System Level Design: a Platform-Based Approach

August 26, 2010

Contents

Chapter 1

Preamble

The guiding principle in the teaching and research agenda related to embedded systems is bringing system theory and computer science closer together. These two fields have drifted apart for years while we believe that the core of embedded systems intended as an engineering discipline lies in the marriage of the two approaches. While computer science traditionally deals with abstractions where the physical world has been carefully and artfully hidden to facilitate the development of application software, system theory deals with the physical foundations of engineering where quantities such as time, power and size play a fundamental role in the models upon which this theory is based. The issue then is how to harmonize the physical view of systems with the abstractions that have been so useful in developing the computer science intellectual agenda. We argue that a novel system theory that is both computational and physical is needed. The basis of this theory must be a set of novel abstractions that partially expose the physical reality to the higher levels as well as methods to manipulate these abstractions and link them in a coherent whole. The research community is indeed developing some of the necessary results to build this novel system theory. We believe it is time to inject these findings into the teaching infrastructure so that students can be exposed to this new way of thinking. By the same token, practitioners should also be exposed to these results that advance the state of embedded system design to a point where reliable and secure distributed systems can be designed quickly, inexpensively and with no errors.

This book is intended to cover the fundamentals of embedded system design as they have been developed over the years by the research and industrial community. Being exhaustive is certainly impossible given the many important contributions and the many industrial and scientific domains the field includes. We chose to follow an organization for the contents of the book that stems out of a methodology, Platform-Based Design, that has been proposed in various forms by several people and that seems to apply well to a wide variety of design problems.

The organization of the book reflects the organization of a graduate course, *EE249, Embedded System Design: Modeling, Analysis and Synthesis* at UC Berkeley. In US Universities, bottom-up aggregation of interests and approaches to education is more common than top-down planning. Hence, education initiatives in novel areas almost always start with advanced graduate course offerings which migrate toward coordinated graduate programs and eventually into undergraduate courses. Thus, it is no wonder that course offerings at UC Berkeley on embedded systems have been strong for years in the advanced course series (the EE and CS 290 series) that are related to faculty research activities. EE249 indeed started more than ten years ago as an advanced course and then migrated in 1998 to a regular offering in the graduate program. In these past ten years, the research area and the course contents have solidified to an extent where we feel confident that they can be used at both the graduate and junior/senior levels. For this reason, we mark sections that in our opinion would be best covered in a graduate course and left out in an undergraduate course.

While the book has been designed having in mind its adoption as a textbook, we feel that it could be used as a reference book for practicing engineers as well.

We would like to acknowledge the support of our families, friends and colleagues during the writing of this book. A special thank you goes to the students who took EE249 and to the ones who helped teach the course.

Chapter 2

Introduction

This book is about *the principles of system level design*. System-Level Design (SLD) means many different things to many different people. In our view, system-level design is about the design of a whole by assembling components where specifications are given in terms of functionality (what the system is supposed to do; for example, a brake-by-wire automotive controller must actuate the braking action activated by the driver so that the wheels never lock) with:

- constraints on the properties the design has to satisfy, (for the brake-by-wire controller, the braking action must be stable), and on the components, (for the same example, the embedded micro-controller used in the implementation must be more reliable than a given threshold) and
- objective functions that express the desirable features of the design when completed (for example, low overall manufacturing cost of the controller).

This definition is general since it relates to many different application domains, from semiconductors to systems such as cars and airplanes, buildings, telecommunication and biological systems. In this book, we focus on a particular, but very wide area, embedded system design.

With the term embedded systems we refer to the electronic components (which almost always include one or more software programmable parts) of a wide variety of personal or broad-use devices, e.g., a mechanical system such as an automobile, a train, a plane, an electrical system such as an electrical motor or generator, a chemical system such as a distillation plant, a health-care equipment such as a pace-maker. Hence, an embedded system is a special-purpose system in which the computing element is completely encapsulated by the device it controls. "Unlike a general-purpose computer, an embedded system performs one or a few pre-defined tasks, usually with very specific requirements" [**?**]. In technical terms, an embedded system interacts with the surrounding environment in a controlled way satisfying a set of requirements on responsiveness in terms of quality and timeliness. Typically it has to satisfy implementation requirements such as cost, power consumed, and use of limited physical resources. Ideally its interaction with the environment should be continuously available for the entire life of the artifact.

Design tools have been important to deliver exponential increase in integrated circuit complexity with much improved designers' productivity. An entire industry, the Electronic Design Automation (EDA) industry, reached maturity in the 1980s. EDA today offers a rich tool set and flows for IC and board design. The same level of maturity has not been reached in the embedded system design tool domain.

There are some notable companies in the space, e.g., the Mathworks, but an agreed upon flow and tool set has not emerged as yet. Also the EDA industry has not yet entered in force in this adjacent market.

It is no wonder that EDA experts have stayed away from system-level design: in fact, EDA experts are essentially tied to the semiconductor industry needs in the *implementation flow* with little or no expertise in the intricacies of embedded systems that include a large amount of software and system integration concerns. The motivation for EDA experts to learn system design has not been there as yet since

- IC companies are still struggling with the understanding of higher levels of abstraction and,
- system companies have not perceived as yet design methodology nor tools to be on their critical path and hence they have not been willing to invest in "expensive" tools.

Clearly as we are hitting a wall in the development of the next generation systems, this situation is rapidly changing. Major productivity gains are needed and better verification and validation is a necessity as the safety and reliability requirements of embedded systems become more stringent and the complexity of chips is hitting an all-time high.

However, our view is that *the issue to address is not developing new tools, albeit essential to advance the state of the art in design, rather it is the understanding of the principles of system design, the necessary change to design methodologies and the dynamics of the supply chain.* Developing this understanding is necessary to define a sound approach to the needs of the system and component industry as they try to serve their customers better, to develop their products faster and with higher quality.

We share with a number of colleagues [?, ?, ?, ?, ?, ?, ?] (this list also provides an excellent set of references for state-of-the-art and directions for embedded system design) the strong belief that a new *design science* must be developed to address the challenges listed above where the physical is married to the abstract, where the world of analog signals is coupled with the one of digital processors, where ubiquitous sensing and actuation make our entire environment safer and more responsive to our needs. System Level Design (SLD) should be based on the new design science to address our needs in a fundamental way. However, the present directions are not completely clear as the new paradigm has not yet fully emerged in the design community with the strength necessary to change the EDA and Design Technology landscape albeit the papers quoted in this paragraph have chartered the field with increasing clarity.

Support for the development of this design science is given in the US by the traditional research funding organizations in collaboration with industrial associations. The Gigascale System Research Center (GSRC) [?] of the MARCO program (a joint initiative of DARPA and the Semiconductor Industry Association (SIA)) and NSF with the Center for Hybrid and Embedded Software Systems (CHESS) ITR [?] are two examples of this effort. However, a much stronger effort is afoot in Europe, where the European Community has been supporting embedded system research and novel methodologies for chip design for years with large Integrated Projects (e.g., SPEEDS) and Networks of Excellence (e.g., Artist 2 [?] and HYCON [?]) and is planning an increased effort for the VII Framework. In addition, a Technology Platform, Artemis [?], was formed three years ago by the leading European industrial companies (the initial founding group included Nokia, Ericsson, ST, ABB, Airbus, Infineon, British Telecom, Siemens, Bosch, Contiteves, Daimler-Chrysler, Thales, FIAT, Finmeccanica, Philips, COMAU, Symbian, Telenor, and PARADES with the support of research organizations such as IMEC, Verimag and the Technical University of Vienna, a healthy combination of Academia, service providers, software companies, system, subsystem and semiconductor manufacturers). The companies have recently formed the Artemis Industrial Association (ARTEMISIA), while the European community decided in 2008 to make Artemis a Joint Technology Initiative, an instrument to funnel substantial resources of the member States. In a meeting of the European Community Prime Ministers on October 2006, Artemis was quoted by some of the participants (in particular, the Finnish Prime Minister) as an example of agenda setting initiative for the industrial future of Europe.

In the following Section, we focus on the pressing concerns of system level design together with the strategic and business concerns in the supply chains of the mobile terminal and automotive vertical domains as examples of the issues to be taken into consideration when we think about expanding the reach of design methodology and tools.

2.1 Setting the Stage: Challenges of System Level Design

In the present technology environment and industrial structure, SLD has to address concerns of individual players in the industrial domain that are facing serious problems in bringing their products to market in time and with the required functionality. I do believe that SLD also needs to be concerned about the entire industrial supply chain that span from customer-facing companies to subsystem and component suppliers, since the health of an industrial sector depends on the smooth interaction among the players of the chain as if they were part of the same company. In this section, I present a view on both challenges that underlines commonalities that allow a unified approach to SLD.

2.1.1 Managing Complexity and Integration

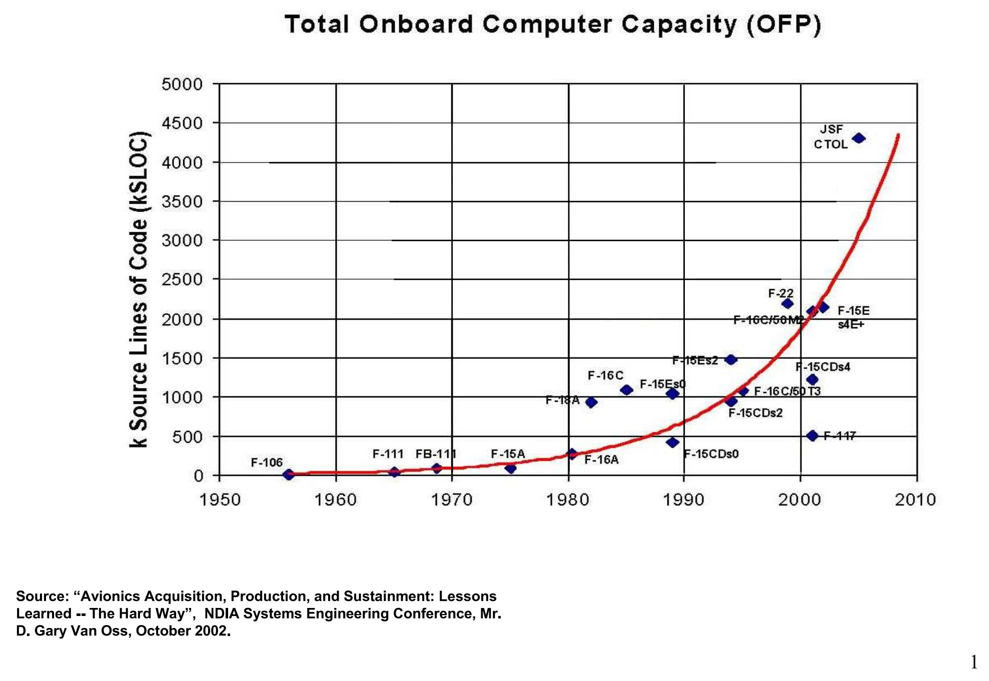
The ability of integrating an exponentially raising number of transistors within a chip, the ever-expanding use of electronic embedded systems to control increasingly many aspects of the "real world", and the trend to interconnect more and more such systems (often from different manufacturers) into a global network, are creating a challenging scenario for embedded system designers. Complexity and scope are exploding into the three inter-related but independently growing directions, while teams are even shrinking in size to further reduce costs. In this scenario the three challenges that are taking center stage are:

Heterogeneity and Complexity of the Hardware Platform

The trends mentioned above result in exponential complexity growth of the features that can be implemented in hardware. The integration capabilities make it possible to build real complex system on a chip including analog and RF components, General Purpose Processors (GPP) and Application Specific Instruction-set Processors (ASIP). The decision of what goes on a chip is no longer dictated by the amount of circuitry that can be placed there, but by reliability, yield, power consumption, performance and ultimately cost (it is well known that analog and RF components force the use of more conservative manufacturing lines with more processing steps than pure digital ICs). Even if manufacturing concerns suggest to implement hardware in separate chips, the resulting package may still be very small given the advances in packaging technology yielding the concept of System-in-Package (SiP). Pure digital chips are also featuring an increasing number of components. Design time, cost and manufacturing unpredictability for deep sub-micron technology make the use of custom hardware implementations appealing only for products that are addressing a very large market and for experienced and financially rich companies. Even for these companies, the present design methodologies are not yielding the necessary productivity forcing them to increase beyond reason the size of design and verification teams. These IC companies (for example Intel, Freescale, ST and TI) are looking increasingly to system design methods to allow them to assemble large chips out of pre-designed components and to reduce validation costs (design re-use). In this context, the adoption of design models above RTL and of communication mechanism among components with guaranteed properties and standard interfaces is only a matter of time.

Embedded Software Complexity

Given the cost and risks associated to developing hardware solutions, an increasing number of companies is selecting hardware platforms that can be customized by reconfiguration and/or by software programmability. In particular, software is taking the lion's share of the implementation budgets and cost. In cell phones, more than 1 Million lines of code is standard today, while in automobiles the estimated number of lines by 2010 is in the order of hundreds of Millions [?]. The number of



Source: "Avionics Acquisition, Production, and Sustainment: Lessons Learned -- The Hard Way", NDIA Systems Engineering Conference, Mr. D. Gary Van Oss, October 2002.

Figure 2.1: Software Growth in Avionics.

lines of source code of embedded software required for defense avionics systems is also growing exponentially as reported on Figure 2.1 (made available by R. Gold, Robert Gold Associate Director, Software and Embedded Systems, Office of the Deputy Under Secretary of Defense (Science and Technology)). However, as this happens, the complexity explosion of the software component causes serious concerns for the final quality of the products and the productivity of the engineering teams. In transportation, the productivity of embedded software writers using the traditional methods of software development ranges in the few tens of lines per day. The reasons for such a low productivity are in the time needed for verification of the system and long redesign cycles that come from the need of developing full system prototypes for the lack of appropriate virtual engineering methods and tools for embedded software. Embedded software is substantially different from traditional software for commercial and corporate applications: by virtue of being embedded in a surrounding system, the software must be able to continuously react to stimuli in the desired way, i.e., within bounds on timing, power consumed and cost. Verifying the correctness of the system requires that the model of the software be transformed to include information that involve physical quantities to retain only what is relevant to the task at hand. In traditional software systems, the abstraction process leaves out *all* the physical aspects of the systems as only the functional aspects of the code matter.

Given the impact that embedded software has on the safety of embedded system devices and on the quality of the final artifact, there is an increasingly strong interest in having high assurance that embedded software is correct. *Software certification* demonstrates the reliability and safety of software systems in such a way that it can be checked by an independent authority with minimal trust in the techniques and tools used in the certification process itself. It builds on existing software assurance, validation, and verification techniques but introduces the notion of explicit software certificates, which contain all the information necessary for an independent assessment of the properties. Software certification has been required by military applications for years and has been recently extended to the US aviation sector: the FAA accepted the DO-178B regulations as the means of certifying all new aviation software. A joint committee with the European authorities has been recently empowered to "promote safe implementation of aeronautical software, to provide clear and consistent ties with the systems and safety processes, to address emerging software trends and technologies, and to implement an approach that can change with the technology"[**?**, **?**]. We believe that certification will expand into new safety-critical domains and will create an additional, serious, burden on the embedded software design process not only for the aviation industry but for an increasingly large number of companies worldwide. Note that today, the main scope of the certification process relates to the process followed to develop software. We believe it will be of extreme importance to link the certification process with the *content* of the software and not only with the development process. This approach will have to include formal verification techniques as we believe this is the only way to increase the confidence in the correctness of the software.

Integration Complexity

A standard technique to deal with complexity is decomposing "top-down" the system into subsystems. This approach, which has been customarily adopted by the semiconductor industry for years, has limitation as a designer or a group of designers has to fully comprehend the entire system and to partition appropriately its various parts, a difficult task given the enormous complexity of today's systems. Hence, the future is one of developing systems by *composing* pieces that all or in part have already been pre-designed or designed independently by other design groups or even companies. This has been done routinely in vertical design chains for example in the avionics and automotive verticals, albeit in a heuristic and *ad hoc* way. The resulting lack of an overall understanding of the interplay of the sub-systems and of the difficulties encountered in integrating very complex parts causes system integration to become a nightmare in the system industry. For example, Jurgen Hubbert, then in charge of the Mercedes-Benz passenger car division, publicly stated in 2003: *"The industry is fighting to solve problems that are coming from electronics. Companies that introduce new technologies face additional risks. We have experienced blackouts on our cockpit management and navigation command system and there have been problems with telephone connections and seat heating."*

I believe that in today's environment this state is the rule rather than the exception for the leading system Original Equipment Manufacturers (OEMs)[1] in all industrial sectors [?]. The source of these problems is clearly the increased complexity but also the difficulty of the OEMs in managing the integration and maintenance process with subsystems that come from different suppliers who use different design methods, different software architecture, different hardware platforms, different (and often proprietary) Real-Time Operating Systems. Therefore, there is a need for standards in the software and hardware domains that will allow plug-and-play of sub-systems and their implementation. The ability to integrate subsystems will then become a commodity item, available to all OEMs. The competitive advantage of an OEM will increasingly reside on novel and compelling functionalities.

There is also the need of improving the interaction among all the players in the supply chain to improve the integration challenges in a substantial way as I argue in the next section.

2.2 The Industrial Supply Chain Landscape

The Design and Supply chains are the backbone for any industrial sector. Their health and efficiency are essential for economic viability. While tools for Supply Chain management have been around for quite some time, support for the Design Chain has not been pursued nearly as vigorously. There are great opportunities for improving the situation substantially at least in the safety-driven industrial sector, which includes the transportation as well as industrial automation domain, with a combination of tools and methodologies. We are just at the beginning.

Integration of electronic and mechanical design tools and frameworks will be essential in the near future. Integration of chemical, electronic and biology tools will also be essential in the further future for nano-systems. Data integration and information flow among the companies forming the chain have to be supported. In other words, it is essential that the fundamental steps of system design (functional partitioning, allocation on computational resources, integration, and verification) be supported across the entire design development cycle. Thus, whether the integrations pertains to SW-SW integration on a distributed network, HW-SW integration on a single Electronic Control Unit (ECU), electronics and mechanical integration for a sub-system, tools and models have to be integrated seamlessly from a static point of view (e.g., data dictionaries and off-line model transformations)

[1]In this paper, OEM is used to refer to the companies that acquire a product or component and reuse or incorporate it into a new product with their own brand names. Examples are Mercedes, GM and Toyota, Boeing and Airbus.

and dynamic point of view (e.g., co-simulation, HW-in-the-loop simulations and emulation).

Assuming the design methodology and the infrastructure for design chain integration are all in place, what will be the implication on the industrial structure?

The dynamics of the system industry is *similar across the vertical domains but certainly there are important differences among them.* For example, for embedded controllers in industrial engineering applications, automotive, avionics, energy production and health related equipment, safety considerations and hence hard real-time constraints, are of paramount importance. In the case of consumer electronics, including entertainment subsystems in cars and airplanes, cell phones, cameras and games, the concerns are on sound, video and recording quality and on the look and feel of the devices in presence of severe cost constraints. I will briefly discuss the cell phone design chain and the automotive design chain as the representatives of the embedded system market dynamics.

The Mobile Communication Design Chain

The cell phone industrial segment is a complex ecosystem in continuous evolution with the following actors:

Application developers who offer products such as gaming, ring tones and video . These companies sell their products directly to the end customer except in cases where these applications come bundled either with standard services like voice offered by service providers such as Cingular, Verizone or Telecom Italia, or with the device itself offered by makers such as Nokia, Motorola, Samsung or Ericsson. Their designs are in general implemented in software running on the platform provided by the device manufacturers who choose also the OS.

Service providers who offer the access to the network infrastructure for voice and data. These providers also offer other services such as news, weather information, and traffic. The GSM standard introduced (and mandated) the use of the Subscriber Identity Module (SIM), a smart card that securely stores the key identifying a mobile phone service subscriber, as well as subscription information, saved telephone numbers, preferences, text messages and other information. The use of the SIM card is important in the dynamics of the vertical segment as it is under control of the service providers. The service provider technology relates to the management of the infrastructure and of the service delivery. They worry, for example, about communication hand-offs when cell boundaries are traversed and base-station location.

Device makers who manufacture the mobile terminal, e.g., the cell phone. Device makers must master a number of different technologies as they manufacture systems with significant software content (more than 1 million lines of code today) and hardware content including computing and communication circuitry involving analog and RF. In most cases, the IC content is obtained by chip manufacturers such as Qualcomm, TI, Freescale and ST, but it may also be designed by captive teams. One of the many challenges of a mobile terminal manufacturer is integrating heterogeneous semiconductors manufactured by different companies (for example, DSPs and microcontrollers for the digital part, base-band and RF circuitry) whose interaction must be accurately predicted and controlled to provide the functionality with no errors. There is a significant IP content acquired by middleware software providers such as the Symbian OS, an operating system designed for mobile devices, with associated libraries, user interface frameworks and reference implementations of common tools, produced by Symbian Ltd. In addition, styling, ergonomics and user-friendliness are major attractions for the end customer.

IC providers who offer semiconductors and other IPs that implement a variety of a mobile terminal functions. Semiconductor technology has had a major impact in the diffusion of mobile terminals as it is responsible for the dimension, power consumption, performance, functionality and cost of the terminal. Because of the complexity of the design and of the need of interfacing with other vendors, IC manufacturers have turned to a particular design style that is the major content of this paper, platform-based design. The TI OMAP [?] platform together with the Nexperia Philips platform for digital video are the first examples of complex semiconductors designed in this style. Given the sale volumes of mobile terminals, IC manufacturers are competing fiercely and to provide the features needed by the device manufacturers, they had to enter into system level design and into the development of significant software components including device drivers and other middleware. The semiconductor manufacturers are themselves integrating third party IPs.

IP providers who provide components to the rest of the design chain. Symbian (with its OS for cell phones), Microsoft (with Windows CE), and ARM (with its processors) are examples of IP providers. These components are integrated in the semiconductors or in the terminal to perform an important function. They are instrumental to the functioning of the devices but cannot be sold to the end customer *per se*.

Outsourcing companies who provide design and manufacturing services to the rest of the chain. For example, Flextronics provides manufacturing services to a large variety of companies in the system domain including mobile terminal manufacturers. E-silicon [?] in US, Accent [?] in Europe and Faraday [?] in Taiwan offer design services to semiconductor and device manufacturers for part or entire chips as well as brokerage services to manage the interactions with silicon foundries. Finally, semiconductor foundries such as TSMC [?], IBM [?] and UMC [?] provide IC manufacturing services

Today, there is a great deal of competition and turf battles to determine where the added value is inserted. For example, the boundary between service providers and device makers as well as the one between device and IC makers is under stress. Service providers favored the SIM card as a way of capturing value in their products and defend it against the device makers. The standard that limits the communication bandwidth between SIM cards and the cell phone electronics defends the device makers turf against the intrusion of the service providers. The device makers defend their added value against IC manufacturers by avoiding being locked into a single provider situation, farming out different components to different companies. In addition, they force whenever possible the IC providers to use standards that favor the possibility of using different IPs as they see fit. The use of the Open Core Protocol [?] standard in the TI OMAP [?] platform is a case where the interest of the device makers and the one of the IC provider aligned since it was also the interest of the IC provider to be able to incorporate quickly external and internal IPs. My opinion is that providing a unified methodology and framework, we will favor the balance of the chain where everyone reaches an equilibrium point that maximizes the welfare of the system.

The Automotive Design Chain

The need of integrating widely different subsystems such as safety, propulsion, communication and entertainment, makes this vertical very interesting for our purposes.

Today, the roles of car makers and their suppliers are relatively stable but they are undergoing a period of stress due to the increased importance of electronics and its added value. The Automotive supply chain includes

Car manufacturers (OEMs) such as GM, Ford, Daimler-Chrysler and Toyota, who provide the final product to the consumer market.

Tier 1 suppliers such as Bosch, Contiteves, Siemens, Nippon Denso, Delphi and Magneti-Marelli, who provide subsystems such as powertrain management, suspension control and brake-by-wire devices to OEMs,

Tier 2 suppliers e.g., chip manufacturers such as Freescale, Infineon, ST, and Renesas, IP providers e.g.,ARM and RTOS suppliers such as WindRiver and ETAS, who serve OEMs and more likely Tier 1 suppliers.

Manufacturing suppliers such as Flextronics and TSMC who provide manufacturing services. Opposite to verticals that are not safety critical, liability issues make the recourse to outside manufacturing not as common. However, there are signs that manufacturing for Tier 1 suppliers is increasingly considered for outsourcing.

Car makers express the desire of gaining a stronger grip on the integration process and on the critical parts of the electronics subsystems. At the same time, there is evidence that sharing IPs among car makers and Tier 1 suppliers could improve substantially time-to-market, development and maintenance costs. The essential technical problem to solve for this vision is the establishment of standards for interoperability among IPs and tools. AUTOSAR [**?**], a world-wide consortium of almost all players in the automotive domain electronics supply chain, has this goal very clear in mind. However, there are technical and business challenges to overcome. In particular, from the technical point of view, while sharing algorithms and functional designs seems feasible at this time, the sharing of *hard real-time* software is difficult even assuming substantial improvements in design methods and technology, if run-time efficiency has to be retained. The issues are related to the interplay that different tasks can have at the RTOS level. The timing of the software tasks depend on the presence or absence of other tasks. A scheduling policy that could prevent timing variability in presence of dynamical changing task characteristics can be conceived (for example, timing isolation or resource reservation policies) but it will carry overhead, albeit potentially not prohibitive; further, this kind of policy is not supported by any of the commercially available RTOS. This situation is the standard trade-off between efficiency and reliability but it has more important business implications than usual. In fact, if software from different sources has to be integrated on a common hardware platform, in absence of composition rules and formal verification of the properties of composed systems, who will be responsible for the correct functioning of the final product?

Whoever will take on this responsibility would need a very strong methodology and an iron fist to make suppliers and partners comply with it. This may not be enough, in the sense that software characteristics are hard to pin down and with the best intentions of this world, one may not be able to guarantee functional *and* timing behavior in the presence of foreign components. The constant growth of complexity of the embedded systems designed today makes manual analysis and design impractical and error prone. The ideal approach would be a tool that could map automatically the set of tasks onto the platform guaranteeing the correct functionality and timing with optimal resource utilization [**?**]. This tool should take the design description at the pure functional level with performance and other constraints and the architecture of the platform and produce correct settings for the RTOS and optimized code. We are still far from this ideal situation. It is likely, then, that the responsibility for subsystem integration will still rest with the car manufacturers but the responsibility for integrating software components onto ECUs will be assigned to Tier 1 suppliers. In this case, the burden of Tier 1 suppliers will be increased at a possibly reduced premium because of the perceived reduction in added value. This is likely to be an unstable model and major attention should be devoted to find a common ground where both car makers and suppliers find their economic return.

If the strategy followed by car makers in AUTOSAR succeeds, then it is likely that a global restructuring of the industry will take place by creating an environment where Tier 1 players with small market share will find themselves in a difficult position unless they find a way of competing on a more leveled ground with the major stake holders. In this scenario, Tier 2 suppliers including IP providers may find themselves in a better position to entertain business relations directly with the car manufacturer. Tool providers will be in a more strategic position as providers of mapping tools that make the business model feasible. Hence, it is likely that a shift of recognized value will take place from Tier 1 suppliers towards tool providers and Tier 2 suppliers. The redistribution of wealth in the design chain may or may not be a positive outcome for the health of the industrial sector. If the discontinuities are sharp, then there may be a period of instability where much effort will be required to keep the products coming out with quality and reliability problems that may be larger than the ones observed lately. However, if it is well managed, then a natural shake-up with stronger players emerging will have a double positive: more quality in the products at lower cost. An additional benefit from a real plug-and-play environment will be the acceleration of the rate of innovation. Today, the automotive sector is considered conservative and the innovations in design methods and electronic components are slow to come. For example, if a well-oiled mechanism existed to migrate from one hardware platform to another, the "optimal" solutions would be selected instead of the ones that have been traditionally used. In this case, the Tier 2 market place will also be rationalized and the rate of innovation will likely be increased.

As a final consequence, the introduction of new functionalities will be a matter of algorithm and architecture rather than detailed software and hardware selection. The trend in electronics for automotive (but for other verticals as well) is clear: less customization, more standardization. For a subsystem supplier, the choice will be richer in terms of platforms but it will not require heavy investment in IC design or RTOS development. For car manufacturers, the granularity of the choices will be also richer because of interoperability. They will have the choice of selecting entire macro systems or components that could be integrated in a large automotive platform. The choice will be guided by cost, quality and product innovation.

The final goal of the strategy is rather clear. The way of getting there is not as clear and the road has many bumps and turns that are difficult to negotiate. A positive outcome will have to come from a process of deep business and technical cooperation among all players in the design chain as well as the research community. It is a unique opportunity and a great challenge.

Remarks on the Needs of the Supply Chains

The design chains should connect seamlessly to minimize design errors and time-to-market delays. Yet, the boundaries among companies are often not as clean as needed and design specs move from one company to the next in non executable and often imprecise form thus yielding misinterpretations and consequent design errors. In addition, errors are often caught only at the final integration step as the specifications were not complete and imprecise; further, non functional specifications (e.g., timing, power consumption, size) are difficult to trace. I believe also that since the design process is fragmented, product optimization is rarely carried out across more than one company boundary. If the design process were carried out as in a unique "virtual" company including all the players shown above, the overall ecosystem would greatly benefit. We have seen that many of the design chain problems are typical of two very diverse verticals, the difference between the two being in the importance given to time-to-market and to the customer appeal of the products versus safety and hard-time constraints. Similar considerations could be drawn also for the consumer electronic market at large that shares many of its characteristics with the wireless communication market. This consideration motivates the view that the unified methodology and framework could be used in several (if not all) industrial vertical domains.

2.3 Organization of the Book

We present our view on how to form a unified approach to embedded system design, Platform-Based Design, that could provide a solution to the challenges presented in the previous sections (Section **??**). In Section.....

In Section **??**, we draw conclusions and indicate future directions for research and industrial developments.

Notably missing from this paper is testing. The topic is extremely important for SLD but to do justice to it, an entire new book would be needed.

Chapter 3

The Principles of a Unified Design Methodology

3.1 Introduction

Most of the present approaches to SLD used in industry have the drawback of primarily addressing either hardware or software but not both at the same time. Hardware/software co-design has been a topic of interest for years, but the proposed methodologies have still treated the two aspects essentially in a segregated way. Software approaches miss time and concurrency in their semantics making it pretty much impossible to describe, synthesize and verify hardware. Hardware approaches are too specific to the hardware semantics to work well for software designers. We also believe that the levels of abstraction available in these approaches are not rich enough to allow the supply chain to exchange design data in a seamless fashion.

These drawbacks cause the presently available approaches to address some of the challenges presented in Section 2.1 but not all, failing especially in the integration and supply chain-support domain. A more powerful approach would be to use an all-encompassing methodology and the supporting tools that:

- would include both hardware and embedded-software design as two faces of the same coin;
- favor the use of high levels of abstraction for the initial design description;
- offer effective architectural design exploration and
- achieves detailed implementation by synthesis or manual refinement.

In this chapter, we present the *Platform-Based Design (PBD)* methodology, and argue that it meets these requirements.

3.2 Platforms

The concept of "platform" has been around for years. The main idea of a platform is one of re-use and of facilitating the work of adapting a common design to a variety of different applications. Several papers and books have appeared in the literature discussing platforms and their use in embedded system design (see for example, [?, ?, ?, ?, ?, ?, ?, ?, ?].

In this section, we first introduce the use of the platform concept in industry, then we present a distilled way of considering platforms as the building blocks for a general design methodology that could be used across different boundaries.

3.2.1 Conventional Use of the Platform Concept

There are many definitions of "platform" that depend on the domain of application.

IC domain: a platform is considered a flexible integrated circuit where customization for a particular application is achieved by programming one or more of the components of the chip. Programming may imply metal customization (gate arrays), electrical modification (FPGA personalization), or software to run on a microprocessor or a DSP. For example, a platform may be based on a fixed micro-architecture to minimize mask-making costs, but flexible enough to warrant its use for a set of applications so that production volume will be high over an extended chip lifetime. Micro-controllers designed for automotive applications such as the Freescale PowerPC are examples of this approach. The problem with this approach is the potential lack of optimization that in some applications, may make performance too low and size too large.

An extension of this concept is a "family" of similar chips that differ for one or more components but that are based on the same microprocessor(s). Freescale developed the Oak Family [**?**] of PowerPC-based micro-controllers that cover the market more efficiently, by differing in flash memory size and peripherals. The TI OMAP platform [**?**] for wireless communication[1] was indeed developed with the platform concept well in mind. Jean-Marc Chateau of ST Microelectronics commenting on its division commitment to platform-based design defines it "as the creation of a stable microprocessor-based architecture that can be rapidly extended, customized for a range of applications, and delivered to customers for quick deployment." .

The use of the platform-based design concept actually started with the Phillips Nexperia Digital Video Platform (DVP). The concept of platform-based design for IC design has not been without its critics. Gary Smith, the former Gartner Data Quest Analyst for CAD, pointed out a number of shortcomings [**?**] that make, in his words, platform-based design work well in an embedded software development context as advocated in [**?**] but not so for chip design. However, not a month later, in an interview [**?**, **?**], McGregor, former CEO of Philips semiconductors was quoted: " .. we redoubled the company's efforts in platform-based design. Philips embraced the idea early in the mid-'90s, The recommitment to the platform approach under my watch is among my most notable accomplishments". In another important quote: "ST's Geyres attributed ST's continued success in the set-top business to its migration from systems-on-chip to application platforms." [**?**]. At this time, there is little doubt that platform-based design has made significant inroads in any semiconductor application domain. The Xilinx Virtex II [**?**] family is a platform rich in flexibility offered by an extensive FPGA fabric coupled with hard software programmable IPs (up to four PowerPC cores and a variety of peripherals. The FPGA fabric is enriched by a set of "soft" library elements such as the micro-blaze processor and a variety of smaller granularity functional blocks such as adders and multipliers.

We believe there will be a converging path towards the platform of the future,

[1]From the TI home page: " TIs OMAP Platform is comprised of market proven, high-performance, power efficient processors, a robust software infrastructure and comprehensive support network for the rapid development of differentiated internet appliances, 2.5G and 3G wireless handsets and PDAs, portable data terminals and other multimedia-enhanced devices."

where traditional semiconductor companies will increase the flexibility of their platforms by possibly adding FPGA-like blocks and heterogeneous programmable processors, while the FPGA-based companies will make their platforms more cost and performance efficient by adding hard macros thus differentiating their offerings according to the markets of interest. The more heterogeneity is added to the platform, the more potential for optimizing an application at the price of a more complex design process for the application engineers who have to allocate functionality to the various components and develop code for the programmable parts. In this context, the interaction among the various components has problems similar to those faced by the system companies in an inherently distributed implementation domain (e.g., cars, airplanes, industrial plants). The "right" balance among the various components is difficult to strike and the methodology we will describe later is an attempt to give the appropriate weapons to fight this battle.

PC Domain: PC makers and application software designers have been able to develop their products quickly and efficiently around a standard "platform" that emerged over the years. The "architecture" platform standards can be summarized in the following list:

- The x86 instruction set architecture (ISA) that makes it possible to re-use the operating system and the software application at the binary level3.
- A fully specified set of busses (ISA, USB, PCI) that make it possible to use the same expansion boards or IC's for different products.
- A full specification of a set of I/O devices, such as keyboard, mouse, audio and video devices.

All PCs should satisfy this set of constraints. Both the application developers and the hardware designers benefited from the existence of a standard layer of abstraction. Software designers have long used well-defined interfaces that are largely independent from the details of the hardware architecture. IC designers could invent new micro-architectures and circuits as long as their designs satisfied the standard. If we examine carefully the structure of a PC platform, we note that it is not the detailed hardware micro-architecture that is standardized, but rather an abstraction characterized by a *set of constraints on the architecture*. The platform is an abstraction of a "family" of micro-architectures. In this case, IC design time is certainly minimized since the essential components of the architecture are fixed and the remaining degrees of freedom allow some optimization of performance and cost. Software can also be developed independently of the new hardware availability, thus offering a real hardware-software co-design approach.

System Domain: The definition of a platform is very loose. This quote from an Ericsson press release is a good example: "Ericsson's Internet Services Platform is a new tool for helping CDMA operators and service providers deploy Mobile Internet applications rapidly, efficiently and cost-effectively." The essential concept outlined here is the aspect of the capabilities a platform offers to develop quickly new applications. It is similar to the application software view of a PC platform, but it is clearly at a higher level of abstraction. The term platform has been also used by car makers to indicate the common features shared between different models. For automobiles, platforms are characterized by common mechanical features such as engines, chassis, and entire powertrains. It is not infrequent to see a number of different models even across brands share many mechanical parts, addressing different markets with optimized interior and styling. Here the focus on subsystem commonality allows for faster time-to-market and less expensive development.

There are clearly common elements in the platform approaches across industrial domains. to make platforms a general framework for system design, a distillation of the principles is needed so that a rigorous methodology can be developed and profitably used across different design domains.

3.3 The Platform-Based Design Methodology

The principles at the basis of platform-based design consist of starting at the highest level of abstraction, hiding unnecessary details of an implementation, summarizing the important parameters of the implementation in an abstract model, limiting the design space exploration to a set of available components and carrying out the design as a sequence of "refinement" steps that go from the initial specification towards the final implementation using platforms at various level of abstraction [**?**, **?**, **?**, **?**].

3.3.1 Platform Definition

A *platform* is defined to be a *library of components* that can be assembled to generate a design at that level of abstraction.

Example 3.3.1 (Libraries) Hardware designers are familiar with the notion of libraries. Figure 3.4 shows the hardware design flow applied to a simple example. The specification of the logic function to implement is captured using a high level description language such as VHDL. This description is then translated into a *boolean network*. For the sake of this example, we selected a combinational function but the same design flow is used for sequential functions. The boolean function is *minimized* to reduce the number of gates necessary for its implementation. At this point, the original description looks like a set of interconnected gates. These gates are covered using a *library* of standard gates called *standard cells*. Each foundry provides its own library of standard cells. Each cell is characterized by metrics such as area, power, input and output capacitance. Together with the library, a set of rules are also provided to interconnect the cells together. For example, there are specific rules that establish the maximum number of cells that can be connected to the output of one given cell of a give size. The standard cell library, together with the rules to compose them, define the class of logic functions that can be implemented. Also, the same function can be implemented in multiple ways for different area-timing-power trade-offs.

Software designers also rely on a platform. An operating system comes with a rich set of libraries that include scheduling policies, inter-task communication, timing and power management, network access and security. At a higher level of abstraction, the operating system can be seen as one component of a library of software modules. Depending on the application domain, other components are available to software designers such as algorithms for signal and image processing and application level protocols.

This library not only contains *computational* blocks that carry out the appropriate computation but also *communication* components that are used to interconnect the computational components.

Example 3.3.2 (Communication components) We discuss here a few example of communication components. The interconnection of functional components using communication elements is also referred to as *network*. At the algorithmic description level, communication is usually point-to-point or through shared variables. In Chapter **??** we present several formal ways of capturing the behavior of a system, called models of computation. Each model defines computational elements and the way in which they can be composed. In fact, the definition of a model of computation also includes the definition of the communication semantics. A simple example is the model used for signal processing applications. In this model, computation blocks perform sub-functions of an algorithm and exchange intermediate results by communicating over First-Input-First-Output (FIFO) queues. According to our framework, this corresponds to having a library of two types of elements: functional blocks and FIFO queues.

At a lower abstraction level, data networks are used to exchange data among components. Point-to-point communication is till possible at this level and it is indeed very common in hardware design. However, the adoption of networked solution offers many advantages by providing flexible connectivity at a lower cost. This is achieved by sharing communication media. In bus-based communication, the same physical connection is accessed by a number of bus participant that, to get the ownership of the bus, must go thorough an arbitration phase. In packet-switched networks, components send packets that are routed in the network by routers and switches. Together with all these components come a set of rules for their composition that span all levels of the protocol stack, from the physical communication to the application layer.

One example of communication platform for System-on-Chips is the AMBA bus defined by ARM. The AMBA specification defines four different components:

- the Advanced eXtensible Interface (AXI) protocol suitable for high-bandwidth and low latency designs;
- the Advanced High Performance Bus (AHB) to interconnect high-performance, high clock frequency components;
- the Advanced System Bus (ASB) also for high-performance components but with limited features compared to the AHB;
- and the Avanced Peripheral Bus (APB) to interconnect low power peripheral with simple interfaces.

The interface of AMBA compliant components is defined by the standard as a list of bit typed signals. The high-performance bus and the peripheral bus can be composed through bridges.

It is important to keep communication and computation elements well separated as we may want to use different methods for representing and refining these blocks. For example, communication plays a fundamental role in determining the properties of models of computation. In addition, designing by aggregation of components requires a great care in defining the communication mechanisms as they may help or hurt design re-use. In design methodologies based on IP assembly, communication is the most important aspect. Unexpected behavior of the composition is often due to negligence in defining the interfaces and the communication among the components.

Example 3.3.3 (Orthogonalization of Computation and Communication) In Example **??** we already discussed the importance of the communication semantics as an integral part of the definition of the meaning of a model. For the same semantics of the computation blocks, the communication semantics can change the properties of a model quite radically. This concept is very important to be able to re-use components and it is also exploited in tools that support heterogeneous design such as Ptolemy and Metropolis. To give a concrete example, consider processes running on their own threads that communicate using two services:

- `out.write(token t)` that is used to write an atomic piece of data `t` of a generic type `token` on an output port `out`, and
- `token in.read()` that reads a token from an input port `in`.

The implementation of the read and write functions are not defined by the processes. A process simply assumes that these services are provided by the components attached to its ports.

Figure 3.1 shows a network of processes communicating over point-to-point channels. The left side of the figure shows some details of implementation of execution semantics of the processes. Each process runs a thread that is an infinite loop. Consider process `P1`. Within its loop, this process reads one token from its input, does some computation, and writes four token on port `out1` and one token on port `out2`. The behavior of the other processes is very similar: they will read tokens from their inputs, do some computation, and write the results to their output ports. The consumption and production rate of each process are written next to their input and output ports. The two branches of the network of processes are unbalanced because `P4` reads one token from each port, making the rate of the upper branch too high for `P4` to keep the buffers empty. However, assuming that the buffers on the point-to-point connections are unbounded (i.e. they can hold an infinite number of tokens), each process can run forever.

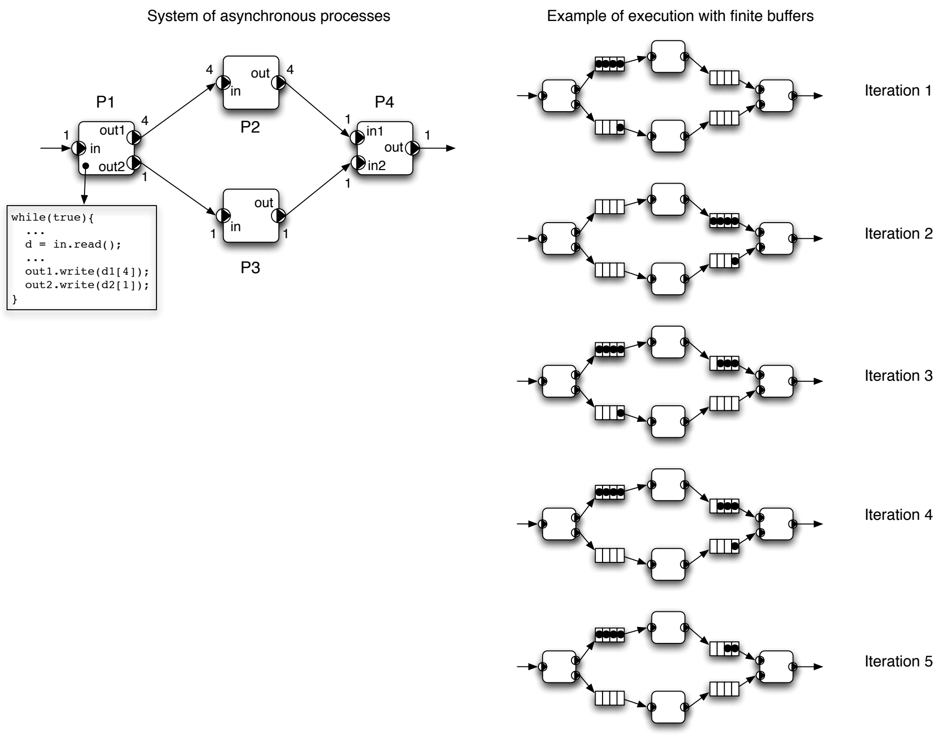
Consider the same network of processes but with finite buffers on the communication channels. The communication semantics is now different. Because the buffers are finite, a process attempting to write on a buffer will be blocked if there is not enough space to complete the writing operation. The right side of Figure 3.1 shows an execution of the same network of processes with finite buffers of length four. At each iteration, all the processes that can proceed will do so. However, after the fifth iteration, no process can make any progress because `P4` is waiting for tokens on input `in2` but `P1` cannot execute its writing operation and, consequently, it cannot feed any more token to the lower branch of the network.

The separation of computation and communication allows for instance several different applications on a computer to access the network. Moreover, the type of access, either wired through an Ethernet cable or wireless through a WiFi connection, does not affect the application.

Each element of the library has a characterization in terms of performance parameters together with the functionality it can support.

Example 3.3.4 (Quantities) A given service, or functionality, can be provided, or implemented, in many ways. Thus, the same functional specification of a system can be implemented by different interconnections of the components of a library. First, each implementation need to provide the required functionality but it also need to satisfy performance constraints. Second, each implementation has a cost. We generalize performance and cost metrics into quantities that characterize a component. The quantities are used to explore the trade-off among different solutions.

For example, the same standard cell library (e.g. a OR gate) can be implemented with different power-area trade-offs. Larger cells can provide higher output current decreasing the switching time and ultimately leading to faster circuits. On the other hand, larger cells consume more power which is of primary concern for battery powered devices.



```
while(true){
  ...
  d = in.read();
  ...
  out1.write(d1[4]);
  out2.write(d2[1]);
}
```

Figure 3.1: Two processes communicating on two different channels.

The library is in some sense a parametrization of the space of possible solutions. Not all elements in the library are pre-existing components. Some may be "place holders" to indicate the flexibility of "customizing" a part of the design that is offered to the designer . For example, in a Xilinx Virtex II Field Programmable Gate Array (FPGA) platform, part of the design may be mapped to a set of virtual gates using logic synthesis and place-and-route tools. For this part, we do not have a complete characterization of the element since its performance parameters depend upon a lower level of abstraction.

Example 3.3.5 Figure 3.2 shows an example of a system mapped on a programmable platform. The system is an abstract example of a device that provides a set of applications and network connectivity. For example, one of the applications sends a video stream captured by a camera on a communication channel. To send data over the network, an application writes the data in the input queue of a function that implements the higher levels of the network protocol stack. The data is encapsulated in a protocol frame that is sent to another function implementing the baseband processing (i.e. the signal processing that transforms a stream of bits into a signal ready to be sent to a digital-to-analog converter and then to the radio).

The Virtex II platform is an FPGA with an embedded processor. The embedded processor can be programmed using standard software development tools. Hardware blocks can be implemented on the FPGA using a hardware design flow. In our example, the MPEG decoder and the baseband processing blocks may be too complex to be executed by the microprocessor. These two blocks will be implemented in hardware. Xilinx also provide a large set of hardware blocks that are optimized for FPGA implementation. This library also contains an MPEG decoder that is well characterized in terms of performance, power consumption and area. The baseband processing block has to be implemented in a custom way in hardware. At this abstraction level, we do not know what will be the performance of such block. Therefore, we define a virtual component and assign the performance metrics that will make our system work correctly. For instance, if we want to transmit at a rate of 2 Mb/s, the total processing time per bit of the virtual component must be half millisecond. The performance assigned to the virtual component becomes the constraint that must be satisfied by the hardware implementation of the baseband block.

A *platform instance* is a set of components that are selected from the library (the platform) and whose parameters are set. In the case of a virtual component, the parameters are set by the requirements rather than by the implementation. In this case, they have to be considered as constraints for the next level of refinement.

Example 3.3.6 In Example 3.3.2 we described the AMBA platform as a set of busses that can be instantiated to connect the cores of a system-on-chip. A particular instantiation and composition of AMBA busses, together with an assignment of their parameters is a platform instance.

Figure 3.3 shows a SoC based on the AMBA platform. The CPU core and the on-chip memory are connected to a AHB. The external memory interface and a Direct Access Memory controller are also cores that require high performance and they are also connected to the AHB. The peripheral cores are instead connected to a APB. The two busses communicate via a bus bridge. The platform instance is not only a particular composition of library elements, but also these components are configured to serve the particular application that is mapped on the platform instance. For example, the width of the data bus in number of bits and the clock speed are

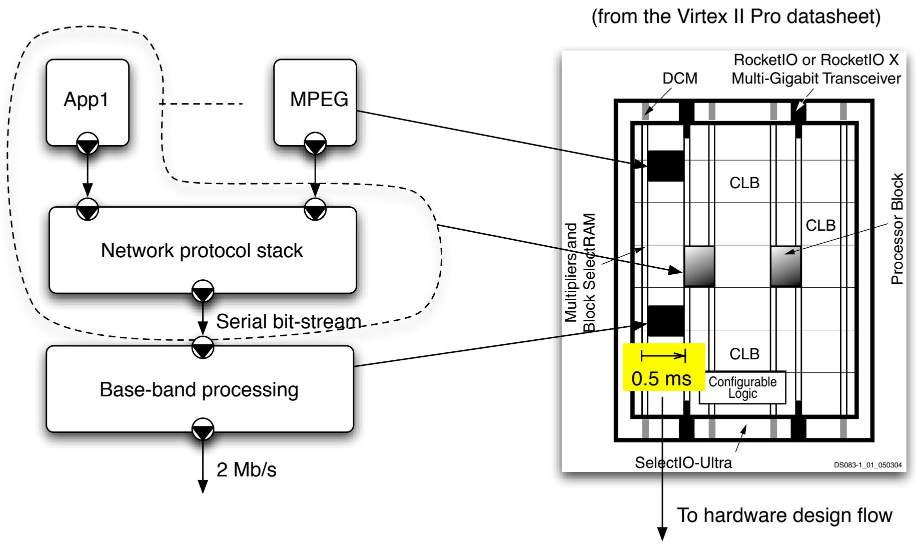


Figure 3.2: Example of an application mapped in software and hardware on a programmable platform. One component is a virtual component that will be synthesized using a dedicated hardware design flow.

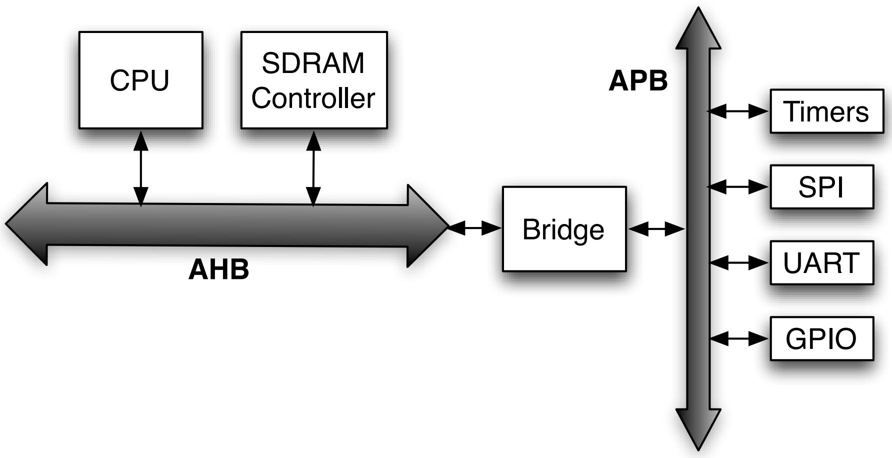


Figure 3.3: Example of a system-on-chip using the AMBA platform.

parameters that are decided when the components are instantiated. Notice that, the selection of these parameters translates into design constraints for the lower abstraction levels. For instance, the clock speed will drive the hardware design and synthesis of the bus arbiter, address decoder, multiplexers and the floor-plan of the SoC.

This concept of platform encapsulates the notion of re-use as a family of solutions that share a set of common features (the elements of the platform). Since we associate the notion of platform to a set of potential solutions to a design problem, we need to capture the process of mapping a functionality (*what* the system is supposed to do) with the platform elements that will be used to build a platform instance or an "architecture" (*how* the system does what is supposed to do). This process is the essential step for refinement and provides a mechanism to proceed towards implementation in a structured way.

We strongly believe that function and architecture should be kept separate as functionality and architectures are often defined independently, by different groups (e.g., video encoding and decoding experts versus hardware/software designers in multi-media applications). Too often we have seen designs being difficult to understand and to debug because the two aspects are intermingled at the design capture stage. If the functional aspects are indistinguishable from the implementation aspects, then it is very difficult to evolve the design over multiple hardware generations.

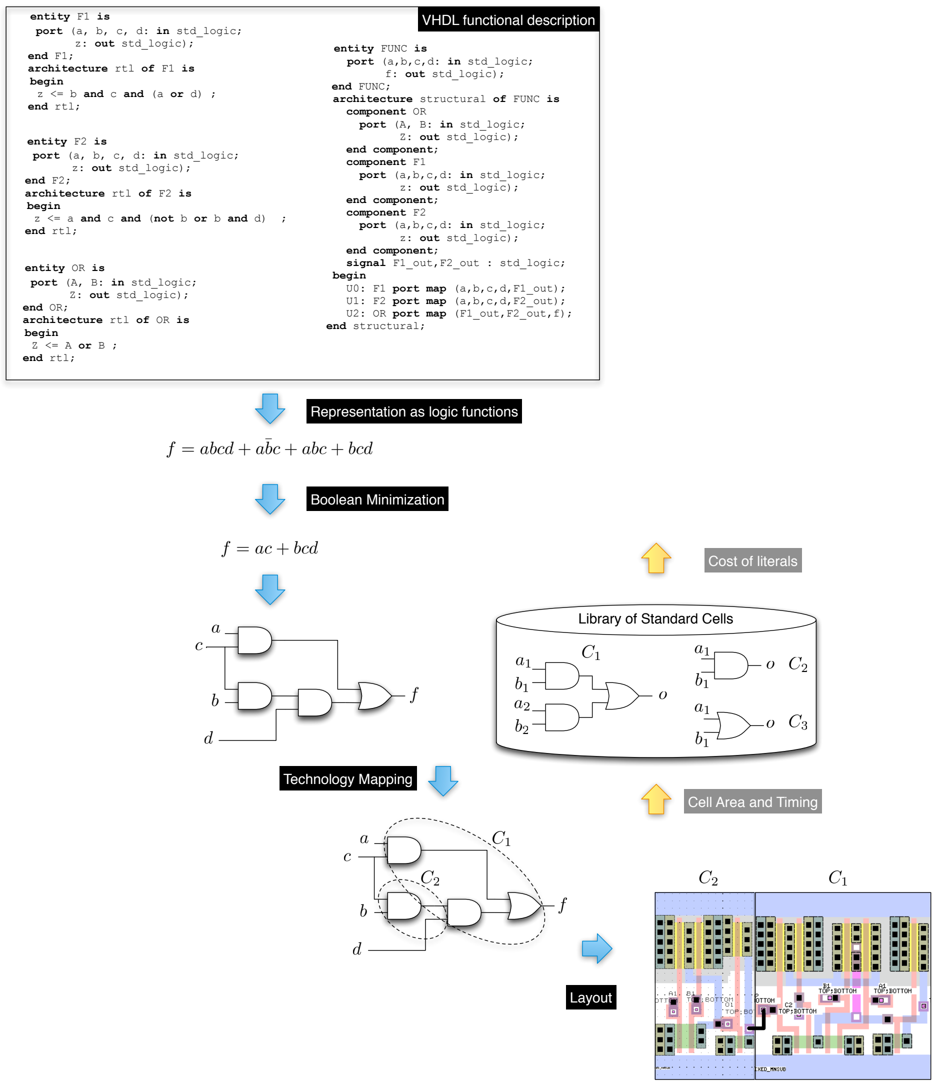
3.3.2 The Design Process

Design is a transformation *process* that takes specifications and turns them into a product. The way in which this process is organized is called a *methodology*. A sound and rigorous methodology has been considered essential in achieving designs with few errors in a relatively short time and better performance. There are designers who consider a methodology as an impediment to creativity as it specifies steps and checks that may slow down the creation of new concepts and distract the designer from the pursue of his/her ideas. However, if the methodology is supported by theoretical results about properties of the intermediate steps in the design process and by tools that carry out some of the transformations automatically and optimally, then instead of being an impediment to creativity, the methodology allows a designer to focus on the concepts rather than on the details.

Traditional Design Processes

In hardware design, the advent of optimized automatic layout generation from schematics followed by the advent of automatic logic synthesis of gates from higher level of abstractions has demonstrably increased design productivity by orders of magnitude allowing the design of circuits of complexity that would not have been possible without these tools and the supported methodology.

In software design, tools of this sort are still in their early stage albeit compilers of high-level languages do share some of the benefits of the design methodology and tools of hardware design. One of the problems of software as it is intended today, is the the expressivity of its mathematical model that being very general does not allow formal analysis and synthesis approaches. In particular, as we shall see later, synchronous reactive languages have rigorous synchronous semantics that allows applying some of the logic synthesis tools to software code generation. However, much work has been done to make the software design process controlled and tightly



```
 entity F1 is
  port (a, b, c, d: in std_logic;
         z: out std_logic);
 end F1;
 architecture rtl of F1 is
 begin
  z <= b and c and (a or d) ;
 end rtl;


entity F2 is
 port (a, b, c, d: in std_logic;
        z: out std_logic);
end F2;
architecture rtl of F2 is
begin
 z <= a and c and (not b or b and d)  ;
end rtl;


entity OR is
 port (A, B: in std_logic;
        Z: out std_logic);
end OR;
architecture rtl of OR is
begin
 Z <= A or B ;
end rtl;
```

```
 entity FUNC is
   port (a,b,c,d: in std_logic;
          f: out std_logic);
 end FUNC;
 architecture structural of FUNC is
   component OR
     port (A, B: in std_logic;
            Z: out std_logic);
   end component;
   component F1
     port (a,b,c,d: in std_logic;
            z: out std_logic);
   end component;
   component F2
     port (a,b,c,d: in std_logic;
            z: out std_logic);
   end component;
   signal F1_out,F2_out : std_logic;
 begin
   U0: F1 port map (a,b,c,d,F1_out);
   U1: F2 port map (a,b,c,d,F2_out);
   U2: OR port map (F1_out,F2_out,f);
end structural;
```

Figure 3.4: An example of hardware design flow from the high level specification to layout.

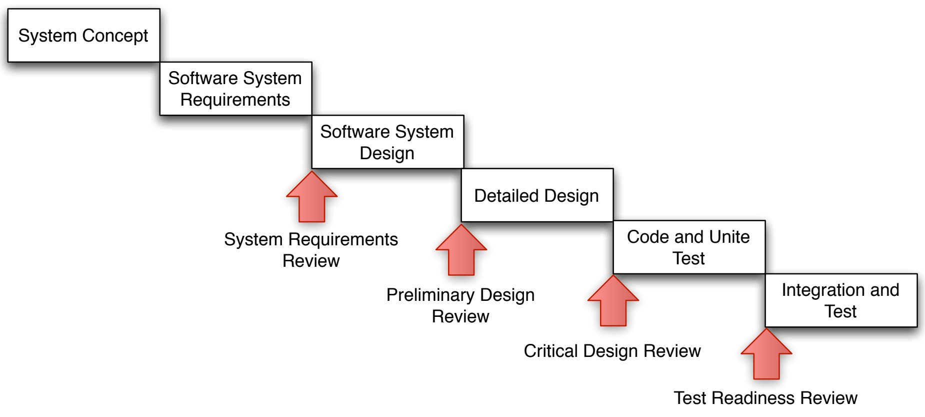


Figure 3.5: The waterfall software development model.

organized in phases with gates that must be passed to continue on. As we mentioned in the introduction, software certification is about the process followed to design the software rather than its contents. Two basic approaches have been followed in the past to organize software design:

- The *waterfall* model shown in Figure 3.5 where the basic steps of the software design process are separated by gates where reviews are carefully organized. Every step and every gate has its own precise set of rules. This model was probably the first approach to a rigorous methodology and was in use in the defense industry where code quality and certification play an essential role and where additional costs are affordable. It has the drawbacks that only at the end one can see the design in its entirety and errors may be discovered too late in the design process in addition to being somewhat rigid and cumbersome (large documents are often generated to legislate the process). In addition, the design team that is responsible for a phase moves on to a new project once its review gate has been passed, so that the personal knowledge of the design may be irreparably lost. This kind of methodology needs streamlining to gain flexibility and speed.

- The *spiral* model shown in Figure 3.6 is a more modern approach where early feedback can be given to the designers on the quality of their design by using rapid prototyping and producing the final design as a “refinement” process of a working but incomplete system. In this case, the initial design team must have a clear understanding of the requirements and of their relative importance as the decision of the refinement steps is crucial to the final quality of the design. In this approach, the design team starts small and remains involved throughout the design process to maintain the knowledge base of the early stage of the design.

The Software Engineering Institute at Carnegie Mellon in Pittsburgh established standards and guidance for developing software engineering disciplines and management known as Capability Maturity Model Integration (CMMI), “a process improvement approach that provides organizations with the essential elements of

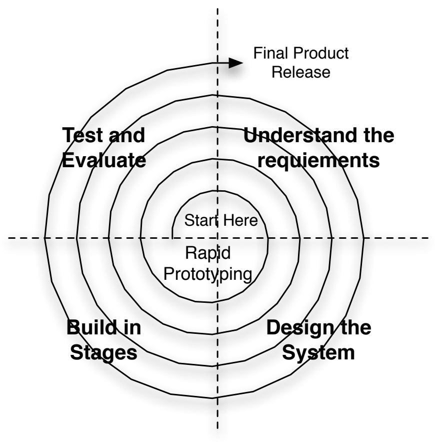


Figure 3.6: Spiral software development model.

effective processes. It can be used to guide process improvement across a project, a division, or an entire organization. CMMI helps integrate traditionally separate organizational functions, set process improvement goals and priorities, provide guidance for quality processes, and provide a point of reference for appraising current processes.” [**?**]. CMMI is widely followed in many companies world-wide that must develop large amount of software in their business. CMMI is not explicitly targeted to the embedded software world.

The waterfall and spiral models are also followed for system design that has significant hardware content. They have a strong *top-down* flavor as the design is conceived as a progression from start to end with no mention of legacy or of pre-designed component assembly. *Bottom-up* design methodologies in the embedded system domain generate their solutions by starting with pre-existing designs that are then adjusted and incremented to respond to a set of new requirements. This process reduces implementation risks since working systems are taken as the initial step but it has severe optimality and requirement satisfaction problems.

The Platform-Based Design Process

The PBD design process is neither a fully top-down nor a fully bottom-up approach in the traditional sense: rather it is a **meet-in-the-middle** process (see Figure 3.7) as it can be seen as the combination of two efforts:

- **top-down**: map an instance of the functionality of the design into an instance of the platform and propagate constraints;
- **bottom-up**: build a platform by choosing the components of the *library* that characterizes it and an associated performance abstraction (e.g., timing of the execution of the instruction set for a processor, power consumed in performing an atomic action, number of literals for technology independent optimization at the logic synthesis level, area and propagation delay for a cell in a standard cell library).

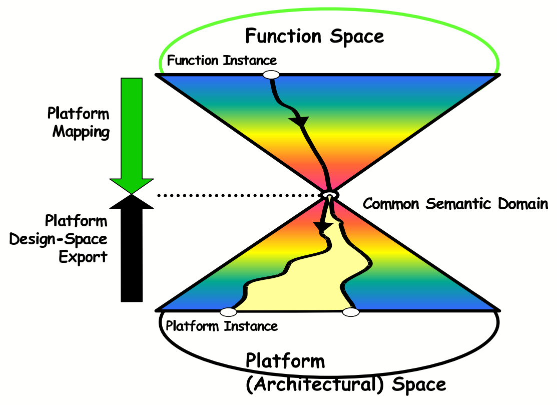


Figure 3.7: Platform-Based Design Triangles.

The “middle” is where functionality meets the platform. Given the original “semantic” difference between the two, the meeting place must be described with a common semantic domain so that the “mapping” of functionality to elements of the platform to yield an implementation can be formalized and automated.

To represent better the refinement process and to stress that platforms may preexist the functionality of the system to be designed, we turn the triangles on the side and represent the “middle” as the mapped functionality. Then the refinement process takes place on the mapped functionality that becomes the “function” at the lower level of the refinement. Another platform is then considered side-by-side with the mapped instance and the process is iterated until all the components are implemented in their final form. This process is applied at all levels of abstraction, thus exposing what we call the “fractal nature of design”. Note that some of the components may have reached their final implementation early in the refinement stage if these elements are fully detailed in the platform.

The resulting Figure 3.8 exemplifies this aspect of the methodology. It is reminiscent of the Y-chart of Gajski **??** albeit it has a different meaning since for us architecture and functionality are peers and architecture is not necessarily derived from functionality but may exist independently[2]. It was used as the basis for the development of Polis [**?**] and of VCC [**?**]. The concept of architecture is well captured by the platform concept presented above.

The result of the mapping process from functionality to architecture can be interpreted as functionality at a lower level of abstraction where a new set of components, interconnects and composition rules are identified. To progress in the design, we have to map the new functionality to the new set of architectural components. In case the previous step used an architectural component that was fully instantiated, then that part of the design is considered concluded and the mapping process involves only the parts that have not been fully specified as yet.

[2]This diagram together with its associated design methodology was presented independently by Bart Kienhuis and colleagues(see e.g., [**?**])

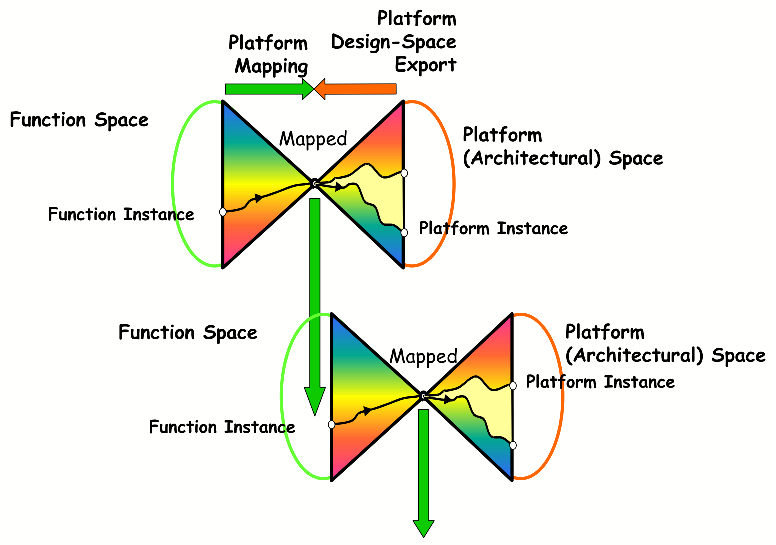


Figure 3.8: The PBD process

While it is rather easy to grasp the notion of a programmable hardware platform, the concept is completely general and should be exploited through the entire design flow to solve the design problem. For example, the functional side of Figure 3.8 can be as high level as a denotational specification (find x such that $f(x) = 0$) and the corresponding platform can be a set of algorithms for operationalizing the specification (e.g., a Newton-Raphson algorithm and a nonlinear successive over relaxation scheme) together with their performance (e.g., quadratic or linear convergence). The choice of a platform instance would be in this case the selection of the algorithm to use together with the constraints that this choice requires (e.g., differentiability of f, non singularity of the Jacobian at the solution for Newton-Raphson). Assuming Newton-Raphson to be the choice, then this platform instance becomes the functional specification for the next layer. In this case, a library of linear equation solvers to be used in the Newton-Raphson algorithm is then the next layer platform. We can continue along this line of reasoning until we decide to use a particular computing platform for the implementation of the fully specified algorithm that is available.

3.4 An Example of Platform-Based Design

In this section we show an example of application of the platform-based design methodology to building automation systems. There are a number of quantities that need to be monitored and controlled inside a building. Temperature, pressure, humidity and pollutants are example of physical quantities that determine the indoor air quality. Depending on the building usage, the indoor air quality must be maintained within precise bounds that are specified by regulations. The building automation system comprises also the intrusion detection system, the control of elevator cars, the secondary power generation system and the fire and security system. Today, these sub-systems are not integrated by there is a common agreement that their cooperation can provide higher comfort, safety and energy efficiency than it

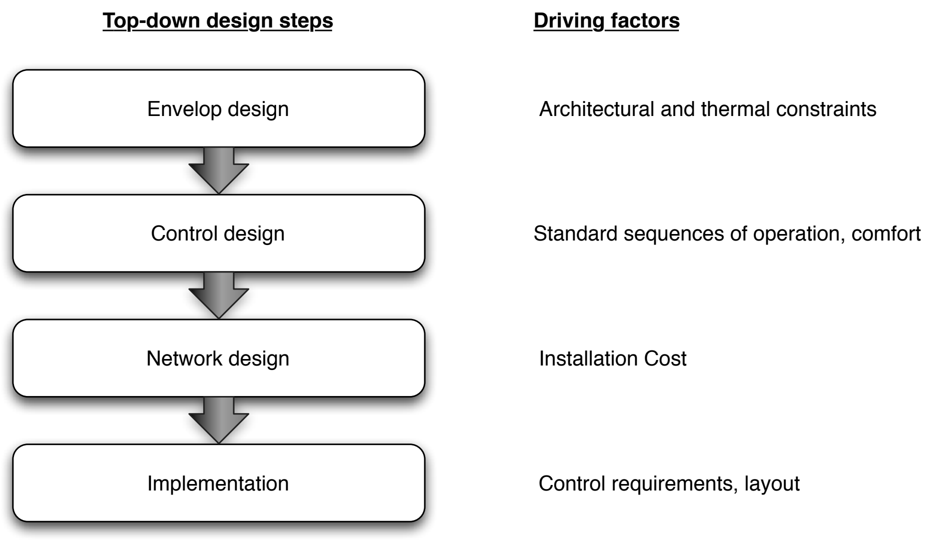


Figure 3.9: Top-down design of building automation systems.

is achieved in current buildings.

The design flow that has been developed over the years is shows in Figure 3.9. First, the building envelope is designed. The geometry and exposure of the building is decided depending on architectural constraints and total thermal loads. Typically, simulation is used to determine the thermal loads. Once the envelope has been designed, the building needs to be instrumented with an automation systems. The control algorithms that are needed to guarantee the desired level of comfort have been developed over the years and standardized. Depending on the type of building (e.g. a commercial or residential building), there are standard sequences of operation that must be implemented. There standard sequences are written in plain English language. They simply define schedules to turn on and off the heat ventilation and air conditioning (HVAC) system, alarm conditions, temperature ranges and even the way in which this variables must be controlled.

The standard sequence of operation is considered the specification of the controls of the building automation system. The next step is indicated as network design in Figure 3.9. In practice, there are companies that provide hardware and software platform solutions that can readily adopted and installed throughout the building to interconnect sensors and actuators to computation units (i.e. controllers) and to the main server. Finally, the hardware platform and the controls specification is taken by control contractors that need to implement the standard sequences on the given platform.

This design flow is purely top-down in the send that the specification is passed to the next abstraction level where it is refined and passed to the subsequent one. For instance, the building envelope is designed without any knowledge of the performance of the control algorithms, and the standard sequences are defined without any knowledge of the underlying implementation platform.

Figure 3.10 shows the same design flow where we added the performance abstrac-

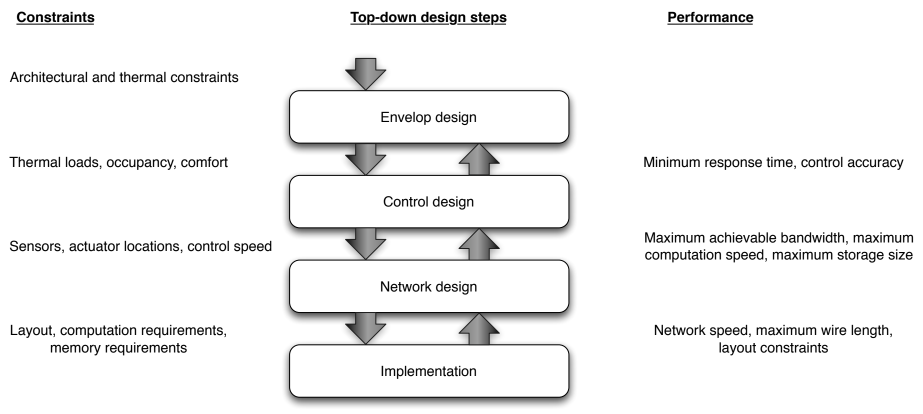

Constraints
Top-down design steps
Performance
Architectural and thermal constraints
Envelop design
Thermal loads, occupancy, comfort
Minimum response time, control accuracy
Control design
Sensors, actuator locations, control speed
Maximum achievable bandwidth, maximum computation speed, maximum storage size
Network design
Layout, computation requirements, memory requirements
Network speed, maximum wire length, layout constraints
Implementation

Figure 3.10: Platform-based design of building automation systems.

tion that flows in the opposite direction of the constraints. In this view, the design of the envelope is done by matching the architectural and thermal constraints with a notion of what the HVAC system and the control algorithm can achieve in terms of performance. Similarly, the control algorithms are defined taking into account the comfort constraints but also an abstraction of the network and hardware resources available at the lower level of abstraction. The result of this design flow is that the final implementation is potentially more cost effective since the entire system is obtained by matching the requirements with the available resources. Moreover, taking into account the restrictions imposed by the available hardware early in the design process avoids design error that are typically discovered in the final implementation like network bottleneck.

A more detailed description of the platform-based design applied to distributed control system is shown in Figure 3.11. The design flow consists of the following steps:

Step 1: *Decentralizing Control*. The control algorithm is initially described in a centralized manner assuming an ideal implementation, i.e. infinite computation and communication resources. This separation, which is consistent with the basic tenets of PBD, is essential to analyze the functionality of the control strategy independently from resource sharing, so that that any misbehavior detected in later stages of the design can be isolated.

Centralized control, especially for complex distributed systems, may not be the optimal choice and, in fact, may not even be implementable. A design option to explore in this step is to distribute the control over a set of smaller cooperating controllers to reduce the size of the state space and consequently the computational requirements. Given a centralized description, there are many techniques that can be used to derive an equivalent distributed algorithm. Among all the distributed controllers, one has to be selected in an optimized way.

To do this, the problem is formulated as mapping of the centralized controller onto a platform instance that combines local controllers sharing some state variables. The local controllers form a library of components that must

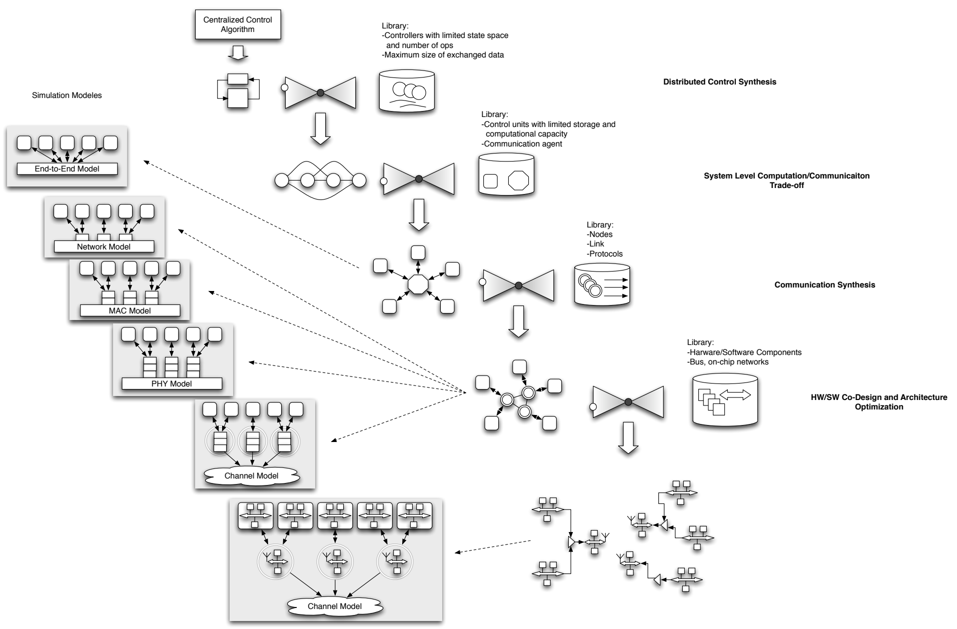

Centralized Control Algorithm
Library:
-Controllers with limited state space and number of ops
-Maximum size of exchanged data
Distributed Control Synthesis
Simulation Modeles
Library:
-Control units with limited storage and computational capacity
-Communication agent
System Level Computation/Communicaiton Trade-off
End-to-End Model
Library:
-Nodes
-Link
-Protocols
Network Model
MAC Model
Communication Synthesis
Library:
-Harware/Software Components
-Bus, on-chip networks
PHY Model
HW/SW Co-Design and Architecture Optimization
Channel Model
Channel Model

Figure 3.11: Platform-based design of distributed control systems.

be characterized by performance and cost indexes. For instance, each local controller may be attributed with a maximum size for its state variables (capturing memory constraints) and a maximum number of multiplications (capturing computational limitations). Also, each shared variable may be limited in size, implicitly capturing a limitation on the communication capabilities at lower levels of abstraction. The mapping process can be cast an optimization problem whose solution gives the best control architecture that performs the given function originally captured as centralized control.

Step2: *Communication/computation trade-off.* The distributed control algorithm is characterized at this level by constraints on the eigenvalues to guarantee convergence, and by an interacting set of local controllers obtained as the solution to the mapping problem of Step 1. Given this constraints, the computation/communication design space exploration is about finding an allocation of computational resources (where to compute the control laws) and communication requirements (the amount of data to transfer between controller and the end-to-end quality of services) such that the overall cost of the system is minimized.

The library of components here contains computation and communication resources. The communication resource is a single monolithic agent that abstracts any network by providing a cost metric that depends on the number of end-to-end connections and their quality of service requirements. The design space exploration consists of balancing the computation and communication cost. In fact, partitioning the computation could be convenient when small and simple units are less costly than a central server. At the same time, breaking down the computation introduces inter-module communication that may require an expensive high-performance network.

Step 3: *Communication Design.* The end-to-end communication requirements and the constraints on the position of computational units, sensors, and actuators obtained by Step 2 are given as inputs (function) to the communication design step. The components of the platform library that are available for implementing the communication network are routers, gateways, protocols and links. First the interconnect topology and the per-link quality of service must be chosen. Given the required routing of messages and the per-link performance requirements, the parameters of the MAC layer and of the physical layer must then be optimally computed in this Step.

Step 4: *HW/SW co-design and architecture optimization.* Once the communication network has been designed, the next step of the design flow is Hardware/Software Co-Design. For each computational resource and communication node, an optimal architecture made of processors and custom hardware must be selected and code must be generated. Here the platform library components are the available hardware modules and microprocessors decorated with their Operating Systems and compilers.

This is just one example of design flow for these types of systems. Other flow may also be appropriate depending on the specific application.

3.4.1 Considerations on the Use of PBD

In PBD, the partitioning of the design into hardware and software is not the essence of system design as many think, rather it is a consequence of decisions taken at a higher level of abstraction. Critical decisions are about the architecture of the system, e.g., processors, buses, hardware accelerators, and memories, that will carry on the computation and communication tasks associated with the overall specification of the design.

In the PBD refinement-based design process, platforms should be defined to eliminate large loop iterations for affordable designs: they should restrict the design space via new forms of regularity and structure that surrender some design potential for lower cost and first-pass success. The library of functional and communication components is the design space that we are allowed to explore at the appropriate level of abstraction.

Establishing the number, location, and components of intermediate "platforms" is the essence of PBD. In fact, designs with different requirements and specifications may use different intermediate platforms, hence different layers of regularity and design-space constraints. The trade-offs involved in the selection of the number and characteristics of platforms relate to the size of the design space to be explored and the accuracy of the estimation of the characteristics of the solution adopted. Naturally, the larger the step across platforms, the more difficult is predicting performance, optimizing at the higher levels of abstraction, and providing a tight lower bound. In fact, the design space for this approach may actually be smaller than the one obtained with smaller steps because it becomes harder to explore meaningful design alternatives and the restriction on search impedes complete design-space exploration. Ultimately, predictions/abstractions may be so inaccurate that design optimizations are misguided and the lower bounds are incorrect.

The identification of precisely defined layers where the mapping processes take place is an important design decision and should be agreed upon at the top design *management level.* Each layer supports a design stage where the performance indices that characterize the architectural components provide an opaque abstraction of lower layers that allows accurate performance estimations used to guide the mapping process.

This approach results in better re-use*, because it decouples independent aspects, that would otherwise be tied, e.g., a given functional specification to low-level implementation details, or to a specific communication paradigm, or to a scheduling algorithm.* It is very important to define only as many aspects as needed at every level of abstraction, in the interest of flexibility and rapid design-space exploration.

3.5 Concluding Remarks on Platform-Based Design

The notion of PBD presented in this section is being adopted rather widely by the EDA companies who are active in the system space or that are eyeing that market. CoWare [**?**] and Mentor Graphics [**?**] use platforms in their architectural design and design-space exploration tools pretty much in the sense we introduced here. Cadence and National Instruments use the concepts of platforms in the description of their tools and approaches using diagrams similar to Figure 3.7.

We believe PBD serves well the purpose of the supply chain as the layers of abstraction represented by the platforms can be used to define the hand-off points of complex designs. In addition, the performance and cost characteristics associated to the platforms represent a "contract" between two players of the design chain. If the platform has been fully specified with performance and cost given by the supplier, then the client can design at his/her level of abstraction with the assumption that the "contract" will be satisfied [**?**, **?**]. If the supplier has done well his/her homework, the design cycles are considerably shortened. If one or more of the components of the platform instance chosen by the client is not made available by the supplier, but it has to be designed anew, the performance assumed by the client can serve as a specification for the supplier. In both cases, the "contract" is expressed in executable form and prevents misunderstandings and long design cycles.

The platform concept is also ideal to raise the level of abstraction since it does not distinguish between hardware and software but between functionality and architecture. Hence, the design-space exploration can take place with a more degrees of freedom than in the traditional flows. In addition, the partitioning between hardware and software components can be done in an intelligent and optimized way.

On the other hand, PBD does require a specific training of designers to guide them in the definition of the "right" levels of abstraction and of the relationships among them. It does benefit from the presence of supporting tools for analysis, simulation and synthesis organized in a well-structured design flow that reflects the relationships among the platforms at the different layers of abstraction. Designers have to be careful in extracting implementation aspects they want to analyze from behavior of their design. In my experience of interaction with industry on importing PBD, this has possibly been the most difficult step to implement. However, once it is well understood, it gave strong benefits not only in terms of design time and quality, but also in terms of documentation of the design.

As we mentioned several times, the methodology, framework and tools presented above can serve as an integration framework to leverage the many years of important work of several researchers. As done in [**?**], we use the diagram of Figure 3.12, a simplification of Figure 3.8, to place in context system-level design approaches. This classification is not only for taxonomy purposes. It also shows how to combine existing approaches into the unified view offered by PBD to build optimized flows that can be customized for particular applications.

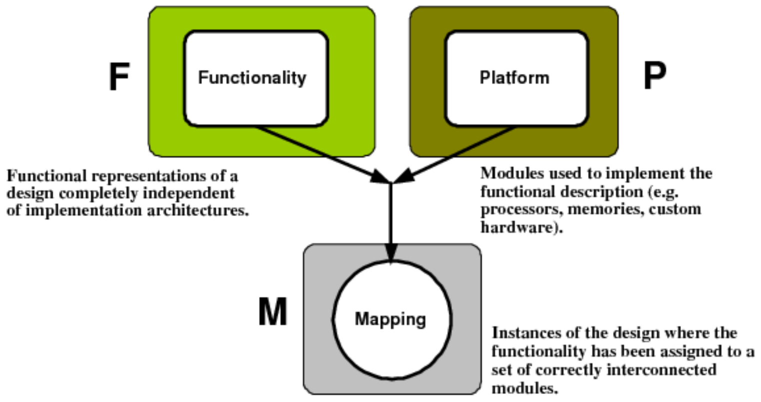

F
Functionality
Platform
P
Functional representations of a design completely independent of implementation architectures.
Modules used to implement the functional description (e.g. processors, memories, custom hardware).
M
Mapping
Instances of the design where the functionality has been assigned to a set of correctly interconnected modules.

Figure 3.12: Function-Architecture-Mapping.

3.6 To Probe Further: Model-Driven Software Development

The paradigm that most closely resembles PBD is Model-Driven (Software) Development (MDD). MDD is subject of intense research and investigation in the software development community as it bears much promise to improve the quality of software. For an excellent review of the state of the art and of challenges that MDD poses to the software community, we recommend the March 2006 issue of the IBM Systems Journal [?] and in particular, the paper "Model-Driven Development: the good, the bad and the ugly" by B. Hailpern and P. Tarr, for a deep analysis of the pros and cons of the approach.

MDD is based on the concept of Model-Driven Architecture. The OMG defines the term Model-Driven Architecture(MDA) to be as follows: "MDA is based on a Platform-Independent Model (PIM) of the application or specifications business functionality and behavior. A complete MDA specification consists of a definitive platform-independent base model, plus one or more Platform-Specific Models (PSMs) and sets of interface definitions, each describing how the base model is implemented on a different middleware platform. A complete MDA application consists of a definitive PIM, plus one or more PSMs and complete implementations, one on each platform that the application developer decides to support. MDA begins with a model concerned with the (business-level) functionality of the system, independent of the underlying technologies (processors, protocols, etc.). MDA tools then support the mapping of the PIM to the PSMs as new technologies become available or implementation decisions change." [?]

The concept of separation of concerns between function and platform is clearly stated. The use of the mapping of functionality to platforms as a mean to move towards the final implementation is also expressed. However, the similarities between the two approaches ends here as the definition of platform is not fully described nor the semantics to be used for embedded software design.

The University of Vanderbilt group [?] has evolved an embedded software design methodology and a set of tools based on MDD. In their approach, models explicitly represent the embedded software and the environment it operates in, and capture the requirements and the design of the application, simultaneously. Models are descriptive, in the sense that they allow the formal analysis, verification, and validation of the embedded system at design time. Models are also generative, in the sense that they carry enough information for automatically generating embedded software using the techniques of program generators. Because of the widely varying nature of embedded systems, the Vanderbilt researchers emphasize that *a single modeling language may not be suitable for all domains*; thus, modeling languages should be domain-specific (DSL). These languages have significant impact on the design process [?] for complex software systems. In embedded systems, where computation and communication interact with the physical world, DSLs offer an effective way to structure information about the system to be designed along the "natural dimensions" of the application [?] . *We take the position that DSLs for embedded systems should have a mathematically manipulable representation.*

This view goes against the use of a general language for embedded systems and favors customization to obtain better optimization and easier adoption. However, customization carries obvious drawbacks in terms of development costs and support efforts. To decrease the cost of defining and integrating domain-specific modeling languages and corresponding analysis and synthesis tools, the Model-Integrated Computing (MIC) [?] approach is applied in an architecture, where formal models of domain-specific modeling languages, called metamodels play a key role in customizing and connecting components of tool chains. The Generic Modeling Environment (GME) [?] provides a framework for model transformations enabling easy exchange of models between tools and offers sophisticated ways to support syntactic (but not semantic) heterogeneity. The KerMeta metamodeling workbench [?, ?] is similar in scope.

In synthesis, MDD emphasizes design by (whenever possible automatic) model transformations. Model-based approaches have been applied for years in the hardware domain where one can argue that since the introduction of logic synthesis, this approach has had great success. Most of the formal approaches to hardware design are indeed model driven in the sense that a design model is successively transformed into hardware. In embedded software, the approach has still to be fully exploited as using a model-driven method requires the description of the software with mathematical models, a step that for most software designers is not easy. Domain-specific languages will probably help in pushing for the adoption of MDD in the embedded software community since it is possible to design these languages to meet the specific needs of a homogeneous group of designers thus allowing them to be more effective in expressing their designs. However, if indeed each design group is going to have its specific language, the problem will be how to interface the various parts of the design so that the composition can be analyzed and verified. We believe that this issue can be resolved only if the semantics of the languages are well understood and the interaction among parts described with different languages is mathematically well characterized. The Vanderbilt group is addressing some of these issues with semantic anchoring of DSLs using abstract semantics based on Abstract State Machines [?, ?]. In addition, the MILAN framework [?] offers a number of simulation, analysis and synthesis tools that leverage the MIC framework. A recent approach to "gluing" parts described by different languages consists of using higher-level programming models and languages for distributed programming, called *coordination* models and languages [?, ?]. In the coordination model approach, one can build a complete programming model out of two separate pieces-the *computation model and the coordination model.* The computation model allows programmers to build a single computational activity: a single-threaded, step-at-a-time computation. The coordination model is the glue that binds separate activities into an ensemble. The similarity with the separation between computation and communication in PBD is strong.

A coordination language is "the linguistic embodiment of a coordination model" [**?**]. The most famous example of a coordination model is the Tuple Space in *Linda*, a language introduced in the mid-1980s, that was the first commercial product to implement a virtual shared memory (VSM), now popularly known as tuples-pace technology for supercomputers and large workstation clusters. It is used at hundreds of sites worldwide [**?**]. Linda can be seen as a sort of assembly level coordination language since it offers:

- very simple coordination entities, namely, active and passive tuples, which represent processes and messages, respectively;
- a unique coordination medium, the Tuple Space, in which all tuples reside;
- a small number of coordination laws embedded in four primitives only.

Coordination languages can be built on Linda to offer higher-level of abstraction construct to simplify the synchronization and message passing among the components. Many coordination languages have been built over the years. An excellent review of Linda derivatives and coordination languages such as *Laura* and *Shade* can be found in [**?**]).

Once more, we advocate the add a strong mathematically-sound semantics to the linguistic approach to composition. This is indeed the contribution of some of the environments for heterogeneous models of computation such as Ptolemy II and Metropolis.

Chapter 4

Capturing the Design: Requirements and Specifications

4.1 Introduction

Any system design and development activity should always begin with a phase where requirements and technical specifications are collected and captured. Requirements engineering is a key problem area in the development of complex, software-intensive Systems. As stated in [**?**]: *"'The hardest single part of building a software system is deciding what to build. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."*

System requirements should clarify what the system is supposed to do, what properties and constraints it should satisfy (what is its behavior) and also environmental and regulatory conditions. The product of this process is a model, from which a document is produced. As such, requirements may be the technical annex to a contract specifying what system part needs to be provided by a supplier in an integrator-supplier relationship, or they may be part of the internal company documentation. In the latter case, they describe the (sub)system acting as a reference point shared between those who investigate the customer's needs, those who implement systems and programs to satisfy those needs, those who test the results, those who write instruction manuals and providing a reference for maintenance and extensions.

Traditionally, requirements have been given informally with many pages of natural language descriptions, sometimes flanked with manually derived diagrams that represent the structure of the design. In the documents, a lengthy elaboration of what is expected of the design is accompanied by a set of properties the design has to satisfy and a set of requirements on the implementation, possibly including indications on the components to be used. For example, in the automotive domain, when a subsystem is specified by a car maker for a Tier 1 supplier, e.g., an engine controller, the characteristics of the subsystems included the prescription of what microprocessor platform to use, e.g., the part number of an IC supplier.

We argue that this approach amounts to far-from-optimal designs since the choice of the implementation platform should be best decided by the Tier 1 supplier who can optimize with respect to his own business criteria while delivering a product that satisfies its function definition and the performance constraints. In addition, requirements and specifications often change during the lifetime of the design project and keeping track of these variation is a constant nightmare. Not unexpectedly, the informality of the description also leads to misunderstandings, delays and cost overrun.

The issues of specification capture and management is considered today as a crucial aspect of system design and one that needs much research and tool development. There is evidence that a ratio of 15 to 1 in design quality and efficiency can be credited to a good specification entry and analysis process. The DoD Software Technology Plan [**?**] states that "early defect fixes are typically two orders of magnitude cheaper than late defect fixes, and the early requirements and design defects typically leave more serious operational consequences." In this chapter, we briefly outline the main issues and approaches to the problem and then present some general principles that are typical of the embedded system world.

4.2 Requirements Engineering

Software engineering is a discipline born out of the need to streamline and formalize wherever possible the software design process. In this domain, two approaches have had significant impact on the way software has been specified (and designed): *Requirements Engineering* (RE) and *Object Oriented Analysis* (OOA). The two came remarkably together on several principles, but were independently derived by separate communities. These efforts have not been widely used in the embedded system design community due to a cultural and a practical mismatch between the languages and the needs of the two communities, albeit recently there have been great efforts in deriving specialized extensions (or *profiles*) for system design out of object oriented formalisms such as UML (see Chapter **??**).

RE is indeed most relevant to our discussion in this chapter. It can be decomposed into the activities of requirements elicitation, specification, and validation.

- ***Requirements elicitation*** is the stage in which, by several means, including interviews, focused groups, product analysis, the requirements are agreed among the stakeholders. Success of the elicitation stage depends on the *use of a language that promotes communication and understanding between (among others) developers, marketing analysts, application domain experts, customers and users.*

- ***Requirements specification*** is the stage in which requirements and the information in them are examined, assimilated and then represented, either formally or informally, in a document and/or a set of diagrams or any other means.

- ***Requirements validation*** amounts to checking that requirements allow for a feasible solution (i.e. are not inconsistent and allow for an implementation) and subsume the properties of interest for the system. The specifications document should be validated and verified to ensure that it is complete, consistent and correct.

The process is seldom a simple cascade among these three steps. Quite often, designers proceed iteratively and, at the end of each iteration, test whether enough information has been gathered and all requirements identified. The process almost invariably starts as informal in the elicitation stage, but it can be formalized in the specification step assuming availability of languages, methods and tools for describing the identified requirements. Indeed, the majority of research work focuses on the validation stage, assuming availability of requirements written in some formal language.

While this is seldom the case in practice, a formal language with a well-defined semantics provides several advantages, including the possibility of automatic parsing and removal of ambiguity. Further, formal languages offer a basis for various ways of reasoning with models, either through consistency checking or simulation/prototyping. Research focused on the verification problem, that is, to determine whether the expressed requirements are implementable, typically using formal methods based on model checking, theorem proving or other automatic logical inference. In addition, a formal specification language can be executable and allow for early prototyping and simulation of behavior or property verification.

RML [**?**] was one of the first attempt at providing a formal foundation to requirement engineering, together with a number of other formal languages, including (among others) Z [**?**] and VDM [**?**], but most models of computation, as outlined in the following chapters, including Finite State Machines (or FSMs), Petri Nets and their timed variants, like the Timed Automata, Timed Petri Nets or even Hybrid Automata can be used to formally express the desired behavior of the system. First order- or timed-logic predicates can be used to express desired properties and constraints.

On the other hand, the appeal and usability of some techniques may be largely due to their relative simplicity and flexibility derived from informality. We note that the use of a formal requirements modeling language does not preclude the concurrent use of informal notations. In fact, the original RML proposal envisioned early use of an informal notation and a transformation process from the informal model into a formal RML one.

While requirement engineering is indeed a most interesting approach to software design, we are still far from using these concepts in system design. There are efforts afoot in the UML community to improve the specification process by offering a system description language that include requirement capture features (SysML described in Chapter **??**).

4.3 System Specification Principles

The PBD principles described in Chapter 3 imply that functionality and architecture may be considered as independent inputs to the design process, albeit any experienced designer would consider the reciprocal influence of the two arms of the Y chart when capturing his/her design. Indeed, when dealing with a system-level design problem, we must consider not only *what* the system is supposed to do but also what are the *restrictions* placed on the space of feasible solutions either by constraints posed by the application, by the need to cope with complexity, i.e., the "size" of the design space to be explored, or by constraints posed on cost, power, reliability and availability of the architectural components to be considered for implementation. Thus, a specification, $\mathcal{S}$, consists of a description of the functionality the system, $\mathcal{F}$, together with a set of constraints, $\mathcal{C}$, that is partitioned into constraints on the behavior of the system, $\mathcal{C}_{\mathcal{F}}$, and on the characteristics of the implementation, $\mathcal{C}_{\mathcal{I}}$:

$$\mathcal{S} \;=\; \{\mathcal{F},\mathcal{C}\},\; \mathcal{C} \;=\; \{\mathcal{C}_{\mathcal{F}},\mathcal{C}_{\mathcal{I}}\}. \tag{4.1}$$

For example, if we are designing a communication protocol, then we may have constraints on the functionality (e.g., absence of deadlock) and on the implementation (e.g., a bound on the power consumed that the implemented system should not exceed).

The question is now how to capture the initial specification of the system. We argue that the *functionality* of the design should be captured at highest level of abstraction, i.e., at the level we "enter" the design, in an *unambiguous* way: given an input, the behavior of the system should have a unique response, i.e., it should be *deterministic* (or *determinate*). If the functional description is non deterministic, then we argue that either the specification is incomplete, i.e., the designer has forgotten to specify the behavior of the system for a particular input or that the designer is not interested in the output of the system for that particular value of the input. For example, assume we consider a sequential system whose input set has three elements. If we implement that system in digital forms then we have to encode each symbol corresponding to the input set a Boolean representation. Given three objects, we need at least two bits to encode them. However, since there are four distinct elements in the Boolean space corresponding to two bits (e.g., $\{0,0\},\{1,0\},\{0,1\},\{1,1\}$) there will be one combination that does not correspond to any input. The implemented digital system will have a behavior corresponding to the "meaningless" bit combination but the output value does not matter to the designer since, in the absence of faulty behaviors, that combination will never occur.

The constraints on the behavior of the system $\mathcal{C}_{\mathcal{F}}$ should be formalized as a set of input-output relationships. In this case then, the system is said to satisfy the requirements if all its behaviors are contained in the set of constraints. The requirements on the implemented system $\mathcal{C}_{\mathcal{I}}$ can also be called *reaction requirements* and are expressed in terms of "physical" quantities such as power or timing. These may be expressed in terms of equalities or inequalities involving the variables of the design and characteristics of the implementation. A common reaction requirement is response time, which bounds the worst-case or average-case delay between an external stimulus (input) of the system and its response (output). These requirements guide the selection of the architecture (platform instance) so that when we substitute for the free parameters of the requirements the actual values of the corresponding parameters of the architecture, we can verify whether the selected implementation is a feasible one.

We consider constraints (sometimes called *execution requirements*) such as limits on available machine cycles, memory space, battery capacity, and channel bandwidth as characteristics of the elements of the implementation platform, consequently, they will be discussed in Chapter **??**.

We argued for deterministic behavior in the specification, yet we showed that nondeterminism has some important aspects in system design that we want to keep in our framework. Nondeterminism in system design refers to two different phenomena.

One that we discussed briefly above is about *don't-care* values in the output behavior of a system. Let u and y be vectors of input and output variables respectively in an appropriate domain. In most cases, u and y take values in the space of real numbers, integer or Boolean but it is not infrequent to think of u and y as members of a functional space, for example when we deal with continuous time systems where the inputs and outputs are related by ordinary differential equations. Then the behavior of the system can be given as an explicit function, i.e., $\mathcal{F} : y = f(u)$ or as an implicit function $\mathcal{F} : g(x,u) = 0$. Given an input u a deterministic system has a unique y that satisfies the behavior equations. However, if for a given u, the corresponding y is not specified uniquely, then we argue that we should augment the output space with a special element called the don't care value so that when the input variable is in the don't care set, the output takes that special value. In this augmented space then, the behavior is indeed unique. This approach has been followed for years in the domain of logic design and has allowed the development of powerful optimization techniques that can take advantage of the knowledge of the "don't care" set. In fact, a don't care does not mean a multiplicity of correct responses of the system with respect to a single input; it simply says that for a particular set of inputs, the response can be selected at will, and we can use this freedom to optimize the implementation of the design. In fact, after an implementation is chosen, the intended behavior of the system is indeed deterministic; given an input, the output is uniquely defined and satisfies all the equations describing the behavior of the system except for the set of don't cares where it always satisfies the equations.

The other form is about the environment of the system, which is in general free to behave in many different ways and we cannot predict which behavior is going to exhibit at any given time. This nondeterminism is associated with the specification of the environment as seen by the system under design and so it is not a property of the design. Therefore, when we refer to a reactive system as deterministic in this context, what we mean is not that there is a unique stream of input and output values, but that for every stream of input values that is provided by the environment, the stream of output values that is computed by the system is unique.

For embedded systems, it is often the case that the input and output streams include "time stamps". More precisely, a timed input stream is a sequence of time-stamped input values, such as sensor readings or user commands; and a timed output stream is a sequence of time-stamped output values, such as actuator updates and other generated events that are observable by the environment. We say that an embedded system is *time-deterministic* if for every timed input stream, the timed output stream that is computed by the system is unique [**?**, **?**]. Note that time-determinism refers not only to input and output values, but also to the times at which input values are given to the system and the times at which output values are made available to the environment. If an embedded system computes a unique output value, but may make the value available to the environment (say, by updating an actuator) at different time instants, then the system is not time-deterministic. Obviously, time determinism is essential for safety-critical real-time systems such as those deployed to control automobiles or aircraft.

4.4 From Informal to Formal Requirements

The general framework outlined with the previous principles needs to be instantiated by an actual language that defines the system functionality $\mathcal{F}$ and the constraints acting on it.

In the early stage of requirements elicitation, informal, natural language (plain English) descriptions are often used. They are necessary to communicate with people with a non-engineering background such as those providing input to this stage and can be effective if properly structured and organized. As already stated, natural language is subject to misinterpretation due to ambiguity, inconsistencies, omissions and redundancy and it is therefore desirable that requirements are refined or translated, until they eventually find an equivalent form in a formal language.

Several proposals exist for how to conduct this translation and also on the formal requirements language that is the final destination of this activity. An example is the methodology developed by Nancy Leveson at MIT and based on Intent specifications [Leveson00a] and the SpecTRM language [Leveson00b]. An approach to requirements collection and analysis requires a methodology, tools, requirements documents formats and requirements writing rules. A possible transition from elicitation to specification consists of the following:

- Elicit requirements in "structured" natural language, enforcing a writing style that identifies states or working modes, assumptions and assertions using a contract-based approach or, alternatively an explicit identification of pre-requisites, post-conditions and invariants to lower the probability of ambiguities and inconsistencies. This structure eases the transition from English language towards an FSM description.
- Enforce the definition of tests associated to each requirement item
- Enforce the early use of a data dictionary to minimize the chance of inconsistencies in the definition of the system names and system variables, including: I/O variable names, system parameters, and system states.
- Use of simple rules to avoid redundancy/inconsistency in the definition of the system reactions.
- The use of semi-formal of formal diagrams (context diagrams, sequence diagrams and state diagrams) to complement/refine the text description.

At all levels, contract-based definitions should be used. According to design-by-contract [Meyerxx], system (subsystem) functioning may be considered as a contract between the system (subsystem) and the environment (other subsystems). In the definition based on Assumptions/Assertions pairs, the contract consists of a set of "assumptions" on the environment or the other subsystems (what the "users" of the subsystem promise to be or to behave). If the environment and/or the subsystems satisfy the assumptions, the system (subsystem) under specification will have the duty to keep its side of the contract, that is, a set of assertions (what it promises to provide/how it promises to behave) Assumptions and assertions can be specified formally or informally using several languages.

Starting from the set of descriptions in the previously listed sections, a refinement in more formal languages and diagrams will have to be produced resulting in a set of semi-formal or even formal diagrams including.

- State Chart Diagrams - depict the required behavior of the feature with the states, triggers, conditions and transitions (base for the definition del system behaviour for refinement)
- Functional Context Diagrams - listing all of the external I/O signals in & out of this feature. The input signals on the left side of the drawing and the output signals on the right side of the drawing. La descrizione deve essere puramente funzionale!. (base per la definizione dell'architettura di sistema e la scomposizione in sottosistemi)
- Sequence diagrams, scenarios - list the sequence of actions/events identified for several typical working cases of the system (base per la definition of the test cases)

Possible formal languages for expressing requirements including finite state machines (FSMs), LTL or CTL (for constraints description) the Z language, and many others.

4.4.1 Z

Z (pronounced Zed), is a specification language that works at a a high level of abstraction to describe formally complex behaviors. It is based on set theory and first-order predicate logic and was originally developed at the Oxford University Computing Laboratory (OUCL) in the late 70s, and used in non-trivial "real world" projects. Z is now defined by an ISO standard and is public domain.

In Z, a specification is decomposed in sections called *schemas*. Each schema can be linked with a commentary which explains informally the significance of the formal mathematics. Synctactically, a Z schema is represented as a named box partitioned into two parts, variable declarations and optional predicates relating the variables. Schemas are used to describe both static and dynamic aspects of a system. The static aspects include:

- the states it can occupy;
- the invariant relationships that are maintained as the system moves from state to state.

The dynamic aspects include:

- the operations that are possible;
- the relationship between their inputs and outputs;
- the changes of state that happen.

A mathematical framework describes both the state space of the system and the operations that can be performed on it. The data objects in the system are described in terms of mathematical data types such as sets and functions. The description of the state space included an invariant relationship between the parts of the state.

The notation of predicate logic is used to describe abstractly the effect of each operation.The effects of the operations are described in terms of the relationship which must hold between the input and the output, abstracting from implementation details.

As an example of Z specifications, Andy [Andy1994] shows an elevator operation model together with proofs on liveness and safety properties. It is particularly suited to non-constructive requirements specifications. Semantics-preserving refinement techniques allow formal translation of Z specifications to executable code (e.g., King, 1990), and formal proof techniques are described in [Diller1990].

4.4.2 UML/SysML with OCL

The Object Constraint Language (OCL), developed by IBM, is a declarative language for describing rules that apply to UML models, initially as a formal specification language extension to UML. It is a precise text language that provides constraint and object query expressions on any MOF model or meta-model that cannot be expressed by diagrammatic notation without OCL. Unified Modeling

Language (UML) is a standardized general-purpose modeling language for software engineering. UML includes a set of graphical notation techniques to create abstract models of specific systems. UML has succeeded the concepts of the Booch method, the Object-modeling technique (OMT) and Object-oriented software engineering (OOSE) and combine them under one framework. It can model concurrent and distributed systems, and targets to be a standard in software engineering.

The SysML (Systems Modeling Language) is a is defined as a dialect (Profile) of UML 2. It supports the specification, analysis, design, verification and validation of a broad range of systems and systems-of-systems. The reason why SysML was developed is that systems engineers are seeking a domain-specific modeling language to specify complex systems that include non-software components (e.g., hardware, information, processes, personnel, and facilities). UML cannot satisfy this need because of its software bias. SysML also reduces UML's size while extending its semantics to model requirements and parametric constraints. These latter capabilities are essential to support requirements engineering and performance analysis.

4.5 Requirements in the Development Process

Requirements are "alive" and "volatile". Very often they are not completely known at the start of a system's development, but rather evolve during the analysis phases of a project and beyond. Users, developers, and customers, all learn and grow during the system's development and maintenance. During the lifetime of a product, function updates, additions and possibly removal are quite common. For example, in response to the rapid changes and strong competition in markets, products are often requested to change or be customized in order to meet customers' needs. As for any other type of document or model that is subject to change and updates, it is important that requirements are managed by a tool that provides for change management, concurrent access control and versioning, access rights control, together with standard features like the association of metadata to requirements documents, ownership management and indexing.

In addition, requirements live in the context of a development process where any change in them prompts a chain of updates in the design documents and models as well as in the hardware or code implementation of the functions. Also, testing procedures are clearly affected by changes in the requirements. There are at least two sets of references that should be maintained from requirements items. One set of references/links should follow the refinement of the requirements into the design elements that are generated in response to the requirements, down to the code implementation. Another set of links should connect each requirements item to the corresponding set of (system-level or component-level) tests, that must be performed to verify satisfaction of the requirement by the system (or one of its subsystems/components).

4.5.1 Tracking Requirements into Design and Further Refinements

Being able to track requirements to design and implementation is necessary for several reasons.

- In case of requirements changes/updates suck links allows to locate quickly the part of the design and implementation that needs to be changed/modified, without the need of going through heavy documentation.

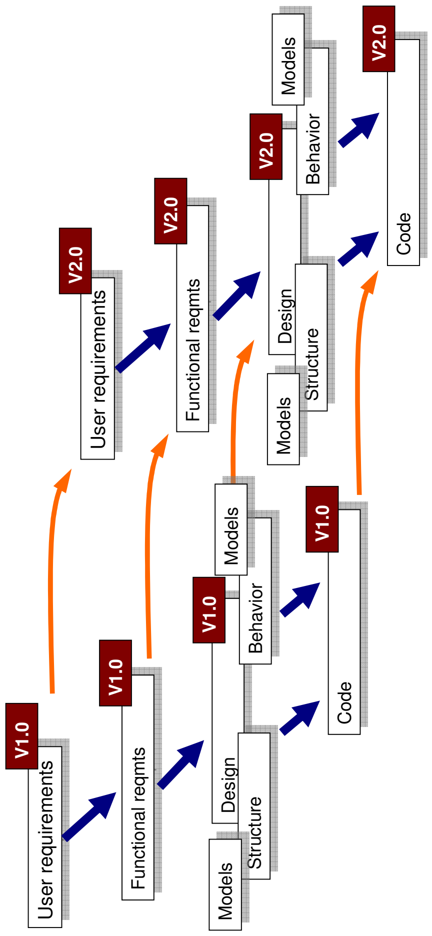


Figure 4.1: Requirements are subject to change management

- In case updates are performed on the code, because of bug fixes or updates/adjustments at test time, reverse links from code to the requirement that originated the code allow to locate the requirement that can possibly be modified.
- When system tests show failure to meet requirements, such capability rapidly provide the indication of the requirements that are affected.

When the requirement document is extensive, performing these tasks manually without the tracking between requirement and implementation become very difficult if not impossible.

4.5.2 Requirements and Testing

A different set of references should be maintained between the requirement items and the tests that are defined to verify those requirements on the final product. These links should not only be defined and maintained, but it is also important that they are defined early, at the same time requirements are produced. Following well established practices (European Space Agency process guidelines, Extreme programming, to name a few), white box or functional tests should be defined together with the requirement they are supposed to test. The enforcement of such a process and the maintenance of such links ensures the following.

1. The definition of one of more tests contextually with the definition of a requirement item helps write better requirements, ensures that they are testable and the definition of the test procedure helps clarify the meaning of the requirement (if needed).
2. Provides a measure of the coverage of the functional tests with respect to requirements. If the procedure is followed, 100% requirements coverage should be automatically achieved.
3. The existence of links between requirement items and tests allows to quickly identify and change a test or set of tests for a requirement in case the requirement is modified or updated.
4. Such links are also useful to identify the affected requirement whenever a test fails during the functional testing stage

Finally, a fundamental part of requirements management include tools and methods for change management and versioning. Even if requirements evolve, keeping track of requirement changes and how different versions of the requirements relate to different products or versions of a product or its software is necessary for building the correct tracking management system between requirements and implementations (Figure 3) Hence, a fundamental part of the project is the selection and use of a set of tools that enables requirements tracking, both with respect to design models and code implementations, and with respect to functional test descriptions. Also, these tools should provide content management and versioning of requirements, design models and code. Examples of use that can be use for this purpose are, of course DOORS (from IBM), Reqtify (from TNI), but also Word documents or HTMl-based documents, possibly maintained by a content management platform (open-source solutions exist), or stored and managed as a set of wiki pages. The generation of links to and from requirements and code implementations is possible

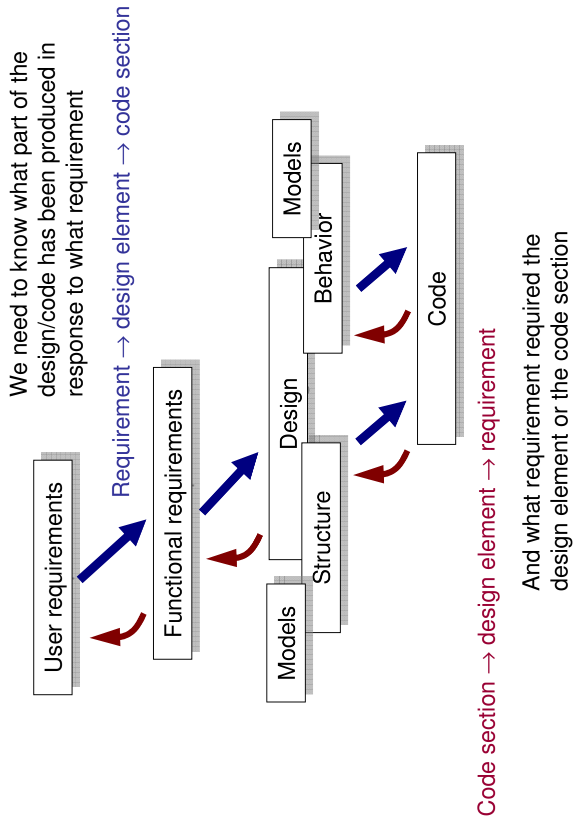

We need to know what part of the design/code has been produced in response to what requirement
Requirement → design element → code section
User requirements
Functional requirements
Design
Models
Behavior
Structure
Models
Code
Code section → design element → requirement
And what requirement required the design element or the code section

Figure 4.2: Tracking requirements

in Simulink models using the ”Report Generator” package of Mathworks. Similarly, for versioning, there are several options, including open source packages like svn or cvs or commercial solutions like ClearCase, Visual SourceSafe, Synergy.

4.6 Tools and Methods for Requirements Engineering

Among tools for the creation, management, tracking and versioning of requirements, IBM Rational (formerly Telelogic) DOORS is probably today the most popular in the industry. DOORS helps capture, track and manage user requirements. It is strongly oriented towards informal specification handling. It does not promote, provide or directly support any formal language for specification, but provides a framework in which formal definitions and propositions can be linked to specification objects and possibly handled by external tools.

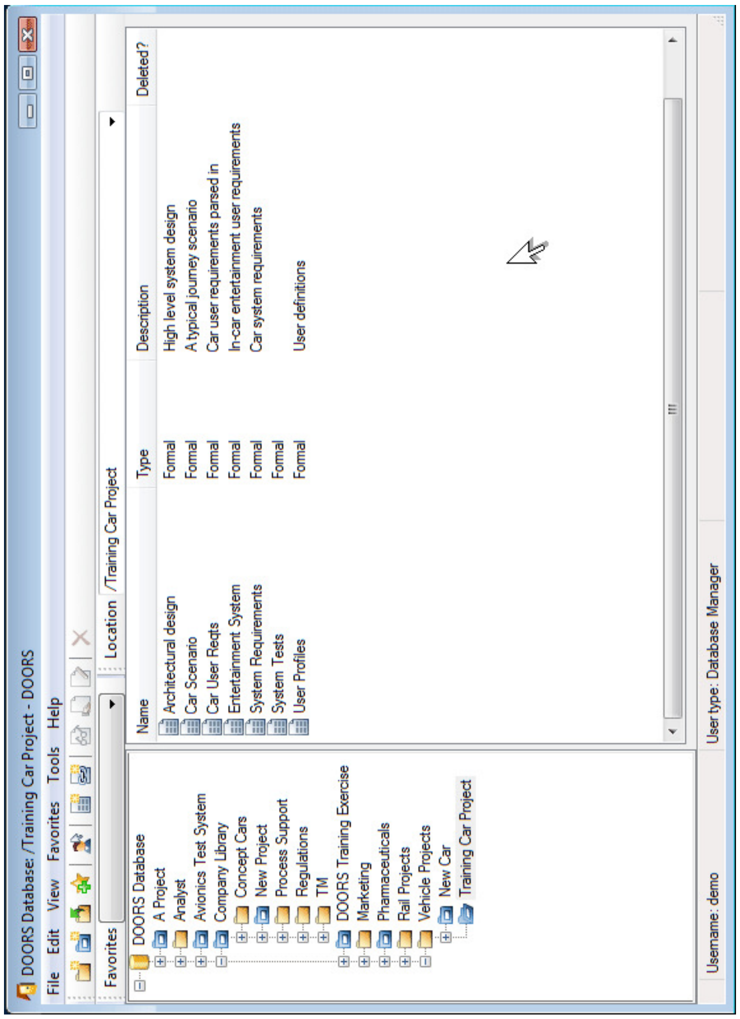
DOORS manages a database in which information about *Projects* is stored and managed. Each Project has *Folders*, *Formal Modules*, *Link Modules* and *Descriptive Modules*. Folders are provided for organization of Modules, which are used to organize requirements and specifications according to a user-defined taxonomy. For example, in Figure 4.3 the project *Training car project* consists of several Formal Modules, containing the requirements, but also user profiles, the definition of tests, and architecture design documents.

Each Module consists of Sections and each Section, in turn, contains Objects. Each Object is an element of the specification, consisting of an Header or Text (possibly both, although it is not recommended), or a picture or a table. Each Object is characterized by a set of default attributes, including:

- A unique identifier, assigned by the Tool
- Information on who created the object and when, as well as the creation mode
- When it was last modified and by who
- The Heading, short text and Text information
- Picture info, in case the object is a picture
- Table attributes, in case it is a table

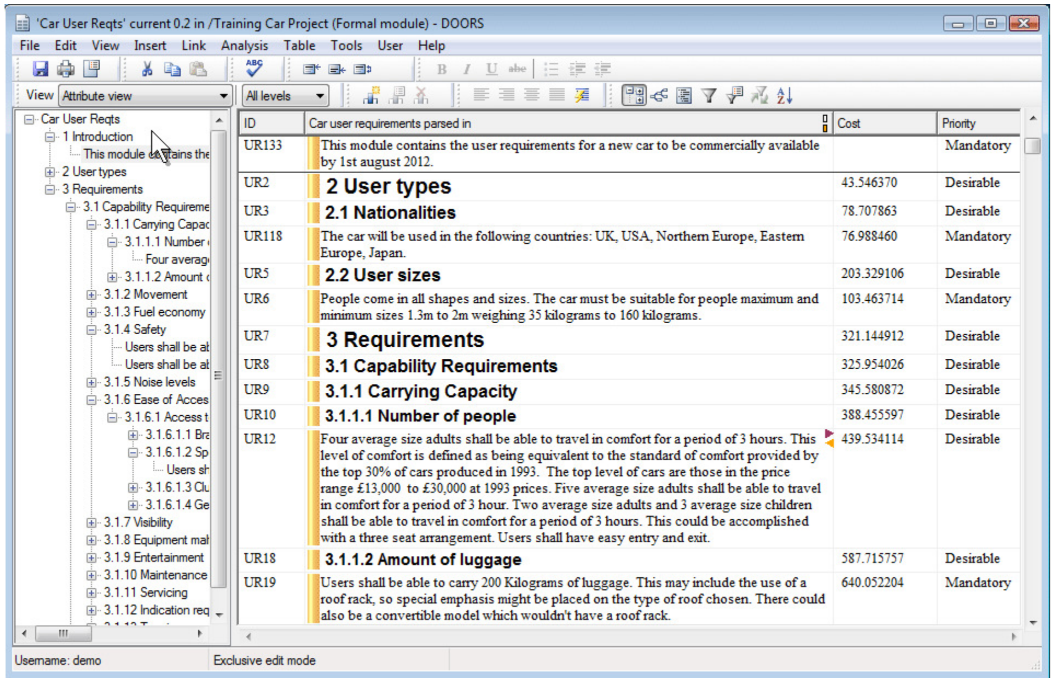
Further, users can customize objects by adding attributes. Each attribute is assigned a type. DOORS provides predefined types, but allows the user to derive custom types from them. Availability of a type checking feature helps enforcing the consistency of the value declarations for specification objects with respect to types. Figure 4.5 shows the objects that make the Formal Module *Car User Reqts* of Figure 4.3. The first column shows the identifier associated to each object. Some of them consist only of the header. Others contain text. Two attributes for Cost and Priority are also associated to Objects. Requirements objects contain requirements items expressed (in this case) in plain English. This is often the case. However, it is in principle possible to use the text field for formal formulas, to be parsed and analyzed by external tools.

In DOORS, access to objects can be controlled in several ways:
access rights
concurrent access control and versioning



Name	Type	Description	Deleted?
Architectural design	Formal	High level system design	
Car Scenario	Formal	A typical journey scenario	
Car User Reqts	Formal	Car user requirements parsed in	
Entertainment System	Formal	In-car entertainment user requirements	
System Requirements	Formal	Car system requirements	
System Tests	Formal		
User Profiles	Formal	User definitions	

Figure 4.3: DOORS screenshot showing Formal Modules



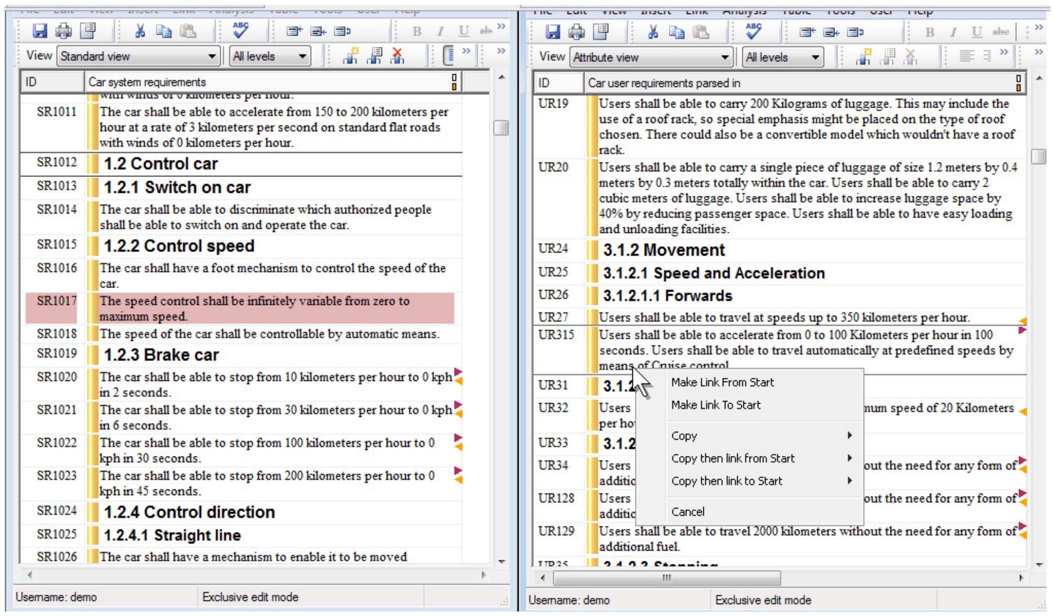
ID	Car user requirements parsed in	Cost	Priority
UR133	This module contains the user requirements for a new car to be commercially available by 1st august 2012.		Mandatory
UR2	**2 User types**	43.546370	Desirable
UR3	**2.1 Nationalities**	78.707863	Desirable
UR118	The car will be used in the following countries: UK, USA, Northern Europe, Eastern Europe, Japan.	76.988460	Mandatory
UR5	**2.2 User sizes**	203.329106	Desirable
UR6	People come in all shapes and sizes. The car must be suitable for people maximum and minimum sizes 1.3m to 2m weighing 35 kilograms to 160 kilograms.	103.463714	Mandatory
UR7	**3 Requirements**	321.144912	Desirable
UR8	**3.1 Capability Requirements**	325.954026	Desirable
UR9	**3.1.1 Carrying Capacity**	345.580872	Desirable
UR10	**3.1.1.1 Number of people**	388.455597	Desirable
UR12	Four average size adults shall be able to travel in comfort for a period of 3 hours. This level of comfort is defined as being equivalent to the standard of comfort provided by the top 30% of cars produced in 1993. The top level of cars are those in the price range £13,000 to £30,000 at 1993 prices. Five average size adults shall be able to travel in comfort for a period of 3 hour. Two average size adults and 3 average size children shall be able to travel in comfort for a period of 3 hours. This could be accomplished with a three seat arrangement. Users shall have easy entry and exit.	439.534114	Desirable
UR18	**3.1.1.2 Amount of luggage**	587.715757	Desirable
UR19	Users shall be able to carry 200 Kilograms of luggage. This may include the use of a roof rack, so special emphasis might be placed on the type of roof chosen. There could also be a convertible model which wouldn't have a roof rack.	640.052204	Mandatory

Figure 4.4: Objects inside a Formal Module

However, the most notable feature of DOORS is probably the capability of managing the links among objects as well as between objects and external models or even code.

4.7 Concluding Remarks

Specifying a system is today an art more than a science but it does have an essential role in the final quality of the design both in terms of quality and performance. Several attempts have been done in the past in the area of software system design to formalize specification or requirement capture but these efforts have not made their way through to system level design. We have reviewed briefly some of the approaches to requirement engineering and we have derived some principles that can be used to understand the issues surrounding the topic and that may yield to a novel way of capturing and analyzing requirements and specifications.



ID	Car system requirements
SR1011	The car shall be able to accelerate from 150 to 200 kilometers per hour at a rate of 3 kilometers per second on standard flat roads with winds of 0 kilometers per hour.
SR1012	1.2 Control car
SR1013	1.2.1 Switch on car
SR1014	The car shall be able to discriminate which authorized people shall be able to switch on and operate the car.
SR1015	1.2.2 Control speed
SR1016	The car shall have a foot mechanism to control the speed of the car.
SR1017	The speed control shall be infinitely variable from zero to maximum speed.
SR1018	The speed of the car shall be controllable by automatic means.
SR1019	1.2.3 Brake car
SR1020	The car shall be able to stop from 10 kilometers per hour to 0 kph in 2 seconds.
SR1021	The car shall be able to stop from 30 kilometers per hour to 0 kph in 6 seconds.
SR1022	The car shall be able to stop from 100 kilometers per hour to 0 kph in 30 seconds.
SR1023	The car shall be able to stop from 200 kilometers per hour to 0 kph in 45 seconds.
SR1024	1.2.4 Control direction
SR1025	1.2.4.1 Straight line
SR1026	The car shall have a mechanism to enable it to be moved

ID	Car user requirements parsed in
UR19	Users shall be able to carry 200 Kilograms of luggage. This may include the use of a roof rack, so special emphasis might be placed on the type of roof chosen. There could also be a convertible model which wouldn't have a roof rack.
UR20	Users shall be able to carry a single piece of luggage of size 1.2 meters by 0.4 meters by 0.3 meters totally within the car. Users shall be able to carry 2 cubic meters of luggage. Users shall be able to increase luggage space by 40% by reducing passenger space. Users shall be able to have easy loading and unloading facilities.
UR24	3.1.2 Movement
UR25	3.1.2.1 Speed and Acceleration
UR26	3.1.2.1.1 Forwards
UR27	Users shall be able to travel at speeds up to 350 kilometers per hour.
UR315	Users shall be able to accelerate from 0 to 100 Kilometers per hour in 100 seconds. Users shall be able to travel automatically at predefined speeds by means of Cruise control
UR31	3.1.2
UR32	Users … mum speed of 20 Kilometers per ho
UR33	3.1.2
UR34	Users … out the need for any form of additic
UR128	Users … out the need for any form of additic
UR129	Users shall be able to travel 2000 kilometers without the need for any form of additional fuel.

Figure 4.5: Link to (simulink) models