INVITED PAPER

Hardware/Software Codesign: The Past, the Present, and Predicting the Future

This paper reviews the past, present, and future prospects for hardware/software codesign, which is used extensively in embedded electronic system products for automobiles, industrial design automation, avionics, mobile devices, home appliances, and other products.

By Jürgen Teich, *Senior Member IEEE*

ABSTRACT | Hardware/software codesign investigates the concurrent design of hardware and software components of complex electronic systems. It tries to exploit the synergy of hardware and software with the goal to optimize and/or satisfy design constraints such as cost, performance, and power of the final product. At the same time, it targets to reduce the time-to-market frame considerably. This paper presents major achievements of two decades of research on methods and tools for hardware/software codesign by starting with a historical survey of its roots, by highlighting its major research directions and achievements until today, and finally, by predicting in which direction research in codesign might evolve in the decades to come.

KEYWORDS | Cosimulation; cosynthesis; coverification; design space exploration; electronic system level (ESL); hardware/software codesign; virtual prototyping

I. INTRODUCTION

Hardware/software codesign emerged basically as a new discipline to design complex integrated circuits (ICs) in the early 1990s. At that time, it was already clear that large 16- and 32-b microprocessors would not just be available as discrete components on board-level systems, but as software-programmable components of any IC design.

At that time, the concurrent design of hardware and software was already daily business, at least to microprocessor companies, by having carefully to decide how to design the interface between the microprocessor hardware and the software. This task involving the definition and implementation of the instruction set architecture has not, however, been consciously treated as a task of codesign yet. Nevertheless, it did motivate and stimulate those research goals that today's codesign methodologies already try to accomplish: satisfying the need for system-level design (SLD) automation, allowing the development of correct electronic systems comprising billions of transistors, running programs with million lines of codes, and finally, integrating not only a single microprocessor, but possibly multiple microprocessors on a single chip [system-on-a-chip (SoC)] with or without any accelerators, and completing such a design process within a typical 18–24-month time-to-market frame.

Driven by the technological advances predicted by Gordon Moore, hardware/software codesign techniques have become a must for a successful electronic system design today and, as such, are used more and more in companies that are developing embedded electronic system products. Here, the number and range of application domains that are steadily increasing include important industrial branches such as automotive, industrial design automation, avionics, mobile devices, home appliances, and many more. Finally, for our future of expected diminishing technical progress—the *life after Moore's law*—codesign might become even more important for two reasons. On the one hand, sale numbers of successful new technical products will not be driven so much any more by

Manuscript received December 16, 2011; accepted December 20, 2011. Date of publication March 22, 2012; date of current version May 10, 2012.

The author is with the Department of Computer Science, University of Erlangen-Nuremberg, 91058 Erlangen, Germany (e-mail: teich@cs.fau.de).

Digital Object Identifier: 10.1109/JPROC.2011.2182009

just technological progress, but more and more by tools to design better quality and more reliable systems with a given technology than any competitor. On the other hand, the slowdown of technological progress will also justify to spend more design time on a careful analysis and exploration of design options.

Now, before delving into technical topics, the major purposes and intentions of hardware/software codesign are summarized. These are explained by looking at different interpretations of the syllable *co* of the word codesign.

- *Co-ordination*: Codesign techniques are used today to coordinate the design steps of interdisciplinary design groups including firmware, operating system (OS), and application developers on the software side, as well as hardware developers and chip designers on the other side to work *together* on all parts of a system. This is also the original interpretation of the Latin syllable *co* in the word codesign. The following other interpretations are more subtle.
- *Co-ncurrency*: Tight time-to-market windows force hardware and software developers to work concurrently instead of starting the firmware and software development as well as their test only after the hardware platform is available. It can be seen that codesign has provided an enormous progress to avoid this bottleneck by either starting from an *executable specification* and/or applying the concept of *virtual platforms* and *virtual prototyping* in order to run the concurrently developed software on a simulation model of the platform already at a very early stage. Also, testing and partitioning to concurrently executing software and hardware components requires special *cosimulation* techniques to reflect concurrency and synchronization of subsystems.
- *Co-rrectness*: Needless to say, the correctness challenges of complex hardware and software require techniques to not only verify the correctness of each individual subsystem, but also *coverify* their correct interactions after their integration.
- *Co-mplexity*: Of course, codesign techniques are mainly driven by the complexity of today's electronic system designs and serve as a means (or at least try) to close the well-known *design gap* to produce correctly working and highly optimized (e.g., with respect to cost, power, performance) system implementations.

This paper gives a survey on the historic achievements and current research directions of hardware/software codesign and is structured as follows. Starting with a summary of the historic roots of codesign techniques and corresponding achievements beginning in the early 1990s in Section II, the major facets and available methodologies of modern codesign are presented in Section III. In Section IV, different variants of hardware/software codesign that have simultaneously emerged are introduced and explained such as the joint design of analog and digital parts of an IC. Subsequently, we try also to predict the future of codesign methodologies that will need to emerge in the next decades. These changes and required future developments are on the one hand stimulated by clouds coming up on Moore's horizon to deal with the growing imperfection and variability at the root components of each codesign: the transistor. Here, the hardware/software tradeoff and interplay will definitely become more important in order to solve dependability issues. On the other hand, we can recognize a trend and desire to build more flexibility [1] and runtime adaption into a system design for the purpose of optimization and reduction of the system cost. Thus, we will see that error resiliency as well as adaptivity will require to change the system functionality and partitioning into hardware and software dynamically at runtime. Hence, some codesign techniques will need to move into the product so as to achieve the required adaptivity *online*. As a matter of fact, this opens a variety of new challenges that need to be solved in the future.

II. THE EVOLUTION OF CODESIGN

In this section, the evolution of codesign is summarized starting as an emerging discipline in the early 1990s to a *mainstream engineering technology* in its second generation.

Note that the literature available on major achievements as well as tools and frameworks on codesign has become so vast[1] that it is impossible to survey all of this meritorious work in a single paper. Therefore, the major research themes addressed in each decade of research may only be emphasized by highlighting some important approaches.

A. Early Work

Although the joint design of hardware and software has been exercised already since the introduction of the first microprocessor by the joint engineering of hardware architecture and instruction set architecture, the discovery and subsequent boom by considering codesign as an essential step toward SLD automation of complex electronic systems started in the 1980s [2] and early 1990s [3], [4], respectively.

Hardware/software codesign, sometimes also named software/hardware codesign or just codesign equivalently, started to be considered as the process of concurrent and coordinated design of an electronic system comprising hardware as well as software components based on a system description that is implementation independent by the aid of design automation.

In the initial work by Prakash and Parker [5], a cosynthesis problem was formulated as a mixed integer-linear

[1]This is similar to the amount of different writing options available for the discipline of hardware/software codesign itself.

program which simultaneously determines a multiprocessor topology as well as a schedule and allocation of tasks onto the architecture. Since then, the problem of automatically partitioning tasks or processes into hardware and software was soon recognized as an increasingly important research topic with the advent of forming a research community starting in 1992 [6]. This first IFIP International Workshop on Hardware/Software Codesign took place in Grassau, Germany. Under this name or simply CODES/CASHE, major subsequent events dedicated to this new research field were organized in Colorado (1992), Cambridge, MA (1993) and Grenoble, France (1994), respectively. Later, the event was simply called CODES and organized in places such as Braunschweig, Germany (1997), Seattle, WA (1998), Rome, Italy (1999), San Diego, CA (2000), Copenhagen, Denmark (2001), and Estes Park, CO (2002). Finally, in 2003, CODES became the International Conference on Hardware/Software Codesign and System Synthesis [7] as a merger with the IEEE International Symposium on System Synthesis. Since then, it has been still the major yearly conference event worldwide focusing on all issues of hardware/software codesign. Since 2006, this important event has been organized under the direction of the Embedded Systems Week (ESWEEK) [7] with other major conferences such as EMSOFT and CASES.

B. Codesign Achievements: The First Generation

In the early years, much focus had been put on tackling the problem of partitioning a given functional specification into hardware and software, hence a problem of *bipartitioning*. A given functional specification such as a set of communicating modules or tasks specified in a C-like dialect is mapped onto a given hardware platform consisting of a single processor central processing unit (CPU) and a user-defined (application-specific) hardware block or application-specific integrated circuit (ASIC) with both parts communicating over a bus and using a shared memory or registers for buffer implementation. Hence, except for the question which parts to implement on the ASIC, the target platform was already fixed. Here, two initial quite complementary approaches are worth highlighting: In the Vulcan approach developed by Gupta *et al.* in Stanford, CA [8], the idea was to start with a hardware-only solution and to migrate as many tasks as possible to software while satisfying performance constraints with the goal to reduce design costs. The Cosyma design system developed simultaneously at the Technical University of Braunschweig [9] took exactly the opposite point of departure by starting with a software-only partition of blocks with subsequent migration of tasks to hardware in order to satisfy performance constraints while trying to minimize the cost of the resulting hardware blocks. Even if these two approaches also involved the application of high-level hardware synthesis within the partitioning loop to estimate the performance of different hardware implementations of tasks, both early approaches were quite restricted not only by the simple architectural model, but also in the execution model: both assumed that the implementation was single-threaded and that the CPU and ASIC worked mutually exclusively. Thus, the CPU had to wait in an idle mode for the hardware to complete a function and was thus treated rather as an accelerator and not even as a coprocessor. Nevertheless, these two early approaches stimulated a lot of work on subsequent investigations of partitioning approaches and architectural extensions.

C. Codesign Achievements: The Second Generation

In the next few years and until the early 2000s, not only the problem of hardware/software partitioning was elaborated on more and extended considerably for more complex types of architectures, including more than one CPU, but also the assumption of just single-threaded program execution was extended to multiprogramming and multiprocessing. Finally, also *cosimulation* [10], [11] started to become an important area of research for the early validation of design decisions.

In cosimulation, the execution of software on the CPU is simulated using a virtual model of the processor hardware or together with the synthesized hardware part of the system design. The reason for doing so is the complexity of the system design when performing a pure gate level or register transfer level (RTL) hardware simulation, which is typically much too slow. Therefore, the processors need to be modeled at a higher level of abstraction than the implemented hardware part. According to [12], the cosimulation problem lies in coupling different models to make the hardware simulation sufficiently accurate. An example product at that time was Seamless CVS from Mentor Graphics that used a processor model (i.e., an instruction set simulator) and bus models to abstract processor interaction with the memory. Although an even more abstract behavioral modeling of the software may abstract completely from the hardware by only modeling the interface timing behavior, and by allowing the software simulation to run on any host coupled to a simulator of the hardware parts, timing and performance analysis is typically restricted here to the hardware part. In the area of cosimulation, Zivojnovic and Meyr [11] proposed to also use compiled simulation to speed up cosimulation. An important framework for cosimulation of heterogeneous systems was and still is the Ptolemy framework from the University of California at Berkeley [13]. It may be used to cosimulate and understand the relationships between several models of computation such as data flow and discrete-event models. Also, formal techniques for timing analysis have evolved in the community, notably Li and Malik's algorithms for worst case execution time (WCET) analysis [14] and the application and performance analysis of scheduling techniques known from real-time systems to estimate best case, worst case, and average case timing behavior of mixed hardware/software systems adequately. As performance is one of the most critical factors for optimization or used as a constraint during codesign, the *real-time calculus* as developed by Thiele *et al.* [15] in the late 1990s and the work on compositional timing analysis by Ernst *et al.* [16] are also noteworthy.

Of course, the architectural assumptions of the presented early works do not hold in today's complex SoC architectures. First, a system may contain not only one, but several CPUs of different types such as reduced instruction set computer (RISC) cores, digital signal processors (DSPs), application-specific instruction set processors (ASIPs), or very long instruction word (VLIW) processors. In 1998, a graph-based approach was published in [17] and presented earlier at CODES 1997 [18] that allows to formally model heterogeneous target architectures including their interconnect. A *specification graph* consisting of a task graph with data dependencies and an architecture graph describing the variety of available hardware components and their communication facilities was defined there with the goal to formalize and generalize the partitioning problem. In this graph-based model, edges between tasks and resources are used to describe restricted mapping possibilities. Each mapping edge may be annotated with any number of cost attributes such as the code size, power, and other mapping-related quality attributes. Also, the timing delays caused by transporting data of the communicating tasks over the communication links are considered and mapped by introducing the so-called communication tasks between the data-dependent tasks and mapping these onto communication resources as an integral part of the partitioning problem. Also in [17], it was shown that the problem of finding a feasible allocation of a set of resources and mapping (binding) of tasks onto these resources is NP-complete. Hence, finding a feasible implementation based on this formalized model of application and architecture graph motivates the application of sophisticated approximation algorithms. The above idea and benefits of separately modeling the application and target architecture spread fast in the community and were refined and developed much further under the name of platform-based design (PBD); see, e.g., [19]–[21].

When looking at even more realistic SoC implementations, as of today, also complex communication networks need to be considered for performance and cost analysis, including not only processor buses and point-to-point connections, but also network-on-a-chip (NoC) type of interconnect. One example of such a complex multiprocessor SoC (MPSoC) target architecture is shown in Fig. 7. The communication of tasks may thus involve several hops and require routing to be determined on top of the mapping of tasks; see, e.g., [22]. Hence, the early bipartitioning approaches that were based on the assumption that the architecture is given and fixed are not applicable when designing a complex SoC. Also, single-objective optimization techniques such as the minimization of hardware cost under performance constraints or maximizing performance under cost constraints were seen as too restrictive and too special as each product in mind may have completely different objectives, including cost, power consumption, or recently also reliability [23]. In order to be able to *explore the design space* of different hardware solutions and partitions, it is thus important to not only allow a designer to implement his/her own evaluation functions easily into a computer-aided design (CAD) environment for codesign, but also to allow multiple objectives to be considered simultaneously and without any *a priori* bias or weighting. Finally, also the problem of *interface synthesis* between hardware and software parts of an SoC was recognized in the codesign community as an important field of research; see, e.g., [24]. In this area, methods to generate interface circuits from timing diagrams, or Petri nets automatically, were investigated. Also, the important problem of automatic refinement of abstract communication protocols such as in process networks onto bus protocols falls in this area. One of the first early attempts to not only propose synthesis for hardware and software alone, but also offer refinement techniques for interfaces is the CoWare system [25].

In the following section, important new achievements to adequately treat deficiencies of the first-generation approaches and tools for codesign are summarized.

For more detailed surveys of the state of the art at that time, the surveys by Ernst [12] of 1998 and by Wolf [26], which appeared in 2003, are recommended.

III. FACETS AND ACHIEVEMENTS OF MODERN ESL-BASED CODESIGN: CODESIGN 3.0

Today, we already live in a third generation of codesign technology with cross-level design environments for the synthesis of complex electronic systems becoming slowly available. During the last decade, many important milestones of progress with respect to the initial findings have been achieved. They were mainly driven by the following system design challenges.

- *Heterogeneous SoC technology* has become a reality today through the advances in microelectronics and nanoelectronics. Today, a complex system comprising several microprocessors of different types ranging from digital signal processors (DSP), application-specific instruction set processors (ASIPs) for special domain-specific functions, and microcontrollers can be integrated into a single multibillion transistor SoC together with customizable hardware accelerator intellectual property (IP) blocks and analog IPs, peripherals, and memory blocks.
- *Hardware and software complexity*: The hardware complexity of many electronic embedded systems is not only manifesting itself in SoC technology on a single chip, but also at the level of distributed communicating electronic devices such as electronic control units (ECUs) in a modern car. Here, as many as 70–90 ECUs providing special services such as stability control, antilock braking, or entertainment functions, are interconnected. So, not only the chip industry but also many companies designing embedded systems have discovered the need to deal with

the enormous complexity of today's systems. Indeed, modeling, simulation, optimization, and synthesis of such networked systems should also be in the scope of codesign technology. Finally, the enormous complexity has risen even faster in the software world for embedded systems. Imagine that again in a single vehicle, more than 100 million lines of code coexist and coexecute today. Imagine also the complexity of testing and verifying properties such as safety in such a complex system.

- *Integration panacea*: From the view of the developer of an electronic *embedded system* [27]–[30], a final obstacle has been and still is the lack of standards and ways to describe and integrate subsystems developed by potentially other companies and reuse these in order to respect reasonable time-to-market windows. Again, this holds very true for the automotive domain. Here, integration of ECUs usually starts on a test board that connects different subsystems. Heavy testing scenarios are then applied to analyze functional correctness and detect potential timing errors before an integration into the vehicle happens. Interestingly, the integration tests are done very often without having and knowing the individual software implemented in each subsystem as these are developed by suppliers. AUTomotive Open System ARchitecture (AUTOSAR) [31] is an open and standardized automotive software architecture, jointly developed by automobile manufacturers, suppliers, and tool developers. One of its major goals is to facilitate the exchange and update of software and hardware over the service life of the vehicle.

Driven by the above challenges, it was soon discovered that there must be a way to raise the abstraction level at which designers express their systems under design, giving birth to the idea of electronic system level (ESL) design as well as ways to interface and reuse designs across different abstraction levels. In the following, some major milestones are summarized, considering the time from roughly 2005 until today as the *third generation of codesign*.

A. Reduction of the Time-to-Market Frame and Design Risks Through the Concurrent Analysis, Exploration, and Design of Hardware and Software

Fig. 1(a) shows a typical development timeline of a classical electronic system design involving hardware as well as software components. Let us assume that the typical time frame of a company's product is bound to a maximum of 24 months. Here, based on a common specification and maybe on some initial high-level simulations, hardware decisions are taken first. In the worst case, the firmware and software teams can therefore not start to develop and test their software until the hardware design is available. This also has the great risk of delaying the whole product design chain in case conceptual hardware design errors or

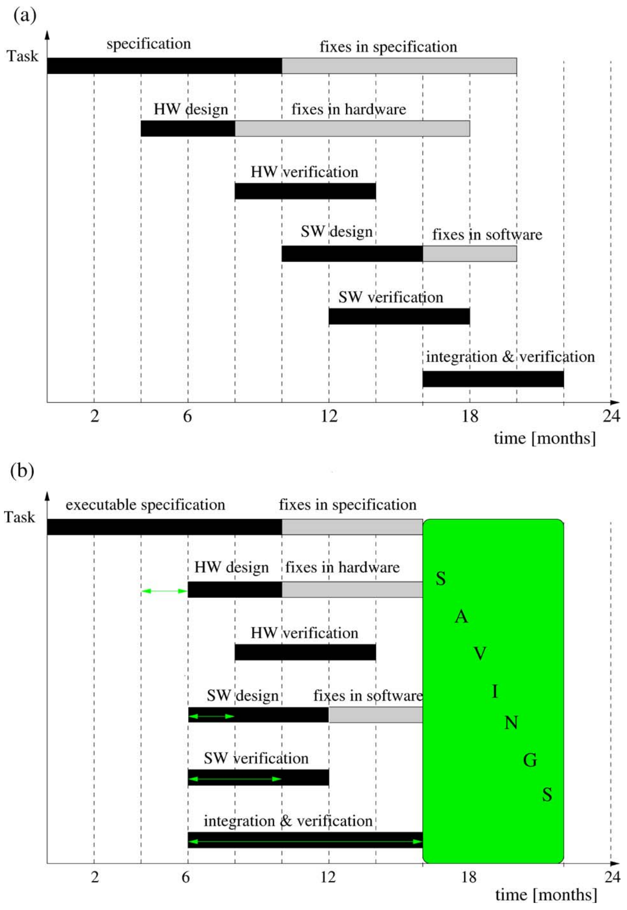


Fig. 1. ***(a) Classical design flow and (b) ESL design flow starting from an executable specification and allowing for concurrent development of hardware and software after an initial delay for specification and design space exploration. Savings of up to six months may be expected. At the same time, the risk of late design errors and of overdesigning and underdesigning a system is reduced.***

production errors are detected late or even only once the software is running on the available prototype. Also, there is no way to explore potential options for different choices of implementations with respect to multiple objectives such as cost versus performance versus extendibility, etc., so that typically, the hardware design decisions are taken based on the experience of a hardware designer team and the progress and success of the software by a software engineering team.

The shortcomings of this classical design chain are multifold:

- *long critical path* resulting in long and often unpredictable time-to-market frame;
- risks for *potential errors* in each part of the design chain uncovered only very late;
- risk for *overdesigning or underdesigning a system* due to the missing early evaluation of design options.

In Fig. 1(b), the experience is shown of companies using modern *ESL-based codesign* tools such as SystemCoDesigner [32], an ESL tool developed by our group. Here, the initial system specification starts with a functional, but already executable, specification of the system, e.g., in C, C++, or SystemC. Also, depending on an existing platform or hardware IPs, a platform model needs to be developed including cost models. Based on this initial modeling overhead, an early design space exploration of potential solutions in a choice of system components, interconnect, and memory layout as well as the distribution of software functions, and thus design tradeoffs, is possible. This step, of course, requires a big rethinking as this may take a considerable amount of time for modeling the system architecture, for parameterizing the models for design space exploration, and also for calibrating these accordingly. Nevertheless, the benefits will be great in the sense that the executable specification may be used as a golden reference model for the software development process, including the firmware and testing environment. Indeed, such a specification and the use of cosimulation tools may also help to reveal errors in the test and firmware development cycle very early. The risk of hidden errors or overdesigning or underdesigning a system with respect to the given constraints may also be reduced considerably. According to our experience, the advantage of introducing codesign may lead to savings of up to six months in the time-to-market frame in many cases, as shown in Fig. 1(b).

B. The Double Roof Model of Codesign

Apart from the necessity of specification, formal analysis, and cosimulation tools for performance and cost analysis, it was soon discovered and agreed on that the major synthesis problem in codesign of electronic systems, in the following also synonymously called ESL or SLD in [32]–[34] today, involves three major so-called *mapping* tasks.

- *Allocation*: Select a set of system resources including processors, hardware IP blocks (e.g., interfaces, memories, etc.), and their interconnection network, thereby composing the system architecture in terms of resources. These resources could be existing as library templates. Alternatively, the design flow should be able to synthesize them.
- *Binding*: Map functionality (e.g., tasks, processes, functions, or basic blocks) onto processing resources, variables and data structures onto memories, and communications to routes between corresponding resources.
- *Scheduling*: Determine when functions are executed on the proper resources including function execution, memory accesses, and communication. This might involve either the definition of a partial order of execution or the specification of schedulers for each CPU and communication and memory resources involved as well as task priorities, etc.

Hence, in contrast to the early work in codesign, the *platform* or resource allocation is not assumed fixed but rather considered a part of the so-called *design space* of different allocations, bindings, and schedule decisions. Also, the binding does not only consider mutually exclusive shared memory and single bus communications, but also it must determine *routes* from sources to destination, thereby reflecting very complex interconnection networks of today's very large-scale integration (VLSI) designs, including not only bus structures but also networks-on-a-chip (NoCs).

Codesign, however, does not only provide important design aids at the system level. At the same time, it should allow to combine existing (semi)automated design steps and interface different abstraction levels to a large degree. Thereby, codesign will accomplish the necessary *design refinements* automatically, save development time, and allow for the fast verification of the above design steps [29].

In the *double roof model* (Fig. 2) according to [28] and [35], typical abstraction levels of electronic design automation are depicted.

The double roof model defines the typical *top–down* design process for embedded hardware/software systems. The left-hand side of the roof shows typical abstraction levels encountered during the *software design process*, whereas the right-hand side corresponds to typical refinement steps during the *hardware design process*. Each side is organized in different *abstraction levels*, e.g., module (task) and block (instruction) levels for typical software design steps, and architecture and logic levels for the hardware design processes, respectively. There is one common level of abstraction: the ESL that has been described above and at which one cannot yet distinguish between hardware and software.

Now, the two shown roofs describe the typical two views a developer encounters when designing a complex hardware/software system. The upper roof describes the *functional* or *specification* view of the system at the corresponding abstraction level, whereas the lower roof describes its *structural implementation*, including allocated

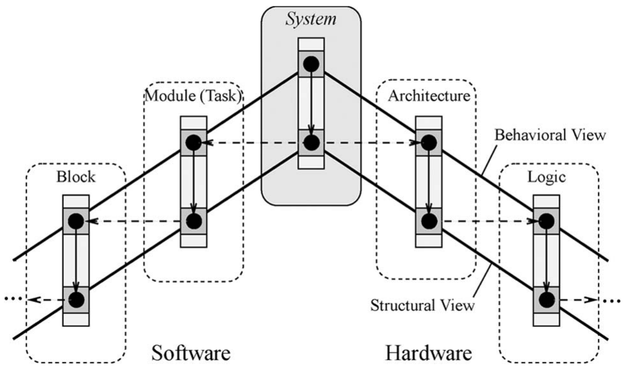


Fig. 2. ***The double roof model of codesign. Shown is the system level at the top connecting the software (left) and hardware development chains (right) through successive synthesis refinements (vertical arrows). Each synthesis step maps a functional specification onto a structural implementation on the next lower level of abstraction.***

resources as well as schedule and binding decisions and the corresponding code.

Design automation is visualized in the double roof model by vertical arrows, each representing a *synthesis step*. For example, during logic synthesis, a given specification of a system of Boolean equations or a finite-state machine (FSM) provided in the form of either a table, diagram, or alternatively hardware description language (HDL) specification is given at the level of the functional roof. Logic synthesis then generates a netlist implementing this FSM by choosing variable encodings, applying logic minimization, and finally allocating logic gates and memory elements from a library. So, at the structural roof of the diagram, one would see the netlist as a result of the refinement. Hence, through a synthesis step, a *specification* is transformed into an *implementation* at the next lower level of abstraction. Horizontal arrows indicate the step of passing information about the implementation at a certain level directly to the next lower level of abstraction as an additional specification information or constraints. For example, at the architecture level on the hardware side, the allocation would involve determining how many functional units such as multipliers and adders of each type will be chosen to compose the data path of the resulting hardware component. Here, the information on the choice of the functional unit such as a carry-ripple-adder with a certain precision would serve as an additional input at the functional level for subsequent logic synthesis of this adder.

The double roof model can be seen as extending the Y-chart [36] by an explicit separation of software and hardware design. Of course, there is no fully automated design flow for all shown abstraction levels available today. Also, each design might require different refinement steps and different tools to choose from during the design of an embedded system.

Also, a pure top–down design might not be possible or desirable for many companies in many product cases. For example, some components such as CPU types and numbers might have been chosen already early or will be reused in new product lines and thus need to be considered during the mapping. Similarly, during hardware design, some existing IP blocks available in the company will be instantiated instead of synthesizing them from the scratch. Such a *meet-in-the-middle* design strategy is not a contradiction, but rather a special case of the design flow, because we will see that resource allocations might either be fixed or chosen during a synthesis step on the basis of synthesis constraints. In particular, it will be shown later that the usage of existing components may be achieved by constraining the synthesis problem to use already preallocated components while at the same time reducing the design space.

Now, let us look at the typical design flow more closely. The design process represented by the double roof model typically starts with an ESL specification given by a functional (behavioral) specification of the whole system (either *model based* or *language based*). Here, the functional entities to be mapped are typically communicating tasks, processes, or subalgorithms that are part of the specification. Additionally, a set of mapping constraints and implementation constraints (maximum area, minimal throughput, etc.) is given. The platform model at ESL is typically a structural model consisting of architectural components such as processors, buses, and other interconnect components such as links and NoCs, memories, and hardware IP blocks that might be used as accelerators or external communication interface blocks. The task of *ESL synthesis* is the process of selecting an appropriate platform architecture out of this variety, determining a mapping of the behavioral model onto that architecture, and generating a corresponding implementation of the behavior running on the platform. The result is a refined model containing all design decisions and typically multiple *nonfunctional quality metrics*, such as throughput, latency, area, and power consumption. If selected, components of this refined model are then used as input to the design process at lower abstraction levels, and basically each hardware component and software processor in the system architecture may be refined further separately.

Synthesis at lower levels of abstraction is a similar process. Also there, a behavioral or functional specification is refined into a structural implementation. However, depending on the abstraction level, the granularity of objects handled during synthesis differs and some decisions might be more important than others. For instance, at the *task (module) level* on the software side, communicating processes/threads are bound to one or multiple processors on which they must be prioritized and scheduled depending on an off-the-shelf real-time operating system (RTOS) or a custom-generated runtime environment. This step might also involve the generation of source code in a target programming language for subsequent code compilation. Finally, at the *block level*, each piece of software code such as a function, a method, or a basic block is (cross-)compiled and linked according to the selected processor and RTOS. Hence, the synthesis tools we encounter here are compiler tools.

On the other hand, at the *architecture level* on the hardware side, processes and tasks selected to be implemented as hardware accelerators are synthesized down to an RTL description in the form of controller state machines that drive a data path consisting of functional units, register files, memories, and proper interconnect. This refinement step is commonly referred to as *behavioral*, *architectural*, or *high-level synthesis* [37]. Today, there are many tools available to perform high-level synthesis automatically; see Cynthesizer from FORTE [38], the CyberWorkbench from NEC [39], the C-to-Silicon-Compiler from Cadence [40], and CatapultC from Mentor Graphics [41] for four commercially available products. Finally, at the *logic level*, the granularity of the objects considered during logic synthesis then corresponds to Boolean formulas implemented by logic gates and flip flops. Certainly, the refinement might continue to even lower abstraction levels such as the transistor level for physical design.

In summary, the double roof model tries not only to put into perspective the *system level* as a new and important abstraction level for the design of electronic embedded systems, but also to concatenate existing design abstraction and synthesis levels for their integration and interplay. It will be shown later when discussing the requirements of modern SLD frameworks that the integration of certain design tools between abstraction levels is not always free and easy to achieve and still needs either some manual interaction or *customizability*. This is a very important feature in ESL design, as each company might want to customize its own tool suite. Even for each individual product, such a tool chain must be customizable according to different design objectives.

Nevertheless, the double roof model helps to reason about the mentioned three major challenges of hardware and software design complexity as well as the problem of integration and reuse of subsystems designed at lower levels of abstraction. As a matter of fact, the three major design tasks that need to be provided by any synthesis tool, namely *allocation*, *binding*, and *scheduling* problems, are existent on each of the presented abstraction levels. In [28], it is shown that the differences are mainly different types of resources, different optimization goals, and different optimization techniques to solve these three abstract problem classes.

Later in Section III-E, we will introduce in more detail one ESL design framework called SystemCoDesigner [32] to give an example of a cross-level ESL design framework that has already been applied successfully to many application areas and systems, varying from SoC design to the design of networked embedded systems such as networks of ECUs in the automotive area. It is, however, beyond the scope of this paper and also impossible to present all existing academic as well as commercially available frameworks that enable a cross-level codesign according to the double roof model presented here. Surveys such as given by Densmore *et al.* [42] provide an excellent comprehensive overview of more than 90 available point tools that focus on individual or subsets of ESL design tasks. Also, worth mentioning is the survey by Sangiovanni-Vincentelli on ESL design frameworks [33] and by Gerstlauer *et al.* [43]. The latter provides a taxonomy for ESL synthesis methodologies and also compares six available academic frameworks including Daedalus [44]–[46] developed in The Netherlands by the groups of Stefanov, Pimentel, and Deprettere; System-On-Chip Environment (SCE) [47] developed jointly between the University of Irvine and the University of Austin by the groups of Gajski, Dömer, and Gerstlauer; SystemCoDesigner [32] developed by our group in Erlangen, Germany; Metropolis [48] from the University of California Berkeley; Koski [49] developed by Kangas *et al.*; and finally PeaCE and Hope [50], [51] developed in Seoul, Korea, by the group of Soonhoi Ha.

C. Model-Based Versus Language-Based Specification of Applications and Platforms

It has been and still is an ongoing debate on which languages are preferable to be used for system specification and whether one single language should be used to fit all requirements. Experience shows that the opinions still differ here quite a lot. Some approaches favor the so-called *model-based design* over the *language-based design* because in order to allow for the analysis of system properties and for guaranteeing them during synthesis, strong mathematical formalisms are needed. In the following, an overview of the pros and cons of several languages used during codesign and also domain-specific model-based approaches is given. In [52], Lee and Sangiovanni-Vincentelli present a framework for comparing models of computation. In [53], Ha *et al.* give a survey of applying model-based design techniques to MPSoC targets.

1) Languages for Hardware, Software, and Codesign: For the design of hardware, expressing concurrency and timing is a must. Therefore, central to the so-called HDLs is the need of a concept of concurrent processes and signals. The concept of a signal differs from that of a variable in a software programming language by the fact that it carries events that appear at certain time steps and which may trigger other signals to change. Simulation of a hardware circuit is then achieved by event-based simulation. For digital circuit designers, the two dominant design languages to start from are VHDL [54] and SystemVerilog; see, e.g., [55]. The latter builds upon the widely used HDL Verilog. Both VHDL and SystemVerilog cover well all functional and structural (netlist) views shown in the hardware side (right) of the double roof model in Fig. 2. Also, the two synthesis levels of architectural (behavioral) as well as logic synthesis are well tool-supported today. However, it is not likely that these languages will be widely accepted by software designers who—at least in the area of the development of electronic embedded systems—prefer widely spread programming languages such as C or C++ instead. Therefore, the question is: Why not use an imperative widespread language such as C or C++ or its derivatives to start the system and hardware synthesis from? Although there also exist tools and frameworks that take C or extensions of C as the point of departure for hardware synthesis, the major problem is that for efficient hardware generation, parallelization techniques need to be applied to extract concurrency from a sequential specification. Often, important sources of parallelism such as complex loop specifications are not efficiently parallelizable. Also, the C or C++ language itself is not able to express timing which is not only important on the hardware side of the roof, but also on the software side, i.e., for safety-critical systems with hard real-time constraints [56].

As an intermediate solution, changes and extensions of pure C and C++ have been developed to cover both sides of the roof well. Here, the languages *SystemC* (see, e.g.,

[57]) and *SpecC* [47], [58] are well known and established. Whereas SpecC is a superset of ANSI C, SystemC is a class library of the C++ language, both offering all useful data types and concurrent programming structures known from the HDLs mentioned above. In order to simulate a SystemC program, an event-based simulation library is available so that timing aspects of a codesign may be evaluated for correctness not only functionally, but also with respect to the timing properties. Concerning hardware synthesis from SystemC, Forte [38], Cadence [40], and Mentor Graphics [41] all offer tools for hardware synthesis from subsets of SystemC, and C++, respectively.

For the development of software in safety-critical domains such as avionics, special programming languages called *synchronous* are successfully applied with strong formal semantics that permit an easier verification of properties as well as automatic code generation. Among these languages are well-known *Esterel* [59], *Signal* [60], and *Lustre* [61]. For the synchronous language *Quartz* [62] developed by Schneider *et al.*, the analysis and synthesis of MPSoC code is also investigated, e.g., by generation of OpenMP parallel code [63].

2) Important Models of Computation for Codesign: Programming languages often miss a rigorous formal semantics or are too expressive to prove important system properties such as *liveness*, *boundedness*, or nonfunctional properties such as on timing or cost. In this context, restricted models of computation such as *FSMs*, *timed automata*, *process networks*, *Petri nets*, and *data-flow* models of computation [64], [65] have their strength. Whereas the first ones are mainly used to describe, analyze, and verify reactive systems, the latter ones are mainly used to describe signal, image, and stream processing systems using the notion of *actors* [52].

D. Design Space Exploration

It has been explained that designing hardware and software separately from each other may lead to underdesigned system implementations not meeting all nonfunctional properties such as timing, cost, or power consumption. Alternatively, design decisions for the allocation of resources might have been taken too strictly, leading to too costly and thus overdesigned system implementations and reducing the later win margins per unit sold.

As a consequence, design space exploration (DSE) has soon started to become a distinguishing element of codesign technology. In the following, the basic elements and achievements in the area of design space exploration of electronic embedded systems at the system level are shortly described. Note that the ideas and techniques particular to design space exploration also hold and may be applied similarly to any other abstraction level shown in the double roof model in Fig. 2 such as the exploration of software schedules for digital signal processor targets [66] at the module level or at the architectural level, e.g., for exploring the design space of massively parallel loop accelerators [67].

In [17] and [18], the distinction between the *functional specification model* and the *architecture model* was made and the tasks of *system synthesis* were defined as the three steps of allocation of resources from the architecture model, binding of tasks or similar entities of the functional specification onto the allocated resources, and of scheduling these properly. Hence, the design space is given by the set of all possible permutations of allocations, bindings, and schedules.[2] Any such triple satisfying a certain number of additional *nonfunctional constraints* such as on cost, performance, power, temperature, etc., is called a *feasible solution*. From a feasible solution, the corresponding structural implementation can be derived easily. Often, the scheduled code for each resource may be generated automatically as a *refinement* of the initial specification.

Now, system design space exploration, as the name says, is the task to explore the set of feasible implementations 1) *efficiently* and 2) finding not only one of these, but 3) *many* and also 4) *optimal* ones. The problem here is threefold and is summarized in Fig. 3.

- *Exploration cost and exploration strategies (algorithms)*: What are good algorithms for exploring vast and in general discrete search spaces with millions of potential solutions? Of course, any known search technique might be applied to find feasible implementation candidates such as randomized search techniques, techniques relying on iterative improvement such as simulated annealing, or exact techniques based on integer-linear program (ILP) formulations.
- *Multiobjective nature and evaluation of nonfunctional properties*: However, classic single-objective search techniques like those mentioned above require a single objective function to optimize. So, in case one wants to find the best design options for two or more objectives such as cost and performance, one has to provide a weighting function to combine both objectives into one function which corresponds already to a *decision making* by the designer because of the choice of a proper weighting. In order to perform a real nonbiased design space exploration and shift the process of decision making later to the design engineer when seeing which tradeoffs for implementation are achievable, true *multiobjective* exploration techniques are advisable. Here, population-based approaches that simultaneously scan the search space such as evolutionary algorithms [17], [68] have become the state of the

[2]Note that a schedule may be a function assigning an absolute or relative start time to a task. Alternatively, it might be just a priority for a scheduling policy such as the fixed-priority scheduling, or a number indicating an absolute or partial order of execution.

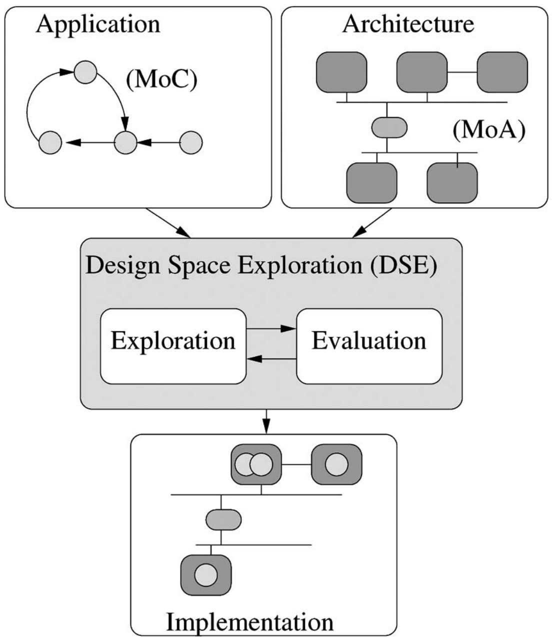


Fig. 3. ***Facets of tools for performing design space exploration (DSE) based on an application model or model of computation (MoC) and an architecture (platform) model (MoA). During DSE, the design space of implementation candidates is explored. Each synthesis candidate is evaluated according to typically multiple user-defined objectives that are implemented by evaluation functions.***

art. Special *Pareto-front*[3] exploring evolutionary algorithms such as SPEA2 [69] and NSGA-II [70] have been tuned to explore Pareto-optimal or at least close to Pareto-optimal quality sets for system-level synthesis problems very efficiently. Here, the dimensionality (number of objectives) as well as the so-called *evaluation functions* for each design objective may be chosen deliberately and in a user-specific way.

- *How to flexibly evaluate the quality of a design point?* Finally, not only the vast search space and the time for exploring it need to be taken into account, but also how flexibly different customizable objectives may be specified and evaluated. In today's exploration tools, a great flexibility is necessary as each company using such an exploration tool might not only have a different tool chain in their double roof model, but also for each individual product and a different number of objectives to evaluate at each abstraction level to be explored. For example, in case of a field-programmable gate array (FPGA) target for system implementation, the cost objectives could be the number of logic gates, flip flops, and block random access memories (RAMs) used together with two performance objectives such as throughput and clock rate of the synthesized system. For a chip on a mobile phone, the objectives could be minimizing the power consumption and chip area. Now, for their evaluation, the exploration model might be annotated and used to write user-specific cost functions. Alternatively, a synthesis or estimation tool for each objective might be used such as a worst case execution time (WCET) [71] estimation tool for determining the WCET of a task when mapped to a certain processor resource. In order to achieve this desired customizability, the concept of general evaluation functions according to Fig. 3 has been developed. Hence, the acceptance of design space exploration tools depends greatly on the flexibility of customizability and automatic integration of user- and system-specific evaluator functions and a highly efficient exploration kernel.

In Section III-E, one example of such a flexible exploration framework called *SystemCoDesigner* will be described in more detail.

1) Some Recent Advances in Design Space Exploration: In [72], Lahiri *et al.* describe a design space exploration methodology for the optimization of on-chip communication architectures. In [73], Bacivarov and Jerraya describe how to explore the interconnect architecture of a system. In [74], Pasricha and Dutt describe a framework for cosynthesis of memory and communication architectures for MPSoC targets. Similarly, Pimentel *et al.* use evolutionary algorithms for exploration of the application mapping problem in MPSoC design [68].

Another important observation when using multiobjective evolutionary algorithms for design space exploration at the ESL is that for highly constrained search spaces, they may not find any feasible solution at all, depending on the coding of the search space and nature of the space of feasible solutions. Thus, extensions have been proposed, e.g., to add the number of constraint violations as an additional objective to be minimized in order to steer the search at least toward the space of feasible solutions. Unfortunately, many points might be explored and evaluated multiple times in case no special care is taken in the implementation of the set of genetic operators that is applied to a given population of points. An important breakthrough to also efficiently explore highly constrained huge search spaces was the introduction of *symbolic techniques* [75], as shown in Fig. 4. In particular, it was shown that the problem of finding a feasible implementation may be formulated as a problem of Boolean satisfiability (SAT). Hence, a SAT solver may be used to find out whether there exists a feasible solution with respect to a number of constraints on allocation, binding,

[3]In multiobjective optimization, a solution candidate is called nondominated (with respect to a given reference set of solutions), if no solution of the reference set is (equal or) better in each objective than the considered candidate. A solution candidate is called *Pareto-optimal* in case it is a nondominated point with respect to the set of all feasible solutions as the reference set. For a given optimization problem, the *Pareto-front* denotes the set of all Pareto-optimal points.

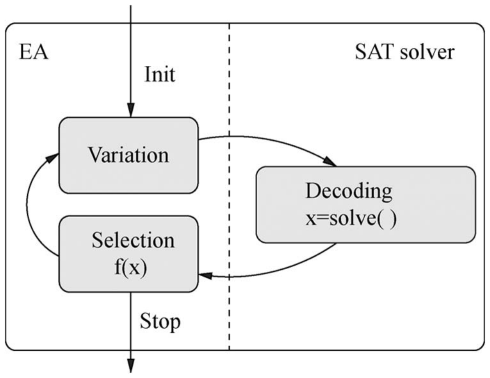


Fig. 4. ***Multiobjective DSE in SystemCoDesigner [75]. Shown is the exploration kernel according to Fig. 3 using a combination of a SAT solver and an evolutionary multiobjective algorithm (EA) that basically steers the solution strategy of the SAT solver (setting variables in which priority and to which phase, i.e., 0 or 1).***

and possibly also scheduling. According to Fig. 4, this SAT solver works in interplay with a multiobjective evolutionary algorithm (EA). This EA, however, does not directly determine (select) a number of next points to explore in the search space but rather controls and varies the strategy of the SAT solver in which priority to assign variables and to which value (0 or 1). The most recent work deals with 1) extending this approach by particular pruning functions and 2) enhancing the SAT techniques and pseudo-Boolean (PB) solvers to allow for more general, i.e., nonlinear constraints to be formulated using the concept of *background theories* [76], [77]. Finally, in [78], Wildermann *et al.* have extended symbolic design space exploration for runtime reconfigurable multimode systems.

E. SystemCoDesigner

This section is concluded by introducing one specific instance of a codesign framework to explain the above features in detail. For the introduction and comparison of other academic codesign frameworks, we refer, e.g., to [43].

The goal of the SystemCoDesigner project [32] is to automatically map applications written in SystemC [57] to a heterogeneous MPSoC platform. By automating as many design steps as possible, an early evaluation of different design options is possible. The overall design flow is shown in Fig. 5.

In the first step, the designer writes an actor-oriented application model using SystemC. In the second step, different hardware accelerators may be generated automatically for actors in the application model using behavioral synthesis and stored in a component library. This library also contains other synthesizable IP cores such as processors, buses, memories, etc. Before automatically exploring the design space, the designer has to define an MPSoC platform model from resources in the component library as well as mapping constraints for the actors. During DSE, allocations as well as bindings of tasks to processing resources and communications to routes in the architecture are determined and evaluated for objectives such as cost, power, and performance using the concept of user-definable *evaluation functions*. From the set of nondominated solutions, the designer may then select promising implementations for subsequent rapid prototyping.

1) Scope of Methodology: Currently, SystemCoDesigner supports mainly the design of streaming applications. These applications are typically modeled by the help of dataflow graphs where vertices represent actors and edges represent data dependencies. Due to the complexity of

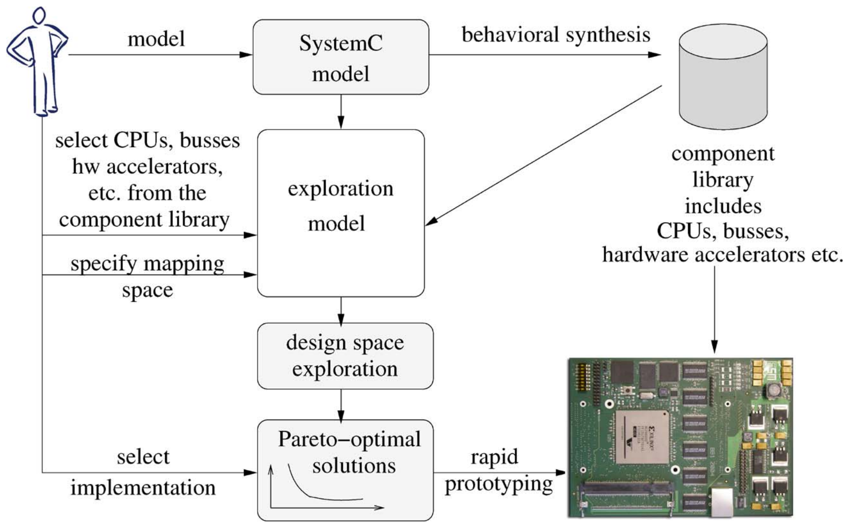


Fig. 5. ***ESL design flow using SystemCoDesigner [32].***

many streaming applications, they often cannot be modeled as static dataflow graphs [64], [65], where consumption and production rates are known at compile time. Rather they are described as a combination of static and dynamic dataflow models, e.g., Kahn process networks [79].

On the other hand, SystemC has become a *de facto* standard in industrial SLD flows. Hence, SystemCoDesigner assumes that the application model is written in SystemC and represents a dataflow model, i.e., SystemC modules (actors) only communicate via SystemC first-in-first-out (FIFO) channels and their functionality is implemented in a single SystemC thread. Such input descriptions can be transformed into a special subset of SystemC called *SysteMoC* [32]. An application modeled in SysteMoC resembles the *FunState* model of computation (functions driven by state machines) [80] that allows to express nondeterministic dynamic dataflow models.

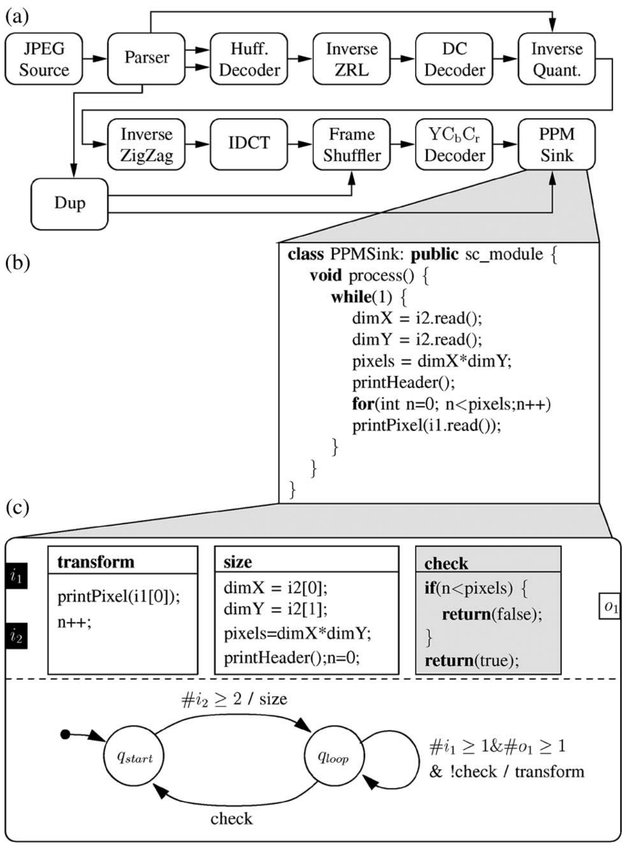
A SysteMoC model is composed of SysteMoC actors that communicate via queues with FIFO semantics. Each SysteMoC actor is defined by an FSM specifying the communication behavior and methods controlled by the FSM. If activated by the FSM, these methods are executed atomically and data consumption and production are only performed after computing a method.

As an example, Fig. 6(a) from [43] shows a Motion-JPEG decoder in SysteMoC. It consists of several actors interconnected by communication channels (edges) processing a stream of data. Fig. 6(b) exemplarily shows the SystemC definition of the PPM sink actor. The corresponding representation as a SysteMoC actor is shown in Fig. 6(c). The FSM controlling the communication behavior of the SysteMoC actor checks for available input data (e.g.,$\#i_1 \geq 1$) and available space on the output channels (e.g.,$\#o_1 \geq 1$) to store results. Furthermore, constant methods called *guards* (e.g., `check`) can be used to test values of internal variables and data in the input channels. If the predicates annotated to a state transition are evaluated to true, this transition can be taken and the annotated methods called *action* (e.g., `transform`) will be processed atomically.

From SysteMoC actor code, both hardware accelerators and software module implementations may be generated automatically [32]. The latter one is achieved by straightforward code transformations, whereas the hardware accelerators are built with Forte's Cynthesizer [38] in the loop. This allows for quick extraction of important performance parameters like the achieved throughput and the required area. These values can be used later to evaluate different solutions found during automatic design space exploration.

The generated hardware accelerators (synthesizable RTL code) are stored in the component library. From this library, including further synthesizable IP cores, the designer can specify his/her MPSoC platform template. Furthermore, the designer has to specify mapping constraints for each SysteMoC actor.

After design space exploration, the designers may select any MPSoC implementation best suited for their



```
class PPMSink: public sc_module {
  void process() {
    while(1) {
      dimX = i2.read();
      dimY = i2.read();
      pixels = dimX*dimY;
      printHeader();
      for(int n=0; n<pixels;n++)
      printPixel(i1.read());
    }
  }
}
```

```
transform
printPixel(i1[0]);
n++;
```

```
size
dimX = i2[0];
dimY = i2[1];
pixels=dimX*dimY;
printHeader();n=0;
```

```
check
if(n<pixels) {
  return(false);
}
return(true);
```

Fig. 6. ***(a) Block diagram of a Motion-JPEG decoder. (b) SystemC code of an actor that can be transformed into a SysteMoC actor given in (c) [43].***

needs. Once this selection has been made, the last step of the proposed ESL design flow is the *rapid prototyping* of a corresponding FPGA-based implementation in terms of the model refinement. For this purpose, the resulting platform is assembled. Moreover, the program code for each processor is generated according to the binding of the actors, resulting in a transaction level model (TLM). In order to generate high-quality software schedules, SystemCoDesigner supports the automatic classification of actors into synchronous or cyclostatic dataflow [81] and clustering static actors bound to the same processor into a single dynamic actor [82]. Finally, the implementation is compiled into an FPGA bit stream using the Xilinx Embedded Development Kit (EDK) [83].

2) SystemCoDesigner Design Steps: All manual work in the SystemCoDesigner design flow has been performed after setting up the platform model together with the mapping constraints. Starting with this input model, SystemCoDesigner automatically explores the design space. For this purpose, it optimizes the implementation of the streaming application while considering several objectives simultaneously, e.g., latency, throughput, area, and power consumption. While area consumption is assumed to be a linear cost function, timing and power estimation may be evaluated through a simulation-based performance evaluation during exploration.

SystemCoDesigner generates task-accurate performance models automatically from the SysteMoC model and the performance values annotated in the input model [32]. For this purpose, the platform model is translated into the so-called *virtual architecture*, again written in SystemC. The performance evaluation is done by linking the SysteMoC model to the virtual architecture. During such a performance simulation, each invocation of an action of an actor is then relayed to the virtual component the actor is bound to. The virtual component then blocks the actor's execution until the estimated execution time of the action and possible other preemption times expire. The resulting combined functional and timing simulation allows the evaluation of arbitrary complex application models. However, in case of the existence of nondeterminism in the application model, this might lead to some inaccuracy also in the performance evaluation.

Beside evaluating a single design point, design space exploration is responsible for covering the search space. In order to perform decision making automatically, SystemCoDesigner translates the input model into a PB formula. The variables of this formula encode the resource allocation, the actor binding, the queue mapping, and the routing of transactions on the communication structure such as in a complex NoC, as shown in Fig. 7. Each variable assignment satisfying this formula corresponds to a feasible implementation of the application. A PB solver is used to identify these solutions [32]; see also Fig. 4. The optimization is performed using the SPEA2 [69] multiobjective evolutionary algorithm. The decision strategy of the PB solver is varied by mutation and crossover performed by SPEA2. This variation leads to different solutions.

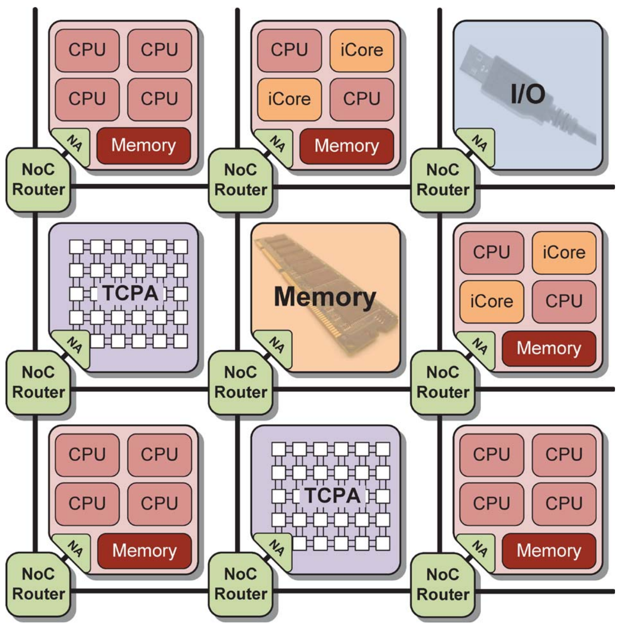


Fig. 7. ***Example of a complex MPSoC target architecture including several tiles of different processor CPUs, tightly coupled processor arrays (TCPAs), memory, and input/output (I/O) tile interconnected by a NoC [84].***

3) SystemCoDesigner Experiences: For the experimental evaluation of the SystemCoDesigner design flow, a Motion-JPEG decoder as shown in Fig. 6(a) is used; see [43] for more details. The Motion-JPEG decoder case study consists of 8000 SysteMoC lines of code, supporting interleaved and noninterleaved baseline profile without subsampling. The complete specification results in about 5×10^{33} possible implementation alternatives. Thanks to the integration of Forte Cynthesizer, the hardware accelerators for the different actors can be obtained directly from the SysteMoC specification. Furthermore, as SysteMoC offers a higher level of abstraction compared to RTL, the designer can progress more quickly. Taking the number of lines of codes as a measure for complexity, the RTL design would be 8–10 times more costly.

On the basis of the above specification, the design space has been explored using SystemCoDesigner. The objectives taken into account during design space exploration were: 1) throughput; 2) latency; 3) number of required flip flops; 4) lookup tables; and 5) block RAMs. During exploration, 7600 different solutions were evaluated in two days, 17 h, and 46 min. The simulation time per solution is about 30 s for Motion-JPEG streams consisting of four quarter common intermediate format (QCIF) frames. As a result, 366 nondominated solutions were found, each of them representing a particular hardware/software implementation.

Finally, many of these solutions were automatically prototyped onto a Xilinx Virtex II FPGA. Concerning the accuracy of model-based estimation and implementation, a discrepancy of up to 30% can be identified when comparing the FPGA implementations with the performance estimations during design space exploration. The differences in the required hardware sizes ($\leq$ 15%) occurring between the predicted values and those measured in hardware may be explained by postsynthesis optimizations such as elimination of unused block RAMs. The discrepancies between the performance estimations for latency and throughput and those measured for hardware/software solutions mainly arise due to schedule overhead on the processors.

IV. CODESIGN 4.0 OR: RESEARCH PERSPECTIVES FOR THE NEXT DECADES OF CODESIGN

In this section, some important directions that codesign needs to take in order to be able to develop complete embedded system implementations in their operating environment are mentioned. This includes, for example, the simultaneous consideration of the analog parts in an electronic system and possibly the mechanical parts as well. So far, most advances in the area of codesign have been made only in automation of the design of the digital parts of a hardware/software system. Notably, the area of cyber–physical systems (CPSs) has already moved in this direction.

Subsequently, we also try to predict the biggest challenges of codesign techniques in order to cope with many upcoming problems of *exploding design complexity* with a special focus on the use and application of *multicore processing*, and problems of *reliability* and *fault tolerance* arising from the imperfections of future nanoelectronic devices and the postsilicon era. As a result, *adaptivity* of future embedded systems to react on changes in the environment and/or internal state of operation will be required.

Nevertheless, the following predictions have to be taken under consideration carefully as they reflect personal opinions of the author only.

A. Variations and Extensions of Codesign

Recently, a new buzzword called CPS [30], [85] has brought a lot of attention to the embedded system community. According to Wikipedia, "a cyber-physical system (CPS) is a system featuring a tight combination of, and coordination between, the system's computational and physical elements. Today, a pre-cursor generation of cyber-physical systems can be found in areas as diverse as aerospace, automotive, chemical processes, civil infrastructure, energy, healthcare, manufacturing, transportation, entertainment, and consumer appliances. This generation is often referred to as embedded systems. In embedded systems the emphasis tends to be more on the computational elements, and less on an intense link between the computational and physical elements [112]."

With respect to the above definition of a CPS that steers and controls physical processes by interacting with its environment and communicating with other (sub)-systems, some extensions of previous work on codesign that take mainly the development of the digital parts into account, or focusing just on a single-chip (SoC) solutions, are of great interest and will be explained next.

1) Analog/Digital Codesign: When looking at the implementation of an electronic embedded system, it might also be of interest to investigate the tradeoff where to place the border between analog and digital signal processing, because today, analog circuits and digital circuits may be integrated on a single die. In radio applications, for example, it is common to implement the radio-frequency (RF) parts in analog circuitry, and process the remaining parts by digital circuitry and in software by digital signal processors and microcontrollers. However, the analog parts do not scale well. Due to this fact and due to the high processing power of today's microprocessors, the wireless transmission community argues that the analog/digital boundary has shifted much to *software-defined radio*. Nevertheless, at least some Ph.D. dissertations have tried to investigate the quest for the best partitioning of functionality into analog and digital circuits systematically; see, e.g., [86], including the placement of analog-to-digital (A/D) converters and digital-to-analog (D/A) converters at their boundaries, or the work by Wolff *et al.* [87].

2) Architecture/Compiler Codesign: Another facet of hardware/software codesign and need for design space exploration techniques has been recognized in the area of architecture/compiler codesign [88]. Here, the idea is to jointly develop the microarchitecture of a domain-specific processor such as an application-specific instruction set processor (ASIP) [89], [90] for a given domain of applications, given by a set of applications (programs). By exploring the joint design space of architecture variants (such as the number of issue units, register file architecture, and instruction set options) and compiler optimizations (such as the order and type of intermediate and architecture-specific code optimizations), the goal is to find the set of Pareto-optimal settings in terms of processor architecture as well as compilation flow. Of course, this might result in a considerable effort of compiler retargeting. Nevertheless, for certain processor platforms, such as customizable very large instruction word (VLIW) processors, it was shown that a considerable amount of design automation is possible in order to optimize program execution latency as well as generated program code and data footprints. See, e.g., the work by Fischer *et al.* [88] or more recent work by Leupers *et al.* [91] and O'Boyle *et al.* [92].

B. Controller/Scheduler Codesign

Also with respect to the design of CPSs, some recent work looks at jointly developing the control application of a plant with its typical objectives of stability, and energy of control together with the often distributed digital system implementation, which may be a network of controllers communicating over a bus system. Obviously, the scheduling of different control functions and the communication delays may also have an impact on the quality of control. Recent work considering these two optimization problems jointly has been reported by Samii *et al.* [93] and Schneider and Chakraborty [94], just to give two examples. Apart from the joint design space exploration of control algorithm and system implementation, the cosimulation of the environment (plant) and embedded system has become a focus of design automation; see, e.g., [95].

C. Codesign for Dependability of Future Nanoelectronic Systems

It has been shown that the double roof model of codesign (Fig. 2) is just becoming ready and waterproof, and recognized as a means to close the gap of design automation on the highest possible level, namely the design and exploration of systems. Yet, we see another leakage at its fundaments that demands renovation. According to technological roadmaps of the silicon semiconductor industry, a big threat here is the imperfection of future nanoelectronics we will experience in the next decades. As soon as we reach technology sizes where each transistor on a chip contains only a few dopant atoms, the variability of transistor switching behavior becomes severely larger. Also, effects of negative bias temperature instability (NBTI) and electromigration will lead to a severe degradation of expected lifetimes of such devices. Finally, the correctness of data processing is threatened by the increased likelihood of *soft errors* caused by particles. Here, one can see that not only memories will be affected, but also logic and wiring.

Inevitably, new techniques will have to be developed on the base of the hardware side of the double roof model to keep the dream of cross-level design automation alive. But, what needs to be done here exactly? In our opinion, this involves not only new modeling efforts of the imperfections of hardware at the transistor level but also new methods to analyze and compensate for errors at this level. Borkar [96] names these challenges as designing reliable systems from unreliable components.

On the one hand, we believe that *deterministic models* will have to be replaced and extended by *stochastic models* for the proper analysis and cosimulation of device effects. We need to work with the distributions of device behaviors and reason about expected system behavior and performance numbers. We also need to deal with time-variant behavioral models in order to cope with wear and tear effects.

Inevitably, this will lead and stimulate also new research directions in codesign in the following way. For example, whereas we have to cope with aging hardware, software obviously does not age. This opportunity might be exploited by applying software redundancy measures such as *spatial redundancy* by code replication or *temporal redundancy* by recomputation to obtain a higher level of reliability of the system. Also, in some applications, errors on the processed data might be negligible to some degree, e.g., in image or audio processing. In other areas such as (bank) transaction processing, a plethora of more sophisticated error detection and correction methods need to be applied in order to avoid any corruption of data. The full potpourri of available *coding technology* (e.g., error detecting and correcting codes) as well as *fault tolerance* (e.g., dual and triple modular redundancy) techniques may be applied here as a design space for *reliability-increasing* and *reliability-preserving* techniques with obvious tradeoffs between:

- error coverage;
- cost of error detection/correction;
- level of gained reliability.

The first works that look at reliability as a design constraint or an objective during design space exploration have already been published; see, e.g., the early work of Vargas *et al.* [97], Glaß *et al.*, [98], Bolchini *et al.* [99], Kandemir *et al.* [100], and Atienza *et al.* [101]. Pop *et al.* [102] investigate the effects of scheduling and voltage scaling for the exploration of energy/reliability tradeoffs in fault-tolerant time-triggered systems. On the software side, Marwedel *et al.* also investigate whether it makes sense to define special data type declarations in software programming languages for instructing a compiler to protect these variables against soft and other errors; see, e.g., [103].

The need for dealing with dependability issues in an embedded system design has been recognized in many countries. For example, in Germany, the German Science Foundation (DFG) has started to fund a priority program called *Dependable Embedded Systems* for research in this area in 2011 (see [104] and [105] for an overview) with J. Henkel from the Karlsruhe Institute of Technology (KIT) being the coordinator. As an another example, the National Science Foundation (NSF) has established an expedition called *Variability-Aware Software for Efficient Computing with Nanoscale Devices* [106], also gathering about a dozen of mostly multi-institutional collaborative projects across five categories of research between the University of California San Diego, University of California Los Angeles, University of California Irvine, University of Illinois at Urbana-Champaign, the University of Michigan, and Stanford University.

In the next one or two decades, we dare to predict that we will try to just fix the transistor levels of abstraction on the hardware side of the double roof model, but leave the way software as well as the system and the architecture are designed unchanged as much as possible. Nevertheless, the consideration of unreliability and aging of hardware will require comprehensive use of remedies coming from all abstraction levels including the logic level, the architecture level, and software levels.

However, it is very difficult to predict what models of computation and effects we will face in 50 years from now. Moreover, we believe that in the next 100 years we will definitely see the *postsilicon era* become a reality, *3-D design* become mature, and new principles such as carbon nanotubes as well as single-electron transistor technologies or quantum computers demanding again a completely new rethinking, remastering, or adding of new abstraction levels into the double roof model.

D. Codesign of Runtime Adaptive Systems

A final challenge to codesign we foresee is also stimulated to some degree by the expected imperfection of future nanoelectronic devices. Obviously, electronic embedded systems will require a certain degree of *runtime adaptivity* in order to cope with unpredicted and unforeseen situations, the more they become connected and the more they become *cyber–physical*.

In such scenarios of application, an electronic embedded system might not be therefore optimally designed any more when being designed statically. Here, we foresee that the classical offline system optimization and design space exploration will shift to the *runtime* more and more. This opportunity is not only available through the adaptation of software running on the platform, but also due to the advent of *reconfigurable hardware technology* such as FPGAs. Today, FPGAs allow to prototype and implement complete SoC designs including multiple processors available as either programmable *soft cores* or *hard macros* integrated into the FPGA. Tools for building hardware modules are available to not only reconfigure the whole configuration of an FPGA, but often any 1-D or 2-D *partially reconfigurable region* on the array of reconfigurable cells; see, e.g., [107].

So, *runtime adaptivity* may be technically achieved on both software and hardware sides opening the doors to thinking about *online techniques for hardware/software codesign* that run in the embedded system itself in order to achieve a situation-aware optimization of the partitioning of hardware and software. One of the first approaches proposed here was by Stitt *et al.* [108]. In [109], the results of a priority program on dynamically reconfigurable hardware that was funded between 2003 and 2009 in Germany, are reported. In [110], to give an example, a distributed online approach to hardware/software codesign for scenarios in the automotive domain was presented by Streichert *et al.* There, the adaptation runs in two phases. Upon detection of an error such as a link transmission failure, new routes of messages are determined decentrally and instantiated. In case of a node failure, software tasks are also migrated to other nodes. This self-adaptive phase is called a *repair phase*. Finally, in a phase of constant and error-free operation, the system performs an *optimization* phase where, according to a decentralized load distribution algorithm, the tasks are assigned to ready nodes depending on and for balancing their individual loads. On each node being an FPGA, the local decision of whether to instantiate an accepted task either as a hardware or a software node is also taken at runtime. Although being quite visionary and currently not ready for product integration due to questions of standardization and testing, we believe that the future will go in this direction and that enough processing power will be available in a system to perform such runtime adaptations automatically, also in the many-core era [111].

V. CONCLUSION

In this paper, we have tried to show that the application and knowledge of hardware/software codesign techniques is a must for all those who want to keep up with the challenges of more and more complex electronic system designs in the future. This does not only hold for SoC designers, but also for software and hardware engineers involved in the development of distributed systems such as complex automotive networked systems, avionics, manufacturing systems, and embedded systems in general.

In this area, the tool-supported specification, modeling, partitioning, and synthesis of subsystems at the system level is of utmost importance in order to be able to build increasingly complex products with tight nonfunctional constraints such as cost, performance, power, and also temperature and reliability in a timely manner.

Our own experience in working with quite a few industrial developers of electronic embedded systems has shown that after an initial hesitation to apply newest methodologies such as virtual prototyping, cosimulation, and design space exploration as well as cosynthesis, an early introduction of codesign techniques has meanwhile shown indispensable benefits for much better optimization results due to an early tradeoff analysis and shorter time-to-market frame due to concurrent development of hardware and software.

Nevertheless, the available tool landscape and support for codesign are still not mature in many aspects. In the future, threats such as the imperfection of future transistor technology, the complexity wall of codesign systems with 100–1000 processors [111] on a single chip that will be available by 2020, and the challenges of systems requiring runtime adaptivity with respect to dynamic requirements will need to be tackled.

Mission Statements on the Future of Codesign: In the following, conclusions and predictions on future research directions in codesign, again reflecting the individual opinion of the author, are summarized.

- *The Wall of Complexity*: SoC technology will be already dominated by 100–1000 core multiprocessing on a chip by 2020. Changes will affect the way companies design embedded software and new languages, and tool chains will need to emerge in order to cope with this enormous complexity. Development of concurrent software and exploitation of parallelism will definitely find a renaissance; this time not for applications running in computer centers and in high-performance computing, but as part of low-cost embedded system daily-life devices.
- *The Wall of Heterogeneity*: In order to cope with the quest to also include the environment in the design of future cyber–physical systems, the heterogeneity will definitely continue to grow as well in SoCs as in distributed *systems of systems*. Here, the facets of A/D, mechanical/electronic, and controller/scheduler codesign have been mentioned as interesting research directions.
- *The Wall of Dependability*: Also, as has been shown, the correctness of future electronic embedded systems is definitely menaced by the imperfection of future transistor technology. Codesign will be mandatory to detect and compensate for such errors as well as aging effects in the future. There is no doubt that changes are required at the transistor level of

Table 1 Historic View at Four Generations of Codesign as Discussed in This Paper

Gen.	Years	Major achievements
1	– 1995	HW/SW bi-partitioning for CPU-ASIC target
2	1995-2004	Co-simulation & PBD for complex targets
3	2005-2011	DSE & Co-Synthesis for Het. Multi-Core Designs
4	2012–	New Variants of Co-Design emerge; Online Co-Design for Adaptive Systems; Co-Design of Systems of Systems; Co-Design for Dependability; . . .

the double roof model. However, we are afraid that potentially also other levels of abstraction will be infiltrated and changes required such as probabilistic logic, revolutionary new ways to code data, and for computer arithmetics. This might also happen due to postsilicon inventions requiring a complete renovation of the codesign house.

- *The Need for Self-Adaptivity*: We do believe that not only due to dependability reasons but also due to the uncertainty of the environment and communication partners of complex interacting cyber–physical systems, runtime adaptivity will be a must for guaranteeing the efficiency of a system. Due to the availability of reconfigurable hardware and multicore processing, which in our view will also take a more important role in the tool chain for system simulation and evaluation, online codesign techniques might become standard in the future.
- *The Need for Cross-Layer Coverification*: One huge part of the design time according to Fig. 1 is already spent on the verification, either in a simulative way or using formal techniques. We believe that with respect to the above threats, coverification [29] will finally require an increasing proportion of the design time in the future. Progress in ESL might therefore diminish in case verification techniques cannot cope with the modeling of errors and ways to retrieve and correct them, or, even better, prove that certain properties formulated as constraints during synthesis will hold in the implementation by construction. In our view, too little effort is spent in this important area of joint coverification of hardware and software. In the double roof model (according to Fig. 2), the verification process on one level of abstraction needs to prove that an implementation (the structural view) indeed satisfies the specification (behavioral view). A textbook dedicated just to presenting which verification techniques exist today for software verification as well as hardware verification and on which level of the double roof model these may be applied has been published in 2010 [29]. In this textbook, the process of coverification is first treated independently of the levels of software or hardware and classified into the three fields of the *verification task* (what will be verified and which kind of proof will be carried out?), the *verification goal* (what will be the expected output of the verification method, e.g., a negative or positive answer?), and the *verification methodology* (e.g., a simulative or a formal proof technique or a prototype implementation). According to this classification, the book then describes which methods currently exist and may be applied at each level of the double roof model in order to verify that an implementation satisfies the properties of a specification at each level. The discussed techniques include techniques for *equivalence checking* as well as *functional* and *nonfunctional property checking*. The area of *cross-layer coverification* reveals, however, many unsolved problems that will require big research efforts in the future.

Summary Generations of Codesign: We conclude this paper with our historic view on four distinguished generations of codesign. These are summarized in Table 1.

Finally, it is also very fortunate to see from a researcher's point of view that many open and fundamental questions will definitely appear and that these will stimulate and keep our lives busy, hopefully for the next 100 years.[4] ■

Acknowledgment

The author would like to particularly thank the following companies that stimulated the research on advances in new codesign technologies through funding and evaluation: IBM for applying codesign during specification and for performance analysis of chip designs and system architectures as well as during firmware and test software development; Siemens Industrial and Healthcare Sectors for design space exploration and system synthesis; Intel Mobile Communications (IMC) for applying ESL

[4]The following list of roughly 100 citations only represents a very small sample of existing relevant work on such a broad field of research as codesign. It is far from being complete in many areas, and several facets of this research discipline were only presented here in an introductory way.