ease of Understanding</th>
<th>Ease of Maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Sequential</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3-4</td>
</tr>
<tr>
<td>Communicational</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Procedural</td>
<td>2-3</td>
<td>2-3</td>
<td>2-3</td>
<td>2</td>
</tr>
<tr>
<td>Temporal</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Logical</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Co-incidental</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Cohesion and coupling analyses provide a good set of metrics by which to begin to assess the high-level architectural aspects of a design. Remember, however, that both are guidelines. The work to ensure that the design is solid and that it is thoroughly tested remains.
There are plenty of good designs that require tightly coupled modules. CDMA cell phones are a good example of this. One can have tightly coupled multiprocessor designs as well as designs based upon message passing, the implication is not that one design is right or wrong, or better than the other, it is just how it was done to meet the requirements.

9.15.4 A Few More Considerations

With today’s systems, a spatial point of view is often essential. This is an external view of the system that leads to a distributed functional architecture. With such a view, performance and intrasystem communication costs are taken into consideration. Coupled with dependability, these are factors that must be considered. Closely associated with a spatial viewpoint is that of resource allocation; again, an external view. Such efforts result in a supporting “resource architecture”.

Finally, the emphasis moves to the hardware and the software. Decomposition into these two categories drives the design process that leads to a hardware architecture. Required performance in the target context now becomes an important consideration.

As embedded developers, we are playing a direct role in the design and selection of the hardware platform and the software environment. Intelligently making trade-offs in these two areas can take us a long way towards developing a safe, robust, and high quality / high performance system the customer has requested.

9.15.5 Approaches to Partitioning and Mapping

As we take this next step in the design process flow, our objective is to allocate operations comprising system behavior to the hardware and/or the software. There is a variety of different methods to attack the partitioning problem. We recognize that there are two extremes: a software oriented model and a hardware oriented model. The former initially puts everything into software while moving time critical pieces of functionality to hardware as necessary to meet time or speed constraints. The latter initially puts everything into hardware while moving non-time critical pieces of functionality to software as appropriate to meet time constraints.

That said, Let us begin by codifying the various approaches according to main properties or characteristics.

- The Model – Typically the system is expressed as a graph (from graph theory) derived from an abstraction of the system design specification.

- Cost Function – One metric for assessing a partition is to ask the question: How does such a partition negatively impact system performance? The question suggests devising a partition cost function that guides the partitioning algorithm towards minimizing the value of the function. In theory, the best partition is one that yields such a minimum value.

Such a function should measure the consequences of hardware / software allocation bounded by two extremes: all hardware and all software and the effects on system timing behavior, power consumption and memory loading.
Partitioning in software analyzes the statistical behavior of the executing program to drive the algorithm. Partitioning in hardware is looking at static allocation of functionality.

- **Granularity** – is usually based upon elements of the specification language: processes, tasks, loops, etc.
- **The Method** – How is the partitioning executed? What is the algorithm? Possible alternatives include aggregating or grouping elements, iteration, mathematical relations, or greedy heuristics that build the solution sequentially then picks the ‘best’ local decision without consideration for consequences. Here, best is subjective based upon the application and the designer(s).

We will now look at several different partitioning methods.

**Method 1** Due to Kumar

We can view the system being designed as comprising the following abstract sets:

- Set of hardware resources.
- Set of available software functions.
- The communications and control between the hardware and software units.
- Set of functions to be implemented. Such functions, determined by an earlier functional decomposition of the system, will be assigned to the hardware, software, or communications.

Formally, as we see expressed in equation Eq. 9.1 and modeled Figure 9.31, we have a mapping, $F$, from the set of comprising our starting functional blocks to the physical hardware, software, and communications blocks.

$$F: \text{functional blocks} \rightarrow \text{hw, sw, comms} \quad (9.1)$$

To these we add,

**Method 2** Due to Vahid

---

- 70 of 105 -
- Set of performance constraints
- A hardware size specification
- A software size specification

For functions to be designed and implemented for real-time systems, performance constraints are generally temporal. As we explore alternate partitions for the system, the goal is to find a partition of loosely coupled highly cohesive modules that satisfies the performance constraints and optimizes the hardware and software sizes. Such an assignment ultimately determines the speed of the operation.

Speed of operation can also be assessed in terms of the delay through the associated functions. However, note that the results of the partition and allocation process can potentially induce additional delays. Such delays are born from the intermodule communications overhead that can affect the available bandwidth of bus between processor and the peripheral hardware, accelerator hardware, and memory system. Alternately, as more software tasks are allocated to processor, we are increasing processor utilization. Such an increase can be viewed as positive or negative.

**Method 3 Due to Mahapatra and Vahid**

Numerous Co-Design tools exist for addressing partitioning problem; *functional partitioning* is one such approach. Functional partitioning refines the system’s functional specification into multiple sub-specifications. Each such sub-specification expresses the functionality of a hardware or software component within the system. The resulting components are synthesized into gates or code for the target system.

Viewed at an abstract level, the system to be partitioned is interpreted as a collection of procedures that we can model as set of procedures with a single top-level procedure:

\[ F = \{ f_0, f_1, \ldots, f_{n-1} \}. \]

Once again, these are the pieces of functionality that we determined earlier during functional decomposition. The procedures are structured into a call graph in which each node is a procedure and each arc is procedure call. A procedure call is modeled as a simple processor with read and write capability. Recall our earlier discussion of FSM based models.
The execution of the set $F$ is modeled as procedures executing sequentially with no concurrent operation. The objective is to map procedures to a software program and an HDL program as illustrated in Figure 9.32. This method is similar to Method 1 discussed earlier.

The functional partition operation creates a partition $P = \{p_0, p_1, \ldots, p_{m-1}\}$ containing $M$ parts or groups $p_k$; $M$ is of size 2 or more. Each part or group may be a hardware processor, a software processor, an ASIC or other programmable logic device such that each $f_i$ is mapped to one and only one $p_k$. As we see in Figure 9.32.

We have equations Eq. 9.2 and Eq. 9.3

$$F = \bigcup p_k \quad \text{and} \quad (9.2)$$

$$p_i \cap p_j = 0 \quad \text{for all } i, j, i \neq j \quad (9.3)$$

Each $p_k$ is implemented on single abstract processor. The abstract processors are mutually exclusive and may exist on same physical hardware device or software procedure. System operation, after partitioning, is the same as prior. Subsequent to the partition, no restriction is placed on the synthesis process. We may implement the process’ procedures such that they operate in parallel, provided that data and timing constraints are satisfied. The procedures are not mutually exclusive after partitioning, the processors remain so.

**Method 4 Due to Vahid**

Vahid proposes the three step partitioning method given in Figure 9.33. Prior to the application of the partitioning algorithm, the method serves as a guide to the process to reduce design time or the algorithm’s runtime.

The goal of the preliminary decomposition is to enter with the grain as large as possible, which supports maximum pre-estimation of partition and the application of heuristics for segregation. Components of system are included in a procedure only if segregation would yield inferior solution.
Method 5 Granularity

A key component in partition trade-offs is the granularity of the procedures entered into the N-way partitioning algorithm. Clearly fine-grained procedures give greater visibility into the system under design and flexibility in the allocation to blocks comprising the partition. However, such a process can be quite computationally intensive and potentially can limit the heuristics that require iterative estimations of the partition.

Course grained procedures can reduce run-time complexity for sophisticated partitioning algorithms. Grouping behaviors into atomic (indivisible) units eliminates partitions that may benefit from separating them.

There are some transformations that can help to guide the granularity selection problem.

- Inlining – Replace a procedure call by its contents. This can make the granularity finer and increases the code size. Inlined procedures may later be eliminated.

- Exlining – The converse of inlining. Sequences of statements are replaced by a procedure containing only those statements. We distinguish two types: Redundancy in which two or more nearly identical procedures are replaced by one and Computation in which large sequences of statements are refined into several smaller procedures. Statements within a procedure are highly correlated and ensure that a group of related statements is not decomposed during N-way assignment. The process moves towards finer granularity.
Cloning – Make a copy of a procedure for use by specific caller. We make a tradeoff between code space, communication needs, and efficiency.

**Method 6 Pre-Clustering**

Pre-Clustering is similar to but different from defining a granularity. The goal now is to reduce number of procedures that need to be allocated to groups then merge those procedures whose separation would never lead to good solution. That is, those that should never be separated in a good partition.

Consider the following: let procedure A contain 25 statements and let A be called from 10 separate locations in procedure B. There will be significant calling overhead between potential groups. Inlining A in B will add 225 new lines to the program. Cloning leads to similar problems. It is better to combine the two into the same cluster prior to partitioning.

Pre-clustering is different from granularity definition. Procedures under consideration to be clustered may not be able to be exlined. Calls to such procedures are not adjacent; typically, they are scattered throughout the specification. Pre-clustering is also different from the N-way assignment that follows. Each cluster does not represent a hardware, software, or other group. Thus, it is not a clustering operation and not guided by normal design metrics.

In the general algorithm for pre-clustering after granularity selection, each procedure is expressed as node in a graph. The graph is fully connected and each edge weighted with closeness of associated nodes. This is similar to routing in telecoms networks. The closest pair merged into a new node; merging is repeated until no pair has closeness measure below a specified minimum. Closeness is an empirical measure. Examples include communication bit density between nodes, shared hardware, or shared procedures, one in each set for example.

**Method 7 N-Way Partition**

With the N-Way Partition algorithm, the objective is to allocate procedures to a set of processors. A variety of heuristics may be used to create an initial solution. Statistical techniques include random distribution, genetic algorithm methods, or simulated Annealing – Hill Climbing. Be careful to avoid local minima with long runtimes. Greedy heuristics are a linear time heuristic that moves nodes to reduce the cost function. Port calling seeks to balance access to shared ports. The guiding heuristic tries for balanced allocation.

**Method 8 Graph Based Models Due to Ernst and Henkel**

The model associates nodes with elementary operations. Such operations derive from statements in the associated specification. The process begins with an all software implementation. The initial step is to identify bottlenecks that can be eliminated by migrating selected operations to hardware. The initial step is followed by a closeness estimation between operations. Principal operations include:
- Data Operations – which explores the number of common variables among operations.
- Loop Index Operations – which comprise initialization, increment, and compare.
- Control Operations – which examine the distance between invocations of the same operation or immediately successive operations.
- Operator Operations – which examine similarities between the operators add and subtract and shift left or right.

The next step is to estimate the communications overhead when elements are moved or exchanged among partitions. One such metric can be the density of data exchange amongst partitions. Finally, partitioning is implemented by two nested loops: the inner loop uses statistical methods such as simulated annealing and the outer loop is synthesis based to refine inner loop estimates.

9.15.6 Evaluation of a Partition

With each partition, we must revisit our boundary conditions. In doing so, we ask: is performance improved when piece of functionality is moved from software to hardware and is hardware complexity and size improved when piece of functionality is moved from hardware to software.
EXAMPLE 9.0
*Designing a Counter*  
*(Cont.)*

Mapping to Hardware and Software

From the functional decomposition, we now map the major functional blocks into the hardware and software modules as illustrated in Figure 9.34.

![Figure 9.34 Mapping to Hardware and Software](image)

**9.16 ARCHITECTURAL DESIGN**

Returning to the design of the system, in executing an architectural design, the goal is to select the most appropriate solution to original problem based upon the exploration of variety of architectures and the choice of the best-suited hardware/software partitioning and allocation of functionality.

**9.16.1 Mapping Functions to Hardware**

The view of a partition now changes to reflect a more detailed understanding of the system and involves the *mapping* or allocation of each functional module onto specific or the appropriate physical hardware or software block(s). Such a mapping completely describes the hardware implementation of the system and the allocation of the software.

As noted earlier, one should always endeavor to broaden the scope of the architectural design so as not to preclude possible future enhancements. Certainly, this involves a balancing act between generality and practicality as well as simultaneously satisfying other specified requirements. Nonetheless, the plan should be for a system that evolves over its lifetime; if this is done well, add-ons or modifications, that are an inevitable component in the life and evolution of today’s systems, will be much easier.

Now, the major objective to the architectural design activity, as we have alluded, is the allocation or mapping of the different pieces of system functionality to the appropriate hardware and software blocks. Work is based upon the detailed functional structure. The performance requirements are analyzed and finally the constraints that are imposed by the available technologies as well as those that arise from the hardware and software specifications are taken into consideration.
The important constraints that must be considered include such items as:

- The geographical distribution,
- Physical and user interfaces,
- System performance specifications,
- Timing constraints and dependability requirements,
- Power consumption,
- Legacy components and cost.

Such constraints are strong factors for deciding which portions of the system should be implemented in (which) hardware and which portions should be done in (which) software. The situation was illustrated earlier in Figure 9.16. The area between the hardware and software where the implementation approach is not constrained was identified as the area of Co-Design. In selecting the components that make up this area, one is making informed and well-reasoned engineering decisions or trade-offs of speed, cost, size, weight, as well as many other factors.

The mapping onto such an architecture initially defines a hardware and a software partition of the system. The constrained hardware portion of the system is specified by a physical architecture that may comprise one or more microprocessors, complex logical devices or array logics, or/and custom integrated circuits. It is important to keep in mind that with today’s systems, these microprocessors and microcontrollers can take on a variety of personalities: (CISC) complex instruction set, (RISC) reduced instruction set, or (DSP) digital signal processing core. The constrained software can be easily separated from the hardware. The remaining, unconstrained hardware and software portions fall into the Co-Design area.

9.16.2 Hardware and Software Specification and Design

The system Design Specification gives a detailed quantification of the system’s inputs, outputs, and functional behavior based upon our original requirements. The functional decomposition is analogous to those steps taken in defining the requirements. As the architecture of the design now begins to take shape, the objective is to determine, as fully as possible, the specifications for each of the physical components in the system and the interfaces between them.

The specification of the hardware for the entire system is decided by defining the hardware architecture and all its properties. The specification of the software is obtained by defining the software implementation or block diagram (using any of a variety of methods) for each software component of the architecture.

Each functional subset to be implemented in software is described by a detailed software specification that expresses the priority of each task and the spatial (data coupling), and temporal dependence relations between tasks. UML diagrams, including detailed state charts, timing diagrams, sequence diagrams, activity diagrams, and collaboration diagrams can all be very useful at this stage in the design.
A software implementation may or may not use a real-time kernel. With an off the shelf real-time kernel, the development time is reduced, but not the factory cost or time based performance specifications. For systems that do not use a real-time kernel (which represents 80% to 90% of small and medium systems) one can achieve a better optimization of the design when addressing high speed hard real-time constraints. Under such circumstances, the solution is being hand tailored to the specific problem rather than adapting a general purpose solution to a specific case.

For the software design, the following must be analyzed and decided:

- Whether to use a real-time kernel or not.
- Can several functions be combined in order to reduce the number of software tasks? If so, how?
- A priority for each task.
- An implementation technique for each inter-task relationship.

When it is appropriate, a real-time kernel or the services of an operating system can be used. In general, the main objective is to reduce the complexity of the organizational part in order to reduce the size and complexity of the software and the resulting development, testing, and debugging times.

Under such circumstances, a frequent choice is the Rate Monotonic Scheduling policy (this will be discussed in Chapter xyz; more frequently executed tasks are assigned a higher priorities). Permanent functions (those that run continuously without an activating event) and some cyclic functions without timing constraints are usually implemented within a background task.

For the implementation of inter-task relationships, it is desirable to use procedure calls as much as possible thereby simplifying the organizational part and reducing the inter-task overhead. Such an implementation is only possible between functions with increasing relative priorities. Tasks triggered by hardware events are invoked through the processor interrupt or polling subsystems.

For each specific sub-part of the system in which the partition is not obvious, a detailed specification is written; the final hardware/software partition is determined through a process of successive refinements as was done in the earlier decomposition process. Each hardware/software partition must also identify and include the hardware interfaces and the software drivers to support any inter-component communication.

The result is a complete mapping of all remaining functions and functional relations onto the hardware architecture. Among the important criteria that we strive to optimize are:

- Implementation (or factory) cost,
- Development time and cost,
- Performance and dependability constraints,
- Power consumption,
- Size
EXAMPLE 9.0

Designing a Counter

(Cont.)

Developing the Architecture

The next step in developing the counter begins with formulating the hardware architecture; the software architecture follows. We then map each of the functions identified earlier onto the architecture. The diagram in Figure 9.35 presents a first cut at the hardware components.

In the design, the microprocessor, the display, the front panel controls, and the power system are clearly hardware. In theory, the clock system as well as the counter-divider chain and associated control could be implemented, in part, in software. However, the frequency at which the counter is intended to operate (200 MHz) biases the decision towards a hardware solution.

The data and control flow diagram in Figure 9.36 identifies the major software tasks, shared data, and I/O. The front panel task is continually checking (directly by polling or indirectly by interrupt) the state of the front panel for user input. A change in input is captured and passed to the display task (which will update the display accordingly) and to the
measurement task. The measurement task issues the appropriate commands to the external counter-divider chain control block. At the end of each measurement, the raw data is read from the counter-divider and passed to the output task.

The output task properly formats the data and sends it to the display task for display on the front panel. The master control task manages the scheduling of all tasks and performs any necessary housekeeping or other duties as necessary.

9.17 FUNCTIONAL MODEL VERSUS ARCHITECTURAL MODEL

With the architectural model formulated, in retrospect, a good question that one might ask at this stage is, ‘Why is it necessary to design a functional model and an architectural model?’ well, Let us see. We start by looking at any system - hardware, software, a mix, it doesn’t matter. It quickly becomes evident that the internal organization of a system is based on a collection of components and interconnections between them. An appropriate model has to include elements both at functional level and at the architectural level to be able to represent, design, and evaluate hardware/software systems. Such a distinction is an essential building block of the Co-Design methodology.
9.17.1 The Functional Model

The functional model describes a system through a set of interacting functional elements. The design proceeds at a high level without an initial bias towards any specific implementation. We have the freedom to explore, to be creative, to make hardware – software tradeoffs. The behavior of a functional element is best described with a hierarchical and graphical model. The functional modules will interact using one of the following three types of relations:

- The shared variable relation - Which defines a data exchange without temporal dependencies.
- The synchronization relation - Which specifies temporal dependency.
- The message transfer by port - Which implies a producer/consumer kind of relationship.

We will discuss each of these when we study processes and inter process communication. All of them are critical in the design and development of today’s embedded systems.

9.17.2 The Architectural Model

The architectural model describes the physical architecture of the system based on real hardware components such as microprocessors, arrayed logics, special purpose processors, custom circuitry, analog and digital components, and the many interconnections between them.

9.17.3 The Need for Both Models

These two views, when considered separately, are not sufficient to completely describe the design of contemporary systems. It is necessary to add the mapping between the functional viewpoint and the architectural one. Such a mapping defines a (functional) partition and the allocation of functional components to the hardware and/or the software elements. Such a process is also called architectural configuration.

The functional model, located between the specification model and the architectural model, is suitable for representing the internal organization of a system. It explains all necessary functions and the couplings between them - expressed from the point of view of the original problem. Using such a scheme leads to a technology-independent solution. In particular, with this kind of model, all or part of the description can be implemented either in software or hardware. Once again, ideal for the Co-Design methodology.

The functional model is the basis for a coarse-grain partitioning of the system. Such a partitioning leads naturally to the selection of which functions to implement in hardware or software. The architectural structure is finer grained and generally follows from the functional model; the architecture may also be imposed a priori.

9.18 Modeling Tools and Languages for Co-Design

We have now formulated a high-level architecture for the system. The next steps are Co-Synthesis and Co-Simulation, i.e. mapping the architecture to an executable model, synthesizing the model, and simulating the system running the modeled hardware and software. Before taking that step, let us look at some of the available modeling tools and methods.
Specifying and modeling complex contemporary embedded designs draws upon wide variety of tools and methods. We want to be able to model and conjoin both the hardware and the software and thus, we must have a seamless way of expressing both and ultimately segueing in the real physical hardware. Solid models, as well as modeling and specification languages, are essential for comprehending, expressing, managing, and validating the design. It is important to keep in mind that no one tool does everything for all phases of the development process.

The Co-Design process utilizes a variety of models at different levels and for different reasons. For the structural and hierarchical aspects, we utilize system description languages like: SDL, SpecCharts, VHDL, Verilog, or Esterel. Digital signal processing requirements may be met with C/C++ or Matlab. Control aspects are addressed by variations on FSMs.

We have already encountered a system level model. Such an early model is generally necessarily behavioral. From the system level, we move to an architectural level model and subsequently to detailed level design models. As we proceed, we will look briefly at the motivation for modeling, what we are modeling, and the essential characteristics for a modeling method. We will then examine several different models and modeling/specification languages and try to point out the utility and limitations of each. This list is neither comprehensive nor complete.

To open, we are looking at what is called a model of computation (MOC). Such a model is an abstract specification of how a computation can progress and is typically hierarchical. In model driven engineering (which is a growing part of contemporary design), an MOC explains how the behavior of the whole system is the result of the behavior of the components. We can classify MOCs into the following five categories: logic circuit, finite state machine (FSM), RAM (comprising a CPU and RAM) a pushdown automata, and a Turing machine.

9.18.1 Why are We Modeling?

Primarily we use models to represent a description of a real system or one that will become real when it is designed. Models give us different views: external, internal, abstract, behavioral, or structural of our system. The model gives us means to describe characteristics of system to be designed and provides a basis for later simulation, test, verification and validation. Models are cheaper and easier to change than building a complete system to test a design concept or parameter variation. Models allow us to execute tests that may be too hazardous or impossible to run during preliminary development considering the underlying physics. Today’s familiar cell phone, sporting a design comprising over one million transistors, is virtually impossible to implement and test on the traditional lab bench. Interconnecting that many transistors is only a small part of the problem; signal integrity issues within the system are the real problem. Modeling offers the only viable solution.

In the design process, the model precedes the actual physical hard design. It provides the opportunity to rapidly explore a variety of alternative approaches cheaply and quickly.

9.18.2 What are We Modeling?
To effectively formulate a good model, we must understand what we are modeling. Our target is embedded applications. We are modeling the system, the environment, and the effects of the environment on the system and vice versa. We are modeling the synchronous or asynchronous control and resulting movement of data through a system. That movement can be sequential or parallel, finite or infinite. We know that embedded systems can be:

- Reactive – The system runs continuously and responds or reacts to signals from the external environment.
- Time based – Events and actions are synchronously scheduled.
- Often real-time – Time constraints are imposed on its behavior.
- Heterogeneous – Composed of hardware and software pieces.
- Supported by different development environments.

We need to distinguish the model from the language used to express the model.

9.18.3 Characterizing the Model

An essential part of the Co-Design process is having a good working model of the system to be designed. We are modeling the functionality of that system, not executing the detailed design of the system. This means that we are examining both the physical details, the implementation of the hardware and software, and the effects of the real-world. The model expresses an abstraction of the real-world. It is intended to give an abstract representation of a portion of the real-world; it allows us to temporarily ignore certain details as we gain understanding of the problem. To be useful, we can hypothesize some essential general capabilities of the model:

- Abstraction – It must allow us to express and examine the behavior of the complete system, unburdened by the details of sub-components.
- Refinement – It must allow us to express and decompose behavior of the system at different levels of granularity.
- Structure – It must be able to express the system as set of interconnected modules.
- Communication – It must support an inter module communication method.
- It should be easy to interpret.
- It must express the anticipated behavior or aspect being modeled in a comprehensible format.
- It should be easy to interpret.

9.18.4 Classes of MoCs

Two classifications or levels of model are particularly useful: conceptual and analytic i.e. control-dataflow and structural.

Conceptual Model

The conceptual model is behavioral in nature and precedes the analytic. It allows us to work at a higher level of abstraction. It uses a symbolic means to capture the qualitative aspects of the problem. It is particularly useful during the early stages of design for formulating the
specification, aiding in the early stages of partitioning the system, then later developing a high-level architecture. It should allow us to grasp and to work with the complexities of a design, to explore the functional behavior and to focus on essential details while ignoring others.

Analytic Model

The analytic model permits analysis at lower levels of detail. It uses mathematical or logical relations to express the quantitative physical behavior of the design. It is useful during the middle and later stages of design, the later stages of partitioning, in modeling and analyzing detailed architectures, verifying detailed performances, making performance trade-offs, and testing real-world effects. It is more structural in nature.

Let us now examine several different classes of MOC.

The Component Interaction class models systems that mix data driven and demand driven styles of computation. The Client-Server model can be viewed as demand driven; the push-pull interaction between producer and consumer exhibits data driven behavior.

The Communicating Sequential Processes class represented by Ptolemy, expresses such processes as Java threads.

The basic FSM enumerates the states and the rules for transitioning between states. Execution is an ordered sequence of state transitions that utilizes the notion of guards that constrain when a state transition should be taken. Such a scheme works well in control type applications. In the Dataflow class, modules react to data available on inputs, perform computations on the data, and emit data on the output.

Dataflow models react to data available on inputs, perform computations on the data then emit results on the output. In the dataflow class, we have the following models.

- With Process Networks, communication is achieved via streams of tokens. Each token is an arbitrary data structure operated on by the MoC. The approach models processes that communicate via buffered message channels. The models are loosely coupled and provide easy support for concurrent or distributed operation.
- Synchronous Data Flow models are a special case of the more general process networks. Restrictions or limitations are on the understanding of deadlocks and bounds on operations. Such modules execute sequentially to complete the task and support computations that operate on streams of data tokens. The schedule of operations is statically determined.
- Discrete Event models react to events occurring at an instant of time. An event comprises a value and a time stamp. Such a model is applicable to systems that respond to input events and produce output events either at the same time or at a future time instant.
- Continuous Time class models components that interact through continuous time signals and provide support for analog aspects of some circuits.

9.18.5 A Look at Some Models

To be effective, models should give us ability to express and support
1. Modularity and hierarchy – they should be able to express static and dynamic behavior and both structural and functional construction.

2. Relationships among subsystems – they should be able to express sequential and concurrent flow of control and inter subsystem synchronization and communication.

3. Communication amongst tools.

4. The use of legacy designs or behaviors and they should be executable and thereby should enable verifying the system throughout design process.

5. Models should be executable.

Let us look at some of the more commonly used models and languages identified earlier. Repeating, we can classify MOCs into the following five categories: logic circuit, finite state machine (FSM), RAM (comprising a CPU and RAM) a pushdown automata, or a Turing machine.

The example MOCs that follow are not a or the final architecture for an embedded system design; rather, we view them as building blocks: a modeled architecture that serves as a platform for a Co-Simulation of our system may comprise / utilize several of these. As we bring a project to its final design, we may iterate and model several variations on the architecture along the way as we make trade-offs and explore alternatives to reach an optimal design that satisfies all specifications.

9.18.5.1 The Logic Circuit

The logic circuit is a physical device that realizes a Boolean function. The logic circuit can be modeled as a directed acyclic graph as shown in Figure 9.37.

9.18.5.2 The Random Access Machine - RAM
The Random Access Machine – RAM MOC is modeled as two FSMs: RAM-CPU and RAM as shown in the diagram is shown in Figure 9.38.

![Diagram of RAM and RAM-CPU MOC](image)

The RAM component holds N words, each identified by an address. It has one output word: `outword` and three input words: `cmd`, `read`, and `write`.

- **cmd:** read, write, nop
- **read:** `outword ← data @ adx`
- **write:** `data@adx ← inword`

The RAM-CPU component comprises a CPU that implements a *fetch* and *execute* cycle as it alternately reads an instruction from RAM and executes it. The RAM MOC is similar to the Turing machine but supports random access.

### 9.18.5.3 The Turing Machine

The basic Turing Machine is a hypothetical computing device formulated by the British mathematician Alan Turing. The machine is capable of simulating any computer algorithm. The system comprises a *program* that drives a *read-write head*. The head interfaces to a *tape* and under the control of the program, it can read from or write to the tape and move to the left or the right i.e. make a state change. The basic Turing machine is shown in the diagram in Figure 9.39.

![Diagram of Turing Machine MOC](image)

### 9.18.5.4 The Pushdown Automaton Machine

The Automaton MoC is a less capable special case of the Turing Machine. It uses a stack rather than a tape.

### 9.18.5.5 The Basic Finite State Machine
The finite state machine (FSM) gives a simple, behavioral description of a hardware or software system. Its input/output function or relationship is computed by a finite automaton expressed as directed cyclic graph. The nodes in the graph define discrete states in the modeled entity. The arcs or edges, labeled with input/output data pairs, reflect relations and changes in state in the modeled entity. Typically one of the comprising states is defined as the initial state; there may also be a node expressing a terminal state.

The basic finite state machine is defined as a quintuple given in Eq. 9.4:

\[ M = \{ I, O, S, \delta, \lambda \} \]  
\[ (9.4) \]

- **I** - Finite nonempty set or vector of inputs
- **O** - Finite nonempty set or vector of outputs
- **S** - Finite nonempty set or vector of states
- **\( \delta \)** - Mapping \( I \times S \rightarrow S \)
- **\( \lambda_1 \)** - Mapping \( I \times S \rightarrow O \) - Mealy Machine
- **\( \lambda_2 \)** - Mapping \( S \rightarrow O \) - Moore Machine

Two models, *Mealy* and *Moore*, are distinguished by their output function. In the Mealy machine, the output is a function of the current state and the input whereas, in the Moore machine, the output is a function of state only. We can express the basic machine as shown in Figure 9.40.

![Figure 9.40 Basic Finite State Machine](image)

**Limitations**

There is a theoretical limit on computational power of the basic FSM model. It has limited useful memory; using states not efficient. One encounters a state space explosion for large problems and it is impractical for large numbers of inputs.

When it is necessary to express concurrent activity, one must use a combination of several machines, coupled by transition conditions between them. Under such conditions, once again we get a state space explosion.

The nature of the FSM makes it difficult to continuously refine. FSMs are inherently synchronous and the system is in single global state at each time instant. If we design a complex system by interconnecting one or more machines, outputs to inputs, we must have some form of synchronization either through a common clock or handshake scheme. There
are a number of variations of the basic FSM at the core of different models and modeling languages. Each addresses one or more of the limitations with this model.

9.18.5.6 Communicating Finite State Machines

To begin to address some of the limitations of the basic FSM, we will examine several examples of what are called *Communicating* finite state machines. Such machines support communications with other similar machines as shown in Figure 9.41.

9.18.5.6 Extended FSM

A first example of a communicating FSM is called an *Extended Finite State Machine* (EFSM). The EFSM model is expressed as a network of FSMs with a communication mechanism that is used to orchestrate the operation of the model. This is a key addition to the basic machine; the communication channel is non-destructive. An event can be read many times by the receiver. The simplest form of communication is the familiar shared variable; intrasystem communication is synchronous. The descriptive power of the EFSM is equivalent to a Turing machine. The extended model appears in Figure 9.42.

Modifications to the basic FSM include:

1. The addition of a (set of) variables or (FIFO) queues. The variables have a name and can hold abstract objects restricted to the integer type.

2. The queues are restricted to transferring integers.
3. The addition of a collection of logical or arithmetic operators that operate on contents of queues.

We see that the next state is function of present state, inputs, variables, and information from other machines. The output also becomes a function of variables that are computed as a function of present state, inputs, and variables.

Limitations

The model is at too low of a level to easily support specification. Generally, formal specification languages such as Esterel, SystemC, VHDL, Verilog, or State Charts are more effective.

9.18.5.7 Co-Design FSM

The second such model is called the Co-Design FSM. Its design is intended to address issues with the synchronous model underlying concurrent FSMs. Such a model implies that all comprising state machines must change state at same time. However, software implemented in a single processor environment requires interleaving actions in time; we cannot have parallel activity. The Co-Design FSM addresses issues of timing and coordination.

Similar to the EFSM, the CFSM model is expressed as a network of FSMs with a communication mechanism that is used to orchestrate the operation of the model. State changes in the comprising FSMs are asynchronous with respect to each other. Thus, the system is globally asynchronous. The time to compute a next state change can be different in each machine and can be unbound in the limit if the implementation is not known.

The communication mechanism between machines is based upon timed events rather than simple integers like the EFSM. An event is defined by the triple given in Eq. 9.5:

\[
\text{Event} = \{ N, V, T \} \tag{9.5}
\]

- **N** - The *name* of the event.
- **V** - The *value*
- **T** - A positive integer denoting the *time* of occurrence

Event transfer uses an unacknowledged protocol: the receiver does not acknowledge receipt of the event. Two kinds of events defined:

- **Trigger** events that provide the basic synchronism mechanism in a CFSM. They are used one time, which is called a *destructive* read, and cause a state transition in the target machine.

- **Pure Value** events that cannot cause a state change directly but can be used to select from alternate possibilities involving the same set of trigger events.

The operation of the CFSM comprises 4 phases:

1. Idle.
2. Detect an input event.
3. Execute a transition according to which events are present and match the associated transition relation.
4. Emit an output event.
Consequently, a state transition triggered by an input signal is reflected sometime later in an output change.

A timed trace of events is an ordered sequence in time. Time is monotonically non-decreasing. No two events with same name can occur simultaneously. This implies that no communication channel can carry two values at the same time, for example, two instances of the same interrupt.

The globally asynchronous nature of the CFSM facilitates partitioning the system model into hardware and software components. The low-level structure of individual CFSMs makes the synthesis of hardware or software rather straightforward.

Limitations
The communication mechanism is very specific. It does not easily support complex data types or data transformations. It can model other schemes such as blocking messages and shared variables; it can be cumbersome, however. Similar to the EFSM, it is too low level to easily support specification. Generally, formal specification languages more effective.

9.18.5.8 Program State Machines
The program state machine extends the FSM by integrating a hierarchical concurrent FSM and programming language concepts. The model is similar to the RAM MOC. In the machine, the concept of state represents a distinct mode of computation. Only a subset of program states can be active at any one time carrying out an associated computation or operation.

A program state can be compound, i.e. made up of a set of program states or elementary in which it comprises a single program state. Compound states may be executing in sequence or in parallel, similar to the UML State Chart and are thus useful for modeling multitasking operation. Elementary states are modeled as a sequential algorithm in an imperative language. Assignments are the only statements in such a language responsible for a change in state.

Transitions between sequential program states are of two types:
1. Transition Immediate arcs in which the transition occurs as soon as a condition becomes true, independent of the state of sub-program states: active – inactive or computation complete – not complete. Such arcs are intended to address reset or exception conditions such as interrupts.
2. Transition on completion arcs in which the transition occurs when the state has finished an activity. Such behavior is similar to signal-wait and signal-continue semantics in semaphores and monitors and are loosely analogous to guard conditions in UML state charts.

Limitations
Communication and synchronization between concurrent states is modeled through shared memory; no other means are provided.

9.18.5.9 UML State Charts

State charts can be viewed as a graphical specification language. They are a powerful tool for modeling and specifying the dynamic behavior of reactive objects. They extend the basic FSM by supporting hierarchical decomposition or refinement, concurrency, the notion of delays and timeouts, and support a history mechanism. Concurrency can be very useful for expressing simultaneous sequential behaviors in control systems. The history mechanism allows one to re-enter state at same point and contest it was in when previously exited.

9.18.5.10 Petri Nets

The Petri Net is useful for expressing and validating the behavior of a system. Like State Charts, they support concurrency and thus can be used to express both sequential and parallel behavior as well as behavior hierarchy.

The Petri Net is a bipartite graph comprised of nodes or places, transitions, and tokens. A node is represented by a circle and corresponds to conditions that may hold in the system. A place may or may not have a token. Transitions are represented by a rectangle and express events that may occur. Tokens signify the assertion of an associated condition and are expressed as a small solid circle or dot within a place.

Figure 9.43 illustrates several basic structures.

![Figure 9.43 Basic Petri Net Structures](image)

A Petri Net is marked by placing tokens on the various places. When all incoming arcs to a transition have a token, the transition fires which corresponds to the occurrence of the associated event. All tokens on input arcs are removed and a token is placed on all outgoing arcs. The result is change in state of the system as illustrated in Figure 9.44.
Limitations

Like the basic FSM, the Petri net is flat, thus, presenting difficulty in expressing structural hierarchy. Its complexity rapidly increases and communication between parallel threads can only occur through shared variables.

9.18.5.11 Kahn Process Network

The Kahn Process Network model is similar in structure to the Extended and Co-Design FSMs. The structure comprises a networked collection of processes. The processes, like finite automata, perform a mapping from input sequence to output sequence. The model supports distributed control and distributed memory.

The processes run autonomously and communicate via tokens through unbounded FIFOs. Like a CFSM, tokens can only be written or read once. Writes are nonblocking and Reads are blocking. Processes synchronize through a blocking Read operation. A process is either executing, following an Execute command, or communicating, following a Send of Get command, and the behavior is deterministic.

9.18.5.12 Control Flow – Data Flow – CDFG Graphs

Control and data flow graphs provide another means to express control and data flow in a system. We have explored them already under our discussion on structured design. The basic organization comprises control and data nodes. Control nodes interconnect to capture the expected control flow operations such as sequences, branches, concurrent operations, and loops. Each is linked to a data flow block.

The data flow block encapsulates a set of data computations. Each computation sequence is expressed as a directed acyclic graph. A data node expresses an arithmetic, logical, relational operation, or a read or write to or from memory or an external port.
9.19 Co-Synthesis

Material based upon

Mahapatra at Texas A&M

Lavagno et al

As we saw in an earlier diagram, partitioning, simulation, and synthesis work in concert. In addition, at the end of the day, they must implement the system we are designing. We will now look at the synthesis component.

Synthesis is a wide-open problem. We find that there are many different commercial, as well as research based tools available. Many of the commercial tools focus on the hardware. We will look at the general considerations without delving into details of specific tools.

Synthesis entails turning HDL code into hardware and software into firmware. Our system comes with a specification, a set of hardware and software resources, a mapping – via the partition onto an architecture. Our target architecture comprises collection of black boxes. These black boxes are charged with executing the hardware and software tasks within the evolving design.

The hardware comprises a microprocessor, microcontroller, microcomputer or combination thereof. It is certainly possible to have multiple copies of each; for now, we consider a single copy. We also have a set of hardware modules including memories, I/O devices, CPLDs, ASICs, and/or FPGAs.

The software, or more appropriately ultimately the firmware, includes a collection of firmware processes: the operating system, a kernel, device drivers, flow control, co-routines, function / procedure calls, concurrent operations, and error / exception management. In addition, we have the data, control, and address busses interconnecting these as well as supporting the interface between the hardware and software and the schedule, i.e. the control flow managing all operations.

A specific implementation of these black boxes is defined during the synthesis portion of the co-development. The basic implementation takes on following form shown in Figure 9.45.

![Figure 9.45 The Modeled System](image)

9.19.1 Constraints
Mapping to a physical implementation can potentially encounter a number of constraints. Such constraints and evaluation criteria are similar to those we encountered when proposing and evaluating partitions. Constraints of particular concern include:

- The ability to meet specification requirements. At the end of the day, one must satisfy requirements specification.
- The ability to meet time constraints including the time performance of constituent algorithms and real-time deadlines.
- The ability to meet communications costs both in time and in the complexity of the implemented solution.
- Size and weight constraints, which hold for both the software and the hardware. For the hardware, the design must fit within hardware components and for the software, the memory loading.
- Signal integrity issues and constraints exacerbated by substantially decreasing hardware footprints and increasing operating frequencies (in reality, the associated signal decreasing rise and fall times).

Today, the current state of the art imposes some restrictions on the hardware architecture that arise from our successes in the hardware arena. The current state of hardware synthesis tools is far ahead of those in software domain. Many of these perform very well. Synthesis from VHDL, Verilog, or SystemC is rather straightforward. Our desire is to make problem computationally more tractable as the power of the tools continues to increase.

Thus, to minimize the cost of a particular partition, we have a contemporary bias towards using off-the-shelf processors and easily synthesizable logic devices such as FPGAs. In such an environment we have libraries of proven components. Today, FPGA technology has advanced to point where a processor core can easily be dropped into middle of a CPLD device. The mapping from an HDL to array logic is well understood. Nonetheless, there remains much room for research and improvement.

ASICs and full custom ICs are expensive in both time and money. Once again, we have a bias towards ASICs that can be implemented from legacy libraries. Although more expensive than FPGAs, they are less costly than a full custom implementation.

Thus, while certainly improvements can always be made, there are also interesting areas are in software synthesis and in developing the associated tools. Once again, there have been many advances in this area. Code generators and template based software development are no longer novelties. Herein our focus can be on software component of the problem.

9.19.2 Software Synthesis

Translating a sequence of behavioral statements into a standard implementation language such as C or C++ is a relatively straightforward task. More difficult is formulating and meeting scheduling requirements and real-time constraints for the application.

Typically, the application is built on single processor – although this is rapidly changing today – accompanied by collections of programmable logics. The implication is that, in reality, with a single CPU we only have a single real thread of control. Consequently, concurrent threads of execution that may work well in a model but must be flattened into
linear or sequential execution order can present a challenge. Thus, in a simple case, four threads executing concurrently in a model only receive one fourth of CPU rather than all of it. We can increase CPU speed by four times, which can often be impractical, or extend the time constraints.

9.19.2.1 System Characterization

The systems we are considering are reactive and real-time. During modeling and partitioning, such systems naturally decompose into concurrent pieces of functionality. The mapping of the model to the software / firmware must respect such a decomposition. Significant elements of modeled concurrency that must be considered when executing the mapping include: communication, synchronization, computational flow, and timing constraints.

9.19.2.2 Scheduling

As is evident with migration to a physical implementation, the scheduling of the software execution on the hardware can be challenging problem. We have already examined scheduling algorithms in context of the study of operating systems. There, we saw that scheduling algorithms fall into several categories:

- Static – The schedule is determined at compile time.
- Quasi-static – Some schedule decisions are made at compile time while others are made at runtime.
- Dynamic – Schedule determined at runtime by a dedicated piece of software.

Such schedulers may rely on pre-determined information to establish execution order and some runtime information; these may be non-preemptive or pre-emptive. The same rules and objectives apply now. As with all other aspects of the Co-Design process, we are making trade-offs. These include: scheduling (of processes) flexibility, minimizing scheduling overhead, the costs of context switching, and reliability and predictability which now grow more important as we move towards final implementation.

For embedded applications with (hard) real-time deadlines, we prefer a static – non-preemptive implementation. We can relax to quasi-static with limited pre-emption and certainly, there are cases in which pre-emption is necessary. Such an approach moves towards reliability and predictability, certainly reduces (runtime) flexibility, and minimizes scheduling overhead.

9.19.2.3 Synthesis Methods

Most approaches to synthesizing the software component decompose the problem into sets of cooperating tasks then schedule according to classical algorithms or ad hoc techniques driven by knowledge of the problem and domain. Such an approach is limited to special purpose systems, which are ideal for such an approach.

Let us now look at several different approaches to this problem.

Synchronous Model

Approach due to Berry, Benveniste
The concepts involved originate with synchronous languages used for embedded development. As we have seen, such languages include: Esterel, in which the semantics are based upon a reactive model for synchronous and parallel systems and compile into FSMs that can then be executed. Lustre, a synchronous data flow language targeted towards reactive systems, specifically control and monitoring systems, and Signal in which the semantics is based upon model of multiple clocked flows of data and events.

The synchronous model utilizes a synchronous interpretation of time and underlies several contemporary synthesis tools. The synchronous approach is a familiar approach in engineering. Continuous time is divided into discrete units. The approach utilizes two common models: event driven and input or data driven. Schematically the models execute as follows in Figure 9.46:

```
for each clock tick
    get inputs
    compute system outputs
    update internal state
end for  

for each input event
    compute system outputs
    update internal state
end for

Data Driven  Event Driven
```

**Figure 9.46 Data Driven and Event Driven Models**

The underlying structure of the authors approach builds a mathematical framework in which the model is time synchronized to one or more clocks. The system advances ‘synchronously’ accordingly and exhibits concurrent deterministic behavior. The authors propose an ideal model of reactive systems in which system outputs are produced synchronously with inputs. Such outputs occur “instantly”. Intra system communication implements an instantaneous broadcast model in which signals are visible and reacted upon at the time of emission. The approach implements a global interleaving of input signals that determines how (asynchronous) inputs are managed and effects computed in time.

**Execution**

The design of the software system is event driven and begins with the Esterel model. It derives a single FSM from a collection of concurrent modules. Inter module communication becomes rather complex. The generated software emulates the syntactically derived hardware implementation from the Esterel program.

The synchronous approach avoids using a schedule. With each sensed event, the reaction by the system to all events present is computed and executed. With such an approach, we have a precise determination of the performance in (idealized) time. The real world component is accommodated, as appropriate, for context. Most other approaches map concurrent processes onto cooperating tasks and formulate a schedule for execution subject to timing constraints given in specification. One such approach uses the rate monotonic scheduler.

**Rate Monotonic Schedule Base**

Approach due to Cochran
Scheduling, based upon the rate monotonic analysis (RMA) algorithm, uses a static assignment of priorities with pre-emption. Deadlines are equal to the invocation period and it is assumed that system overhead is negligible. Recall that given such a hypotheses, if a given set of tasks can be scheduled by a static priority algorithm, it can be scheduled by the RMA algorithm.

The algorithm is extended to include: synchronization constraints, high priority I/O via interrupt service routines (ISRs), context switching overhead, multiple processors, and deadline management. The author accommodates multiple processors to evaluate the ability to schedule a selected allocation of tasks to the processors and provides feedback on potential bottlenecks and deadlocks.

Rate Monotonic Extension

Approach due to Chou, Walkup, Boriello

Earlier work (Esterel and StateCharts) used an idealized timing model as we have seen with the Synchronous model. It assumes that simple computations take zero time. Computations that violated the assumption are modeled as external signals to the system. Time constraints on such signals are prohibited. The authors extend such a model by adding timing constraints. The method accommodates both fine and coarse-grained timing constraints.

The specification is Verilog based and uses such constructs to provide structured concurrency that is augmented by watchdog-style preemption similar to the UML guard or the familiar watchdog timer. An associated action is invoked when a specified condition is met.

Timing constraints are specified using **modes**. A mode requires a scope or context in which a specified set of timing constraints must be met until one of the watchdogs triggers and either disables or forces an exit from the mode. When such a transition is initiated, each concurrent branch target to be disabled is allowed to run until a safe exit point is reached.

Similar to State Charts, modes express different states for the operation: initialization, normal operation, error recovery. Constraints on min and max inter event separation can be defined either within the mode or based upon a set of events in several modes.

Scheduling is performed within a mode and proceeds by identifying a cyclic order of operations that preserves I/O rates and timing constraints. Then, transforming each mode into an acyclic partial ordering is done by unrolling and splitting, if there are multiple parallel loops. Finally, the partial order is linearized, based upon a longest path algorithm and checked for feasibility then start times are assigned to all operations.

The Co-Synthesis phase leads to an operational system prototype. A prototype implementation includes:

- Detailed Design
- Debugging
- Validation
- Testing
Prototyping is naturally a bottom-up process since it consists of assembling individual parts and fleshing out more and more of the abstract functionalities. Each level of the implementation must be validated. That is, it must be checked for compliance with the specifications on the corresponding level in the top-down design.

Under the Co-Design methodology, the hardware and software implementations can be developed simultaneously and potentially involve specialists in both domains, hopefully reducing the total implementation time and yielding a top quality product. Often this doesn’t happen in reality. Typically, the software leads hardware; however, the Co-Design approach is gaining in popularity. In either case, a complete solution can be generated and/or synthesized both for the hardware (in the form of ASICs and standard cores etc.) and the software (in the form of software functions and the hardware / software interfaces). The resulting prototype can then be verified and trade-offs made as the design evolves.

### 9.20 Implementing the System

Activities in this step are highly dependent on the technology used. Remember, the prototype is a tool for understanding and confirming system design as well as making implementation trade-offs. It is a proof of concept. A word of caution, do not rush the analysis or design to get to the prototype. Also, do not be afraid to throw the prototype away. For small projects, it sometimes works to try to transform the prototype into the final product. For larger projects, it is usually more of a proof-of-concept that almost never can be migrated.

Remember, those who hurry through the design and coding because there’s a lot of testing to do are going to be spending long nights getting things to work and even longer nights with unhappy customers. For some reason, customers don’t seem to have much of a sense of humor when the failure of a product they have purchased has just cost them several million dollars. If you are selling to a general market, *your company* has just lost several million in R&D costs and you still do not have a product to take to market. So now, it is even worse, because you have missed an opportunity for sales revenue with a product that you cannot sell because it is poorly conceived, or it still is not ready.

### 9.20.1 Analyzing the System Design

We have been studying the system design process while moving from requirements to a design and ultimately an implementation. Now that the first level design is in place, it must be critically analyzed. This step provides several important checks on the design, first, and foremost, it verifies that the solution meets the original requirements and specifications. At this stage in the design flow, it may also be necessary to trade-off different architectural and functional or hardware and software aspects of the design. Such trade-offs must be made according to criteria identified in the original specification.

The first step entails a static analysis of the system. At this stage, the architecture of the system is examined. Of immediate interest is not how the system will behave at run time. The major objectives are to have a system that is easy to understand, build, test, and maintain. All too often new designers (and, unfortunately, some who should know better), proclaim “…but it works!!!” or “…we can always send out updates later!!!”. If a design needs routine and repeated updates, the design is fundamentally flawed. For systems that we're proud to put our name on, getting it to work is only one very small part of the job. Moreover, it is easy to get a
one-off version of any system to work. Making one or ten million of the same design in production that will ultimately work safely and reliably is a much larger challenge.

The main goal in any design is to work ourselves out of a job. We want the design to be so reliable and so well documented that any future modifications and extensions are effortless. The caveat, of course, is that one must also know what sufficient reliability is and when to stop documenting. Well documented means just enough so that people can easily understand the design, but not so much that it becomes the primary deliverable.
9.20.1.1 Static Analysis

Static analysis should consider 3 areas:

1. Coupling
   We have examined this aspect of a design already. Coupling is related to number and complexity of the relationships that exist amongst the various system modules. It also gives a measure of implications of a change. The goal is loose coupling.

2. Cohesiveness
   Another issue that is worth re-stressing, cohesiveness is a measure of functional homogeneity of elements that comprise the modules. This applies to both the components and the relations. One must consider both external and internal views. External cohesion begins with the appropriate naming and meaning of the constituent elements. Internally, the structure and relationships among components are analyzed. For example, coupling through shared data is more cohesive than messages. Messages imply a temporal dependency.

3. Complexity
   Two kinds of complexity are identified: *functional* and *behavioral*.
   Functional complexity is characterized by:
   - The number of internal functions and relational components.
     The goal is to keep these small. Generally, as the number of functions and relations decreases so does the complexity of the design. Note, this does not mean to sacrifice clarity.
   - Interconnections amongst elements comprising each module.
     The earlier discussion of coupling applies here as well. Keep things simple.
   Behavioral complexity is characterized by
   - The number of inputs and outputs
     Once again, the target is a smaller number.
   - The length and ease of reading and understanding the description of the module.
     If several paragraphs or a page of written text (in sub 6 point font) are required to describe the function of one of the modules, that module is probably too complex. To simplify such descriptions, use tables, logical equations, or pseudo code.
   - The flow of control through the module and the number and structure of state variables.
     Have a single major thread of control through the module and keep the number of states small.
9.20.1.2 Dynamic Analysis

The objective in performing a dynamic analysis on the system is to determine how it will behave in a context that closely approximates the ultimate working environment. Dynamic analysis considers the following:

- **Behavior Verification**

  The goal is to ensure that the behavior of system, in its operating environment, meets the operational specification. That is, does it perform the functions it was intended to perform as it was intended to perform it? This verification includes behavior at the boundaries of those functions. To be able to do so, of course, we need a good specification in the first place.

- **Performance Analysis**

  Performance Analysis ensures that the system, in its operating environment, meets the performance specification. The focus is on specific values for inputs and outputs. We will discuss this in a later chapter.

- **Trade-off Analysis**

  A Trade-off Analysis is necessary to determine optimal solution for the given constraints and objectives. Such an analysis, based upon only small set of performance criteria, may affect the ultimate success or failure of the product.

9.21 Co-Simulation

Material based upon

- Polis and Ptolemy projects at UCB
- Mahapatra at Texas A&M
- Lavagno et.al.

In the early days of embedded design, hardware always lagged the software. Design errors and flaws often remained hidden until later stages of the project development. Today partitioning and simulation go hand in hand. Using behavioral model simulation conjoined with other simulation tools, we can produce a much better simulation earlier in the project development cycle.

Simulation, in its many forms, remains one of our best tools for verifying and validating an evolving design against our incoming specification. Recall that Co-Design is iterative process. Part of the process involves partitioning system into hardware and software parts. To ensure a good partition, we must simulate against the initial specification, use the simulation results to modify partition, and repeat the process until a satisfactory solution reached.

We see this in the diagram in Figure 9.47 that reflects a portion of the Polis design flow.
Let us examine the co-simulation block first then the synthesis block. If we examine the flow above, we cannot afford to build a hardware platform for each iteration as we try out the design, make adjustments, and try again. It is simply too expensive and takes too much time. We want to be able iterate through process number of times, refining the design with each iteration. The desideratum during the early stages of design is that we want maximum flexibility. During the later stages of a design, we would like to be able to introduce physical hardware including the processors, FPGAs, and ASICs. Of these, the ASICs would be the most difficult since take most time to build. Certainly, we could have processors and FPGAs earlier.

To satisfy our objects during early stages, we use simulated hardware running real software. Later we migrate towards real hardware. Our goal is to be able to execute the software as fast as possible and we want to keep both simulations synchronized to ensure proper performance in the ultimate target platform.

To accomplish such a goal, one could simulate the target hardware design running on a general purpose HDL based simulator then execute the software solution on such a simulation or on a host that is faster than the target platform. For larger systems, such an approach becomes quickly impractical.

### 9.21.1 Tools

Co-Design tools enable today’s designers to work at a high level of abstraction as well as at the detailed level to enable them to gain a high level of confidence that when the system is ultimately fabricated it will perform according to initial specifications. Our typical simulation platform comprises the computer and the hardware and software each modeled as a process. Hybrid environments add co-processors to assume part of the computational load. The remainder remains in software.

We distinguish between simulation and emulation. With simulation, we model the software and hardware and we make best effort to duplicate the real system. With emulation, we
augment with hardware, as necessary, to ensure that we meet all timing requirements. Emulation is termed to be cycle accurate. This is usually done with FPGAs or some similar device. Emulation becomes essential when trying to identify higher speed, time critical operations. If we are designing a system to operate in the GHz region, we are not going to do that with a software simulation. Under such a circumstance, we are using a hybrid or extensible simulation. Emulator based or driven simulators have difficulty with behavioral based models.

9.21.2 Types of Simulations

We identify several major types of simulation: gate level, cycle based, event driven, and data flow. We may use several different simulators or simulation methods during the course of design. It is important to always choose the best tool for the job.

Gate Level Simulations

Gate level simulations try to replicate, as much as possible, the gate level implementation behavior of the CPU. The method is highly inefficient and impractical for larger systems.

Cycle Based Simulations

Cycle Based simulations simulate the bus interface and associated timing. The program is executed on a high-level instruction set interpreter. Signal state is evaluated at positive or negative going clock edges which provides information about the number of clock cycles necessary for given instruction sequence between I/O operations. Such an approach is used most often in large complex designs that have a significant number of tests/necessary computations. It is approximately 10 times faster than an event driven approach. Such simulators have difficulty with asynchronous designs and suffer a severe time penalty.

Event Driven Simulations

Event driven simulations are more accurate than cycle based. The value of every active signal is computed for every device during a clock cycle as a signal propagates. A signal is simulated for both value and temporal behavior. The event driven approach is used most often when a timing analysis must be performed or there is a need to test for race conditions and their consequences. The approach is clearly very computationally intensive and requires a significant computational platform if it is to be completed in reasonable time.

Data Flow Simulations

With data flow simulation, as the name suggests, the focus is on the flow of the data through the system. Generally, this approach is used for high-level simulation to verify overall functionality and correctness/validity of algorithms. Signals are expressed as a stream with no notion of time. Functional blocks are linked by signals and execute when data is present. See Petri nets. A simulator scheduler determines order of execution.
9.21.3 Approaches

There are various approaches that we can take to executing the co-simulation. In such an execution, we must address several issues:

- At what level are we simulating.
- How do we simulate the components comprising a mixed environment.
- How do we simulate all components at the same time.
- The software simulator typically runs faster that the hardware one, how do we synchronize the two.

Let us look at a couple ways to address these issues.

Detailed Processor Model

Processor components simulated by an event driven model include: memory, data path, control path, bus, instruction decoder, etc. Interaction between the processor and other components also uses the event driven model. Gate level simulation is done using a behavioral model; an event model is too slow. Our high-level model looks like that in Figure 9.48.

![Figure 9.48 High Level Processor Model](image)

Cycle Based Simulation – Bus Model

Cycle based simulation uses discrete event shells to simulate the bus interface. Such a model is not concerned with executing the software associated with the processor(s). The focus is low-level bus – memory types of interactions. Software is simulated using an instruction set architecture (ISA) model. The model provides timing information, in clock cycles, for a given sequence of instructions between I/O operations and trades speed for accuracy. The detailed model is now extended as illustrated in Figure 9.49.

![Figure 9.49 Cycle Based Simulation Bus Model](image)
Instruction Set Architecture – ISA Model

Using the ISA model, we simulate the instruction set in C. The C program then acts as interpreter for the embedded application. The software is executed on the ISA model which provides the timing details for the co-simulation. The ISA model can be more efficient than a detailed model of processor. It does not need the high overhead of a discrete event model. The detailed model now modified as given in Figure 9.50.

![Figure 9.50 ISA Model](image)

Compiled Model

Again, attacking the hardware speed problem; assuming a slower target processor. If we are trying for a more accurate software timing model, we translate the embedded software specification to native software code for the processor upon which the simulation being executed. Software execution on the host provides timing details on the interface (to the software) to the co-simulation. Accuracy depends upon interface information. The model is now modified as in Figure 9.51.

![Figure 9.51 Compiled Model](image)

Hardware Model

If target processor(s) are available or can be modeled using an FPGA, the physical hardware can be incorporated into the simulation. This is an ideal model during the later stages of development and we gain a substantial improvement in simulation speed. The disadvantage is that such hardware must be available. We now have the hardware model in Figure 9.52.
We have seen several approaches to Co-Simulation. These work well when focused on one or a few aspects of the process. We have commented several times earlier that we want the simulation to be able to follow design evolution. As the design matures, the tools should be able to support evolution.

We are suggesting that the Co-Simulation support follows several levels of abstraction. During the early stages, the hardware and hardware synthesized models are not available. We use a functional model to make hardware - software tradeoffs. During the middle stages, the functional model segues into a netlist model. Later, with the hardware confirmed, we can migrate back to a higher-level model for hardware.

**Master Slave Model**

One approach to attack problem is to decompose the simulation into pieces as we might do with a network. We architect a simulation comprised of a master simulator and several slave simulators. The master invokes the slave by procedure call as necessary. With such a scheme, we compromise concurrent simulation. Such a model appears as Figure 9.53.

**Distributed Co-Simulation**

An alternate approach utilizes a more distributed simulation. We can easily envision a widely distributed model with pieces of simulation residing at different vendors, universities, and companies all conjoined into a sophisticated simulation for very complex problems.
The architecture utilizes a co-simulation (software) bus and transfers data between simulators using procedure calls. A number of such schemes are available. The bus can take many forms: a ‘local’ bus, the Ethernet, or wireless. Such an architecture can support concurrency amongst simulators. Managing time is the critical element. We need to ensure that all elements of the simulation are synchronized.

Such a model appears as in Figure 9.54.

Heterogeneous Modeling – The Ptolemy Project

The current incarnation of this environment is Ptolemy II. It is designed as an environment for the simulation and prototyping of heterogeneous systems using the strengths of the object oriented methodology for modeling each subsystem efficiently and naturally. The system uses Java for its object capabilities and is designed to take advantage of several capabilities built into Java. Network integration can support distributed simulations, migrating code, built-in threading, and user interface capabilities. It uses XML for persistent data representation and designs are expressed in XML.

Its sophisticated type system supports type inference and data polymorphism. Components can operate on multiple different data types. Supporting behavior types, the components and domains have interface definitions. There is further support for static and dynamic structure.

The Ptolemy environment introduced the notion of domain polymorphism. Components can be designed to operate in multiple domains and interact with other components in a wide variety of domains.

Domains

The key to the heterogeneous approach is support for wide a variety of different design styles including real-time and distributed computing, distributed discrete events, and timed multitasking. Such styles are called Domains.

A domain implements a computational MoC for each particular type of subsystem comprising a design. The MoC is the semantics of interaction between modules or components and provides the means to support concurrency and time in different ways. Supported domains include: synchronous data flow which is data driven and statically scheduled, dynamic data flow which is data driven and dynamically scheduled, discrete event, and digital hardware model. It uses a functional simulator developed by Stanford and supports models ranging from detailed gate (structural) to behavioral.
9.22 Co-Verification

It is assumed that verification has followed each iteration through the partition-simulate-synthesize cycle. It is further assumed that we have been doing all the appropriate analysis of our design throughout. We are now in the latter stages of the development. We have completed a ‘final’ synthesis of our design and must confirm that it satisfies our original system specification.

9.22.1 Hardware-Software Co-Verification

Hardware-Software Co-Verification gives us a means for testing and stressing our design long before we have a physical prototype available. We endeavor to reduce risk by identifying problems earlier rather than later. As today’s designs become increasingly integrated, visibility into the internals of the devices becomes more and more limited. Unfortunately traditional tools such as oscilloscopes and hardware logic analyzers rely on being able to connect to and monitor important signals such as system busses or other data and control signals. With no places to connect to, they lose their effectiveness.

The JTAG port was developed with intention of giving some access. The attendant difficulty is that it must be designed into the system early in process and consumes valuable internal real estate and I/O pins. Another tool, Signal Tap developed initially by Altera provides a very effective way of implementing a soft logic analyzer onto an FPGA.

Driven by the high costs of errors in silicon coupled with increasing demands to reduce time to market, our desire is to verify that our design is correct prior to first silicon. Through Co-Simulation and Co-Verification we strive to verify the hardware – software interface, accelerate the firmware debug process, and exercise the boot code, hardware and software diagnostics, device drivers, board support packages, and some application code.

It is recognized that a good model improves the probability of a solid design. However, models are not perfect. Modeling every hardware and hardware – software interaction quickly becomes impractical as we move towards smaller and higher performance systems. A circuit that may perform superbly on paper or as a distributed prototype may suffer signal integrity issues such as severe noise, crosstalk, and EMI contamination as geometries are reduced and operating frequencies increase (rise and fall times decrease) in today’s designs.

9.22.2 Tools

To enable us to effectively Co-Simulate and Co-Verify our designs, we must have good tools at all levels of the process. Today there is a great variety of commercially available tools to help the embedded developer. Information on each of these can be found with the various vendors; Let us look at capabilities such tools have or should have.

Basic Capabilities

Most such tools share a common set of basic capabilities. These include control of hardware and software simulation at all levels of the development process. Essential are support for initialization and breakpoint synchronization set in either the hardware or software domains. Encountering a breakpoint in either should support halting and potentially later resuming the system. The ability to pass variables between hardware and software environments is also
necessary. Finally, support for various levels of abstraction that includes high-level behavioral HDL or C/C++ models, detailed component structural level and hardware emulators that can range from complete software abstraction with hardware stubbed out and interface modeled to system in silicon.

Levels of Abstraction

Let us now examine the necessary abstraction levels supported by many tools in a bit more detail.

- **Software only - Complete abstraction**
  The hardware is stubbed out and the interface is modeled or the application interfaces with stubbed out or modeled hardware through a device driver. Such an approach enables the functionality of the software to be tested and excludes detailed and constrained timing and asynchronous types of events.

- **HDL simulation of hardware**
  The approach can include a functional model of system bus. Such a model can execute atomic (indivisible) bus cycles, include reads and writes to supporting arrayed logics with correct bus timings and support modeling and handling asynchronous events such as interrupts. The method does not reflect the internals of the system CPU, missing registers, internal peripherals such as timers or signal converters but aids in debugging peripheral module functionality from the bus side when developing device drivers.

- **Instruction Set Simulation (ISS) – Modeling the CPU**
  Modeling of the system CPU or digital signal processor is usually done as a C/C++ behavioral level model. The simulation may run as a separate process under Unix / Linux. Such a process is separate from the HDL simulation sockets or similar inter-process communication scheme used to exchange information. The approach is not cycle accurate; however, an HDL wrapper surrounding the ISS model approach can be cycle accurate.

- **CPU Emulator**
  With the CPU/DSP model extricated from the system and replaced by ICE module(s) for actual processor(s), it is possible to accurately replicate CPU bus cycle timing thereby provide a robust platform for software development.

- **Complete HDL Model**
  Supporting a complete HDL model gives a full HDL implementation of a complete system that can be executing at the register transfer level (RTL) or discrete device level. The approach tends to be more useful for developing the hardware rather than the software side. It is useful for verifying detailed and critical timing, hardware and software interactions, the boot process and memory interfaces and subsystems.

- **Hardware Emulation**
Under such an emulation, the hardware is modeled through some form of arrayed logic. Specifically this is an emulation; however, execution speed approaches that of a full ASIC. Contemporary arrayed logics can support the inclusion of CPU cores that will enable and facilitate both hardware and software debug.

9.23 OTHER CONSIDERATIONS

Several other factors must also be taken into consideration. The product must be able to be manufactured and tested. Such factors must be addressed early in definition and design process. The three additional complementary and concurrent activities that need to be considered in today’s business world are capitalization, reuse, and requirements and traceability management. Let us look briefly at capitalization; design reuse is one of the central threads in this text.

9.23.1 Capitalization and Reuse

Capitalization
Capitalization is an essential element of the contemporary design process. Specifically, proper and efficient exploitation of intellectual properties (IPs) is very important today. Intellectual properties are designs, often patented, that can be sold or licensed to another party to develop and sell as (a part of) their product. The company MIPS, for example, designs computer architectures. They don’t actually do any implementation themselves, the design is their product; the ARM corporation does a similar thing.

Reuse
Any consideration of component reuse is an activity that should be addressed and done during the functional and architectural design phases of the development process. Such activities can be considered sometimes during prototyping as well. The end purpose is to help designers shorten development cycles,

Component reuse is facilitated in two ways: present and future

- Present
  By identifying external (existing) functional or architectural components that can satisfy some portions of the current system’s desired functionality,

- Future
  By identifying components of the solution under design that can or will be reusable on other or future projects or products.

To be reused, a component needs to be

- Well-defined
- Properly modularized
- Conform to some interchange standard

A well thought out, well designed module will be much easier to adapt to a new situation than one that someone pieced together for some ad hoc purpose and barely got working. The same is true for a portion of a well modularized system.
If, during the design phase, one makes decisions with an eye toward technology that could be reused, the chances of such reuse are greatly enhanced. Finally, if the goal is for a module to have wider applicability than a local venue, then the designs must accommodate existing national and international standards. With today’s international market growing daily, it is incumbent upon us design to such standards.

Once again, the real-world intrudes, and trade-offs are part of the process. While designing for reusability or striving for a modular design, there are other factors to be considered. Suppose there is not enough ROM space for the code if it is designed to be completely modular, then possibly modified to be very application specific. While it is known that the decision go modular might create problems in the future, we will end up with a non-competitive product if the budget is exceeded.

9.23.2 Requirements Traceability and Management

Requirements Traceability

Requirements traceability refers to the ability to follow the life of a requirement (from the original specification) in both the forward and reverse directions through the entire design process and the design. It should be clear that traceability is potentially a one-to-many relationship between a requirement and the components it relates or traces to (or that implement it).

An accurate and complete record of traceability between requirements and system components provides several important pieces of information through the product life cycle. Among these are included,

- The means for the project manager and (potentially) the customer to monitor the development progress.
- A path that can be used during the verification and validation of the product against the original specification. Knowing where and how a specified requirement has been implemented facilitates confirming that the requirement has been faithfully implemented.
- A means of identifying which hardware or software modules are affected if a requirement changes.

Requirements Management

Requirements management addresses issues during the development process

- Requirement modifications
- Changes
- Improvements
- Corrections

During the design, such changes are difficult to avoid for many reasons. Therefore, a clear procedure that facilitates a way to accommodate and track such modifications has to be used during the complete design process.
9.24 ARCHIVING THE PROJECT

When the product has finally been released to production, there is some work that remains to be done. During development, a tremendous amount of important design information has been produced and accumulated. Most of that information should be retained for a variety of reasons. If the product follows the typical life cycle,

- Bugs that must be fixed will be discovered as customers use the product,
- There will be future revisions,
- New features will be expected and added,
- The next generation product will build on the current.

to name just a few. The obvious question is: What must be saved?

The problem of dealing with what to archive is no different from those confronting the original design. That said, we can use the same approach and start at the top. The typical project will have had many contributors. A basic list can include,

- Product Planning
- Design and Development
- Test
- Manufacturing
- Marketing
- Sales

Each group will have information, knowledge, documentation, tools that will be important in future. Let us focus on the technical subset of these: design and development, test, and manufacturing. In earlier studies of safe and robust design, we identified a typical software project directory structure. That diagram is presented in Figure 9.55 here for reference.

![Figure 9.55 A Typical Software Directory](image)

Each of the groups participating in the development should have a similar directory documenting their portion of the project. The project directories and all their contents are one of the main items that must be archived. These are obvious.

Now, the less obvious. Today, software, firmware, and software tools are essential to the design and development of any embedded system. If the source code it is lost or the ability to rebuild from sources is lost, any future work on the project will be seriously impaired. Today, source code no longer means just the C, C++, Java, or assembler listing in electronic form or on magnetic media.
In previous years, hardware was generally supported by hand drawn hard copy documentation. If a drawing was lost, it could be regenerated by skilled designers by reverse engineering the existing part. Today a rich set of CAD (computer aided design), CAM (computer aided manufacturing), IC, FPGA, and HDL modeling and synthesis tools have supplanted the old methods.

All of those tools run on a computer. All of those tools are routinely modified or updated by their vendor. All of those tools also have a product life cycle and ultimately will no longer be supported. If the archived tools, designed and developed for yesterday’s computers will not execute on today’s running today’s operating system, they are of little use. Today, in addition to archiving the end product, archiving the complete development environment – computer, hard drive, operating system, etc. – is well worth considering. The documentation for the tools should be included in that list as well.

An essential step, once the collected archive has been set, is to conduct what is called a virgin build. A virgin build begins with a completely new environment or context. Next, the archived tools are installed and set up. The tools are run, as appropriate, and tested to see if the designated components of the product can be recreated. If the process fails, the missing components must be identified, added to the archive, and the process repeated.

Too often, over the span of the product development, we build simple special purpose tools to help manage the tasks of developing and building the system then forget that they are an essential part of the build or synthesis when creating the archive. The virgin build quickly reveals when those tools are missing.

Today, the financial investment in all aspects of a project development is significant. Retaining and protecting that investment for future use is an important closure to the cycle.

9.25 SUMMARY

In this chapter, we have reviewed the major phases of the traditional development process for embedded systems. We have learned that the complexity of contemporary systems now demands a more formal approach, more formal methods, and tools to support that process. To accommodate such demands, we introduced the hardware/software Co-Design development cycle and methodology. We learned that the main goal of the methodology to provide the means by which engineers can simultaneously develop the hardware and the software and identify and solve design problems early in process. The more detailed aspects of that process are covered in conjunction with the study of the design and test of the specific hardware and software elements of the system.

We have seen that design begins with an understanding the environment in which the system being developed will need to work then moves to translating customer requirements into a working system. Following a formal specification, under the Co-Design process, we identify the major functional blocks, and then look at ways to partition the functions onto the high-level hardware and software modules as a preliminary step leading to a mapping onto an architectural structure. With an architecture in hand, we introduced and examined tools and methods for creating a system model as input to the Co-Synthesis phase. The synthesized model became the input to the Co-Simulation phase for test, verification, and validation. We concluded with a working prototype, meanwhile, analyzing the system design both during
and after development. We have learned that design is an iterative process in which the
described Co-Design phases are repeated as necessary to ultimately yield a safe and reliable
system that meets the customer’s requirements.
We have looked at several other important considerations in the design lifecycle. These
include intellectual property, component/module reuse, requirements management and
traceability, and the archival process.