A Survey of Co-Design Ideas and Methodologies (draft)

G. Bosman

February 25, 2003

Contents

Chapter 1

Introduction

Designing embedded systems becomes more and more complex due to the increasing size of integrated circuits, increasing software complexity and decreasing time-to-market requirements and product costs. This ongoing increasing complexity of embedded systems motivates the search for a high-level system design approach at Chess. A high-level system design notation could also offer substantial performance benefits as there can be a smarter partitioning of the system's parts onto hardware and software. High-level system design is currently a major academic research area world-wide and it is strongly rooted in traditional areas such as control theory and digital design theory. For Chess such a design approach could mean a faster and better design process. This paper investigates aspects of high-level system design and the current state of the art in existing tools and methods that allow high-level system design. It will provide input for the Chess design strategy for the following years.

A system design notation should allow reasoning at a high level of abstraction. However, to be useful it must also be possible to synthesis hardware and software. Questions like how to divide the functionality of the system into hardware and software are a very important aspect of this synthesis. Traditionally the choice what to implement in hardware and what to implement in software is made early in the design process. Both parts are then made in separate tracks by separate design teams.

When the emphasis lies on the hardware-software partitioning problem, system-level design methods are also called "Co-Design" methods. These terms are often used mixed; in this paper *Co-Design* will be mainly used.

The goal of Co-Design is to explore the whole design-space to be able to make well-informed critical design decisions. This would lead to a more optimal partitioning and more flexibility in the process from design to implementation. To really understand the improvement a high-level system design paradigm makes it is important to see how traditional system design works and what that approach lacks.

1.1   Traditional design

1.1.1   Microprocessors

Traditionally complex embedded systems are designed around a microprocessor in a Von Neumann architecture. A Von Neumann system is fundamentally a sequential system. There is heavy optimalisation inside the CPU itself (for instance using pipelining) but ultimately the commands are executed one at a time. Systems based on microprocessors have several benefits. Microprocessors are very well optimized and they allow design of families of products that can be built. Such a family of products can provide various feature sets at a different price point and can be extended to keep up with rapidly changing markets[43].

1.1.2   Von Neumann bottleneck

However, the fact that each command is executed sequentially leads to a fundamental limitation. Backus calls this the "Von Neumann bottleneck"[2]. He points out that this bottleneck is not only a physical limitation, but has also been an "intellectual bottleneck" in limiting the way we think about computation and how to program.

1.1.3   Paradigm shift

[Guus: explain difference between hardware and software as related to concurrency.] This same aspect is a major motivation for Chess to do this investigation: how to prevent the 'paradigm-shift' that often occurs in designing systems. [Guus: this paragraph should be better. Explain more what the paradigm shift is.]

1.2   Other types of hardware

1.2.1   Application specific integrated circuits

Not all embedded systems are designed around microprocessors. It is possible to design embedded chips that compute in a parallel way. Application Specific Integrated Circuits (ASICs) are specialized chips that are used for example to implement encryption algorithms. They are similar to processors in the sense that they are also 'hardwired' solutions. It is very expensive to design an ASIC, and the design is a time-consuming process. Therefore, customizing an ASIC for a single application is only feasible when the project is reasonably large. In any case an ASIC can only be produced when the layout is final: it is not possible to experiment with the layout and try several versions.

Although you'll loose the optimalizations found in microprocessors there is a huge potential gain when designing hardware that is parallel in nature because the layout of the hardware can then be tailored exactly to the functional requirements. This can be extremely profitable, especially when the problem to be solved is mainly parallel in character. ASICs are therefore often used in areas such as compression or encryption which are parallel 'in nature'.

An important motivation for the rising interest in Co-Design is the introduction of another type of chip that is much more flexible than ASICs: programmable logic devices.

1.2.2   Programmable logic devices

Programmable Logic Devices (PLDs) are computer chips that can be programmed to implement circuits requiring both combinational and sequential logic. Reconfigurable logic devices are a class of programmable logic devices that may be reprogrammed as often as desired. A type of reconfigurable logic devices that are becoming very popular[10] are Field Programmable Gate Arrays (FPGAs). Three direct benefits of the reconfigurable approach can be recognized: specialization, reconfigurability and parallelism[36] [Guus: use more of Tessier on future developments [36]].

1.2.3   New design strategy

FPGA's shorten the development cycle dramatically and are much cheaper to use than ASICs. This allowed research in Co-Design to increase a lot. It also made researchers consider more fine-grained approaches.

Early approaches in Co-Design started with components of high granularity, such as ASICs and microprocessors. Today parts of the hardware are designed from scratch on a FPGA in combination with microprocessor and ASICs. This allows the system to benefit from both the microprocessor's specialization and the parallelism offered by the FPGA and ASICs. This difference in granularity is an important feature to discriminate on various Co-Design approaches.

Obviously designing hardware and software for a system in an integral manner is an extremely complex task with many aspects. There is a wide range of architectures and design methods, so the hardware/software Co-Design problem is treated in many different ways.

1.3   Assignment

"To survey development methods and to investigate how an integrated approach of hardware-software design can improve system development at Chess."

The focus in this paper will be on the shift of paradigms when traversing through the various levels of detail.

1.4   Outline of this thesis

In Chapter 3 we'll look into the field of Co-Design and the problems it tries to solve. In Chapter 3.2 different types of Co-Design approaches are described, and the classifications that can be made. It turns out that the paradigms, the computational models, are very important to understand the design strategy to use and they are directly related to the paradigm shift. Therefore Chapter 4 describes computational models that are commonly used to model (parts of) systems.

A single paradigm approach has serious disadvantages so various models that allows a mixture of 2 or more computational models have been proposed

in literature. In Chapter 5 the various aspects of system-level modelling will be discussed. This Chapter will serve as a guide for Chess in assessment of future developments and it Chapter 6 a number of existing projects and models will be described and analyzed using the classifications and ideas found in the first part.

Chapter 2

Background on Co-Design

This chapter describes the field of Co-Design.

2.1   History of Co-Design research

The field of Co-Design is about 10 years old now. [Guus: about Gupta paper etc]

   In 1998 a paper was published that described a method to synthesize code for both hardware and software, for a specific type of data-flow programs[12].

2.2   Related work

- The SAVE project of the Linköping University in Sweden did a survey on Co-Design representation models in 1999[8].

- At the Leiden University in the Netherlands a survey was done comparing 8 tools and their underlying methodologies[42].

- O'Nils presented ComSys, an approach to the generation of interfaces between application software and hardware IP components. In his PhD thesis from 1999 he reviewed several Co-Design methods[31].

- Some authors made an broad overview of various Co-Design development methods, e.g. [44], [16].

- Edward Lee wrote a very readable introduction to the field of Embedded Software and various computational models[22] and how it is different from traditional computer science areas.

2.3   Related approaches

A number of areas that are related to Co-Design have been researched.

2.3.1   Software in the loop

Some of the issues Co-Design tries to solve are also handled by 'Software-In-The-Loop'. This is developing software in a virtual hardware environment. Although this easies the design of software for hardware it does not allow the full improvements made possible by Co-Design because the partitioning between hardware and software is not flexible. Within Chess the SHAM has been developed. This is a device (a printed circuit board) that allows software testing for onboard software[39].

2.3.2   Reconfigurable hardware

Reconfigurable systems exploit FPGA technology, so that they can be personalized after manufacturing to fit a specific application.

"A promise of reconfigurable hardware is that it should allow the logic and memory resources in a chip to be used more efficiently, especially in applications that need massive computing power. But there is a further commercial advantage. It could turn finished products into a source of service revenue. Imagine a music player that includes programmable logic. When a new music-compression format emerges to replace MP3, owners of the player could download, for a fee, a new decompression algorithm for their player from the maker's website"[37].

[Guus: Add reference to Viales or ITA project here (reconfigurable parts), or SHAM.]

The operation of reconfigurable systems can either involve a configuration phase followed by an execution phase or have concurrent (partial) configuration and execution.[27]. The major problem in this type of systems consists of identifying the critical segments of the software programs and compiling them efficiently to run on the programmable hardware. This is a different field and will not be treated in this thesis.

2.3.3   Adaptive computing

The field of adaptive computation is closely related to reconfigurable hardware. According to Neema the primary challenge of the Adaptive Computing approach is in system design[29].

2.3.4   Hybrid systems

Most traditional Co-Design methods explore ways of modelling digital systems. Embedded systems however, often interact with an analog environment. Traditionally, this is the domain of control theory and defined in the digital domain. Because of the way models are often treated (digital) the analog environment is often translated to a digital representation by computer scientists. This translation is not trivial but must be done very carefully otherwise it's not possible to formally verify guarantee safety and/or performance of the embedded device.

To address this issue *hybrid* embedded system models have been designed[1, 35]. The issues that Co-Design faces, such as combining various models of computations and making sure properties are valid throughout the whole design phase, can of course also be found in this hybrid system modelling.

Chapter 3

Co-Design

Co-Design is "A design methodology supporting the concurrent development of hardware and software in order to achieve system functionality and performance goals. In particular, Co-Design often refers to design activities prior to the partitioning into Hardware and Software and the activity of design partitioning itself."[41]

The Co-Design process for embedded systems includes specification, modelling, validation, and implementation[27]. To comprehend the benefits of various Co-Design technologies it is important to understand how the design process works. This process is the subject of this chapter.

3.1 Specification and modelling

Modelling is the process of conceptualizing and refining the specification. A specification undergoes a manual synthesis process that generates a model of an implementation. The result of the modelling phase is a model, which is specified in an Internal Design Representation (IDR)[16].

That model may contain multiple models of computation and there is a trade-off between scalability and expressiveness in this IDR[40].

Modelling in the context of Co-Design is sometimes called *cospecification*.

3.2 Modelling approaches

Hardware/software Co-Design of embedded systems can be differentiated in several ways. This thesis will distinguish approaches in the difference between the internal design representation. If there is a single IDR that is used throughout the development process, we'll call it a *homogeneous modelling*. Otherwise, when multiple IDRs are we'll call the approach *heterogeneous*.

Another possibility is to consider the number of system-level specification language. Mooney et. al. make this distinction: if a method has only a single specification language they'll call it a homogeneous approach, otherwise a heterogeneous approach [28]. I believe this is a bit to superficial, as very often multiple specification languages can be used in the beginning of the specification process that are then mapped in one Internal Design Representation.

3.2.1 Modelling using a single IDR

Co-design starts with a global specification given in either a single language or in multiple languages that are converted into one IDR. This IDR should be independent of the future implementation and of the partitioning of the system into hardware and software parts. In this case Co-Design includes a partitioning step aimed to split this model into hardware and software. The outcome is an architecture made of hardware processors and software processors. This is generally called a virtual prototype and may be given in a single language or different languages[16].

This approaches is called compositional by Mooney, Coste[9]. It aims at integrating the partial specification of sub-systems into a unified representation which is used for the verification and design of the global behavior.

Many researchers are doubting whether a grand unified approach will work. Specifically the group of Edward A. Lee (who created the Ptolemy system) has doubt about this[6]. He calls modelling using a single IDR the 'grand unified method' [21]. In order to be sufficiently rich to encompass the varied computational models of the competing approaches. They become unwieldy, too complex for formal analysis and high quality synthesis. Lee sees it as a big problem that a homogenous approach is based on a single IDR. This IDR would impose a model of computation which might be good for a subset of systems but bad for others[21].

Examples of homogeneous methods are Polis, COSYMA and SpecC.

3.2.2 Modelling using multiple IDRs

Lee states that generality can be achieved through heterogeneity, where explicit more than one model of computation is used. Coste calls this the co-simulation based approach[9].

Heterogenous modelling allows the use of separate languages for the hardware and software parts. The Co-Design starts with a virtual prototype where the hardware/software partitioning is already made for the biggest part. This a fundamental difference with the homogeneous approach. Here the emphasis is on the integral designing of the parts to make sure that the overall system has the required properties. The key issues are validation and interfacing [16]. A lot of research is done on the integration of different system parts that enables system optimization across language boundaries.

The co-simulation-based approach consists in interconnecting the design environments associated to each of the partial specifications. Like its name suggests, with co-simulation the software processors and the hardware processors of an system and their interactions are simulated in one simulation. It does not provide such a deep integration as compositioning does [stelling, leg uit]. However it does allow for modular design. Communication is between the processors would typically be implented using a cosimulation bus which is in charge of transferring data between the different simulators. [Guus: include remark busses].

Sometimes cosimulation is used to simulate the behavior of a system consisting of 2 models: the hardware and the software, and sometime it is used to model on a more abstract level where the hardware vs software decision has not been made yet. [Hmm. Strijdig met hierboven!] A typical example of such a

co-simulation approach is Ptolemy II.

3.3 Validation

The validation process a design model should give the designer a reasonable level of confidence about how much of the original embedded system design will be in fact be reflected in the final implementation[40]. There has been a lot of research in the simulation of heterogeneous hardware/software systems [6, 20, 40]. There are 3 three methods for validation:

1. Simulation

2. Prototyping

3. Formal Verification

Formal verification allows for a more thorough evaluation of the embedded system behavior (maximum behavioral coverage) by means of logics.

It is good to note that in simulation the hardware is often simulated (although often not real-time, as it's just a simulation). However there has also been some research in replacing the hardware simulator with an FPGA (or multiple FPGAs) that simulate the real target hardware[20].

3.4 Implementation

The model found throughout the modelling phase has to be implemented into hardware and software. It is an important notion in a Co-Design approach that there must be no continuity problem. That is: the steps from model to the synthesis should be all in the design process[34].

In the implementation phase architectural information is taken into account. Varea[40] calls this a merger between the IDR and the *technology library*. It is important that the intermediate IDR or specification is not too much influenced by the current target technology. Steps in the implementation phase include:

1. Hardware/Software partitioning

2. Synthesis of the code for software and hardware

3.4.1 Hardware/software partitioning

'The partition of a system into hardware and software is of critical importance because it has a huge impact on the unit costs and performance characteristics of the final design. In the case of embedded systems, a hardware/software partition represents a physical partition of system functionality into application-specific hardware and software executing on one (or more) processor(s).'[27].

Obviously is important to look at the architectural organization of the system. Although it is possible to generate a complete system using only FPGA's, it is very common to use a combination of 1 (or more) processors with dedicated hardware or with flexible hardware (FPGAs). This is called *coprocessing*[27]

On the lowest level, FPGA's can be used to implement SM's, datapaths and nearly any digital circuit. The outcome of the synthesis process is a final implementation of the embedded system.

3.4.2 Conclusion

The discussion of homogeneous vs. heterogeneous modelling and about IDRs is related to the fact that a particular IDR has an area of application to which it fits best.

The following chapter gives a rationale about this phenomena.

Chapter 4

Computational Models

Modelling is at the heart of all development methods. All Co-Design systems are based on a computational model, or combine a few of them. The computational models can be found in the Internal Representation Language.

A computational model is a formal, abstract definition of a computer. It describes the components in a system and how they communicate and execute[25].

4.1 Properties of computational models

There are three characteric properties of computational models important for system design:

1. Modelling of time

2. Communication mechanism

3. Main application

4.1.1 Modelling of time

An essential difference between concurrent models of computation is their modelling of time. [21] (page 11). Lui[24] states that the different notions of time make programming of embedded systems significantly different from programming in desktop, enterprize or Internet applications. Lee[23] proposed a mathematical framework to compare certain properties of models of computation. This allows for a precise definition of the various computational models. Time modelling alternatives are:

- continuous time, real numbers are taken as time-axes (see 4.3.2)

- discrete time, there's a global clocktick

- partial time, events are ordered

- No explicit notion of time

[Guus: explain about global and local notion of time.]

4.1.2   Communication mechanism

[TBD].

4.1.3   Main application

Fundamental to embedded software is the notion of concurrency. There is a lot of research done on compiling concurrent languages into sequential code that can be run on a microprocessor, see for example [17]. For this thesis however it is more interesting to see what happens when this paradigm-shift does not have to be made.

There are models that are designed to describe dataflow oriented systems (ie DSPs) and there are models more suitable for control-flow systems.However this approach lacks generality as most systems are not easily put in either one category. It should be noted that the difference between a control-flow or data-flow oriented computational model is important for both control and data-oriented systems. Many control-systems use complex sensors or subsystems such as image processing algorithms that are best specified using a type of data-oriented computational model[25].

4.2   Example: internal combustion engine

A good illustration of the need for multiple computational models can be found in [24]:
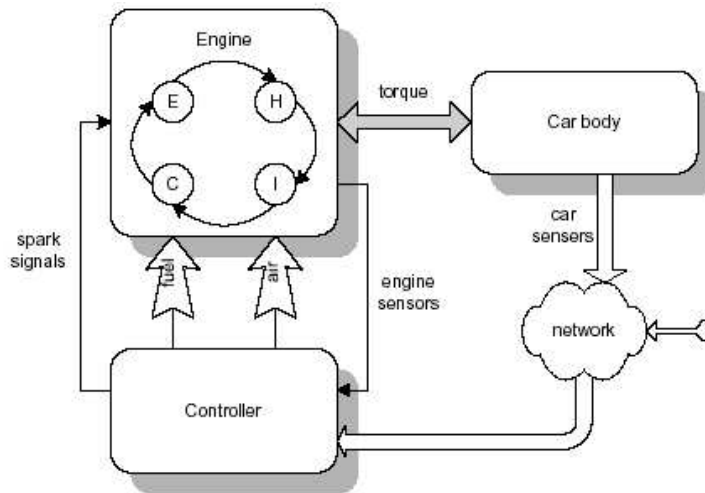
Figure 4.1: An internal combustion engine is made up of several parts that should be modelled by different MoCs.

A cylinder of an internal combustion engine has four working phases: intake, compress, explode, and exhaust. The engine generates torque that drives the power train and the car body. Depending on the car body dynamics, the fuel and air supply, and the spark signal timing, the engine works at different speeds, and thus makes phase transitions at various time transitions. The job of the

engine controller is to control the fuel and air supplies as well as the spark signal timing, corresponding to the drivers demand and available sensor information from the engine and the car body.

The engine and the car body in this example are mechanical systems, which are naturally modelled using differential equations. The four phases of the engine can be modelled as a finite state machine, with a more detailed continuous dynamics for the engine in each of the phases. While all the mechanical parts interact in a continuous-time style, the embedded controller,. which may be implemented by some hardware and software, works discretely.

Additionally, sensor information and driver's demands may arrive through some kind of network. The controller receives this information, computers the control law, controls the air and fuel values, and produces spark signals, discretely. So, we want to use a model that is suitable for handling discrete events for the network and the controller.

In this very common example, we have seen both continuous-time models and several discrete models: finite state-machines, discrete events, and real-time scheduling.

4.3 Common computational models

4.3.1 Finite-state machines

The finite-state machine model has been widely used in control theory and is the foundation for the development of several models for control-dominated embedded systems. The classical Finite State Machine (FSM) consists of a set of states, a set of inputs, a set of outputs, a function which defines the outputs in terms of input and states and a next-state function.[8].

Time	*time*
Com.	*comm*
Main app.	*control*

FSMs model systems where the system at any given point in time can exist in one of finitely many unique states. This makes them excellent for control logic in embedded systems. They can very well be formally analyzed and it is relatively straightforward to synthesis code from this model[21].

FSM can be visualized very well using a state transition graph (see Figure 4.3.1). FSM have a number of weaknesses. They are not very expressive, and the

Figure 4.2: A state transition graph.
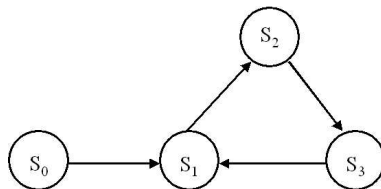
number of states can get very large even in the face of only modest complexity. Is intended for control-oriented systems with relatively low algorithmic complexity. A number of variations has been proposed to overcome to weaknesses of the classical FSM model. Using FSMs in a hierarchical model was first made popular by Harel[25]. He proposed StateCharts, which combine hierarchical FSMs and
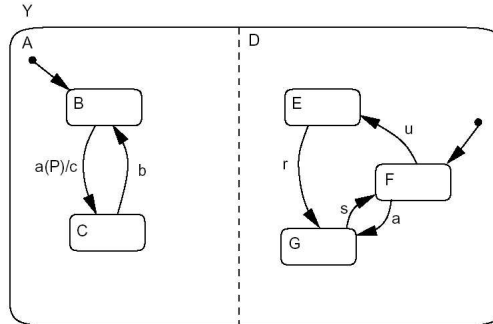
Figure 4.3: Hierarchical concurrent states[14].

concurrency. Statecharts are essentially a combination of FSMs with a SR. The tools Statemate from Ilogix uses statecharts as its control specification model.

In cases when a FSM must represent integer or floating-point numbers, a state explosion problem could be encountered. If each possible value for a number requires a state, the FSM could require a enormous amount of states. This can be solved by introducing a data-path to the FSM. This solution was described by Gaiski[14]; it could be used for computation-dominated systems.

Hierarchical concurrent finite-state machine

Like the FSM, the HCFSM models consists of a set of states and transitions. However, each state of the HCFSM can be further decomposed into aa set of substates, thus modelling hierarchy. Each state can also be decomposed into concurrent substates which execute in paralelel and communicate through global variables. An example can be seen in picture 4.3.1. State Y is decomposed in two concurrent states, A and D. The bold dots in the figure show when the starting point of the states are.

4.3.2 Continuous Time

Time	Continuous, global
Com.	-
Main app.	Mechanical parts

As we saw above a computational model describes components and their interactions. Embedded systems often contain components that can best be described using differential equations. Differential equations describe the rate at which variables change. They are often used to model (electro-)mechanical dynamics, analog circuits, chemical processes and many other physical systems[13].

In our car controller example of Chapter 4.2 the mechanical behavior of the engine and the car body are good candidates of components that can well be described using differential equations. Mechanical systems are often described using large number of difference equations, that can be hard to understand. In [30] a relatively easy example is given. In Picture 4.3.2 the chassis of a car is depicted and how it's connected to the road with a spring and damper. The interaction of these 3 physical forces (gravity, the spring and the damper) lead
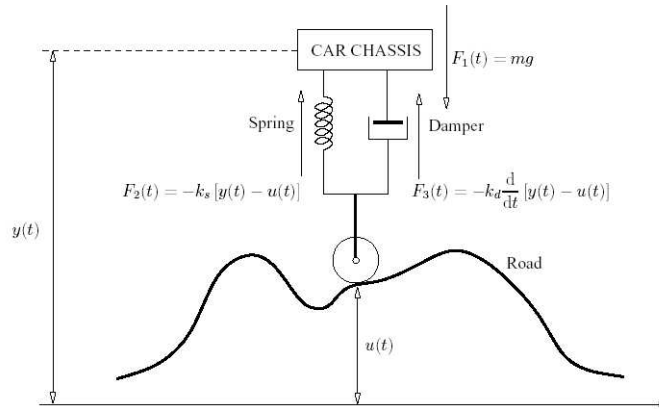
Figure 4.4: Forces on a car shock absorber that result in a differential equation relationship between road height signal $u(t)$ and the car height signal $y(t)$ [30].

to a differential equation (Equation 4.1).

$$d2dt2y(t) + kdmddty(t) + ksmy(t) = g + kdmddtu(t) + ksmu(t) \qquad (4.1)$$

This differential equation gives a relationship between the road-height and the car-height. For an embedded car-control system this can be important information; for example to slow down the car when the shocks are getting bigger.

Fundamental to this computational model is that it uses real numbers as time model – that's why this model is called continuous time. Formally said: "continuous-time systems are active over the entire time axis processing their input and producing output".[35]. The 'execution' of a continuous time models mean that a so-called *fixed point* must be found by the execution environment. This means that a set of functions of time must be found that satisfy all the relations. Hybrid systems (see Chapter 2.3.4) specifically deal with the integration of these type of Computational Models with others.

4.3.3 Discrete Time

Difference Equations are a discrete counterpart of differential equations. Where the latter works in the continuous time; difference equations in discrete time. Discrete-time systems can only react to their input and produce new output at distinct, equidistant time instances [35]. Difference equations are often rearranged as a recursive formula so that a systems output can be computed from the input signal and past outputs. Difference equations are commonly used for *filters* (that manipulate sound). They can also been seen as discretized version of differential equations.
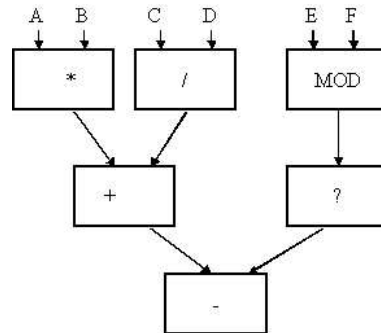
Figure 4.5: A dataflow graph for $(a * b) + (c/d) - \sqrt{(e \bmod f)}$.

4.3.4   Process networks

In a process network model of computation the arcs represent sequences of data values (token) and the bubbles represent functions that map input sequences into output sequences. Certain technical restriction are necessary to ensure determinacy.[21].

4.3.5   Data flow

It is a common representation formalism for modeling algorithms. The graph representation can be interpreted asynchronous (ADF) or synchronous (SDF). [Guus: uitwerken wat voor soort data-flow modellen er allemaal bestaan; split it into two?]

4.3.6   Discrete-event models

In a discrete-event system, modules react to event that occurs at a given time instant and produce other events either at the same time instant or at some future time instant. Execution is chronological[6]; time is an integral part of the model. Events will typically carry a time stamp, which is an indicator of the time at which the event occurs within the model. A simulator for Discrete-Event models will typically maintain a global event queue that sorts events by time stamp. This sorting can be computationally costly. [Guus: hard to simulate, nice in hardware].

4.3.7   Petri Nets

The Petri-Net is a state-oriented model. In the classical approach a Petri net is composed of 4 basic elements: a set of places, a set of transition, an input function that maps transitions to places, and an output function which is also a mapping from transition to places. This is an well-understood modelling tool.

A Petri net executes by mean of firing transitions. A transition can fire only if it is enabled – that is, if each of its input places has at least one token.

Two important features of Petri nets are its concurrency and asynchronous nature.[8], this can be modelled quite naturally as seen in Picture 4.3.7.
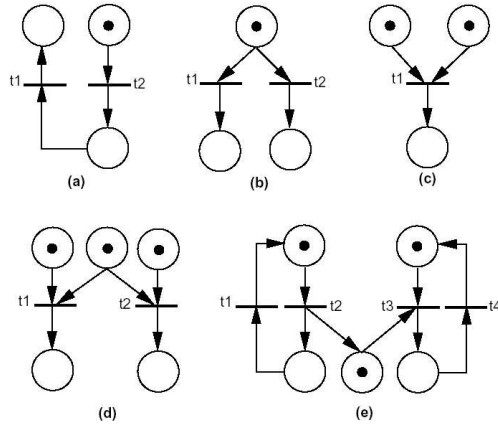
Figure 4.6: Petri net representing: (a) sequential order, (b) branching, (c) synchronization, (d) resource contention and (e) concurrency[14].

4.3.8 Synchronous models

In synchronous languages, modules simultaneously react to a set of input events and instantaneously produce output events. If cyclic dependencies are allowed, then execution involves finding a fixed point, or a consistent value for all events at a given time instant.[6]

Very often real-time systems are specified by means of concurrent processes, which communicate asynchronously [33].

The synchrony hypothesis forms the base for the family of synchronous languages. It assumes, that the outputs of a systems are synchronized with the system inputs, while the reaction of the system takes no observable time. So time is abstracted away. The synchrony hypothesis abstracts from physical time and serves as a base of a mathematical formalism. All synchronous languages are defined formally and system models are deterministic.

'In synchronous languages, every signal is conceptually (or explicitly) accompanied by a clock signal. The clock signal has meaning relative to other clock signals. It defines the global ordering or events. Thus, when comparing two signals, the associated clock signals indicate which events are simultaneous and which precede or follow others. A clock calculus allows a compiler to reason about these ordering relationships and to detect inconsistencies in the definition.'[6].

4.3.9 Rendezvous models

In a rendezvous model, the arcs represent sequences of atomic exchanges of data between sequential processes, and the bubbles presents the processes. [21]. Examples are Hoare's CSP and Milner's CCS. This model of computation has been realized in a number of concurrent languages, like Lotos and Occam. [Guus: based on algebra? How is timing handled?]

4.4 Comparison

Table 4.1: The main characteristics of the models of computation described in this Chapter.

Computational model	Main application	Timing	Modus
Continuous Time	Continuous control laws, analog circuits	?	?
Discrete Time	Periodically sampled data systems, cycle accurate modelling	Global notion of time. Every signal has a value at every clock tick[25]	?
Finite State Machine	Control		
Kahn Process Networks	Digital Signal Processing	No explicit timing	Asynchronous
SDF	Digital Signal Processing	No explicit timing	Synchronous
ADF	Digital Signal Processing	No explicit timing	Asynchronous
Discrete-event models	?	Globally sorted events with time tag	?
Petri Nets	?	No explicit timing. Just order of transitions[8]	Asynchronous
Synchronous/reactive models	Reactive real-time	No explicit timing	Synchronous
Rendez-vous	?	Atomic events along line of time[8]	Asynchronous

Three important characteristics have been described in Table 4.4. It shows that there are a number of computational models suitable for digital signal processing (the dataflow and process networks). Others are suitable mainly for control-oriented work. This indicates that for a real system various MoCs are necessary. Experience also suggest that several MoCs are required for the design of a complete system[6].

Because of the very different notions of time (from continuous time to no time-notion at all) in the various MoC's, integrating them is by no means trivial. This is one of the problems Co-Design approaches must solve before they're usable in system-design. In the next chapter various problems will be analyzed.

Chapter 5

Requirements for Co-Design methodologies

In this chapter requirement for usable Co-Design methods are discussed. [Guus: ik wil per paragraaf 1 of 2 'vragen' aan de een co-design methode kunnen stellen. Deze vragen komen in hoofdstuk 5.7 terug. Misschien is het een goed idee om daarnaast ook per paragraaf duidelijk de 'vragen' aan te geven, misschien in een klein kader in de marge ofzo.]

5.1   Paradigm shift

It has been recognized in literature that there is an important relationship between the model of computation and the target-architecture. Kienhuis et al.[19] speak in this context about a mapping between a model of computation and the architecture: "In mapping we say that a natural fit exist if the model of computation used to specify applications matches the model of architecture used to specify architectures and that the data types used in both models are similar."

This concept of a natural fit is the same concept of preventing a paradigm shift; when there is a natural fit between the IDR, the target architecture and the problem then there is no paradigm shift. This concept is thus recognized by the research community. However, a question they are still facing is: how to integrate these various models of computation. As we saw in Chapter 3 there are basically two approaches:

- Compositional

- Co-simulation based

The main difference in the two approaches in my view is that the co-simulation approach allows for more MoCs in a system and are better capable of working with optionally new developed MoCs. The other option promises a closer integration. This relates to the conventional wisdom that high performance while minimizing resources needed (or time needed) can be obtained by matching the architecture to the algorithm[29].

5.2   Design-space exploration

Many parts in the design of embedded systems require manual decisions. This remains so when using Co-Design methods. The integration Co-Design offers is valuable because of validation and easier synthesis of code from a model that allows hard- and software to be generated.

Because of the complexity of most systems, optimal manual decisions are sometimes not feasible. There are simply too many possibilities to consider. To use all of the potential improvements that a later HW/SW partition decision allows, it is therefore very important to reduce the user decisions as far as possible. This has been recognized and there are various methods for systematic design space exploration [5]. "In order to perform rigorous analysis and synthesis it is essential to prune the design space retaining only the most viable alternatives." In the past heuristics have been used to prune large design spaces. However, due to the complex behavior and interactions in multi-modal systems it is difficult to come up with effective heuristics. A better approach is to use constraints to explore and prune the design spaces; constraint satisfaction can eliminate the designs that do not meet the constraints. The pruned design space contains only the designs that are correct with respect to the applied constraints. These designs can then be simulated, synthesized and tested."[29].

5.3   Target architecture

When a Co-Design methodology allows for the generation of both software and hardware, it must also generate the communication mechanisms between these two parts. This include the operating system perhaps, and the device drivers of some sort.

O'Nils point out that very often off-the-shelf IP components are used in system design, and that often a major part of the work will be in interfacing these IP components. Tools like Polis are primarily designed for cases in which the whole design functionality is captured within the tools environment and communication refined during system synthesis. That is, the device drivers are generated together with the custom hardware and the operating system. However, if users want tot use IP blocks and off-the-shelf operating systems they will face the same problems that occur in manual design[31]. This has important implications for the commercial use of Co-Design methods and design space exploration tools: if they don't take into account off-the-shelf IP components they are not suitable for many types of projects.

An useful extension to most Co-Design approaches would be an explicit mapping between the IDR and the target architecture. Keutzer et al. state: "We actually believe that worrying about HW-SW boundaries without considering higher levels of abstraction is the wrong approach. HW/SW design and verification happens after some essential decisions have been already made, and this is what makes the verification and the synthesis problem hard. SW is really the form that a behavior is taking if it is "mapped" into a programmable microprocessor or DSP. [...] The origin of HW and SW is in behavior that the system must implement"[18]. If this mapping is recognized together and used in design-space exploration, a typical approach would be the Y-Chart[1] as proposed

---

[1]This term is rather unfortunate, as there exists another Y-Chart in Co-Design. That one
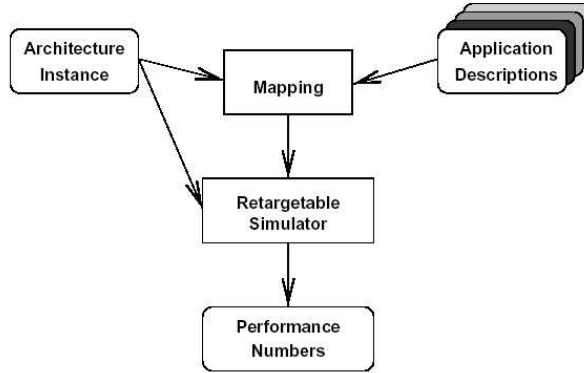
Figure 5.1: The Y-Chart approach[19].

by Kienhuis[19] (see Figure 5.3). [Guus: describe Y-Chart approach.]

5.4 Dealing with complexity

5.4.1 Abstraction

It is obvious that low-level languages such as VHDL are able to implement different models of computation. An imperative language can be used to implement for example a dataflow MoC[6].

However, lack of abstraction disqualifies low-level languages as candidates for modelling combined computational model-systems because it leaves programmers no freedom to make trade-offs between programmability, utilization of resources and silicon area. It's common for a specification language to allow more than 1 model of computation. However this does not always mean that this allows for suitable high-level mixture of 2 models.

A goal of all design methods is to allow systems to be designed on a higher level than the implementation level. The ultimate goal is to allow a very high-level design, that then automatically can be converted into the implementation level. This is a fundamental notion in computer science. Examples are programming languages, that allow humans to reason about variables or flow-of-control on a much higher level than machine code allows. There has also been a lot of research into finding languages to design hardware from a higher level. Nowadays it is very common to use a language as VHDL to define hardware. There are compilers available to generate netlists (hardware descriptions) from languages like VHDL. Although VHDL is probably considered by software engineers to be low-level, it is a major step forward compared to the arcane art of programming cells and gates directly.

This looking for a higher level of abstraction is an ongoing quest, and as so it can also be found in Co-Design methods. Ultimately the goal is to be able to design a system in a textual or graphical way, in such a manner that

---

relates to the level of abstraction of a system [Guus: a bit sharper!]

there will be an automatic compiler from this high-level representation into the implementation level: hardware, software or (often) a combination of both. Sometimes such an approach is called model compilation[34].

A higher level of abstraction in modelling decreases the gap between functional requirements of a system and the modelling process, leading to a better fit between these.

5.4.2   Hierarchy

'Brute-force composition of heterogeneous models may cause emergent behavior. Model behavior is emergent if it is caused by the interaction between characteristics of different formal models and was not intended by the model designer.'[13]

A common way to prevent unwanted emergent behavior is isolating various subcomponents and letting these subcomponents work together in a hierarchical way. Hierarchical in the sense of a containment relation, where an aggregation of components can be treated as a (composite) component at a higher level. In general, hierarchies help manage the complexity of a model by information hiding – to make the aggregation details invisible from the outside and thus a model can be more modularized and understandable[24].

"Note that in Ptolemy, models of computations are mixed hierarchically. This means that two MoC's do no interacts as peers. Instead, a foreign MoC may appear inside a process. In the old version of Ptolemy, such a process is called a wormhole. It encapsulates as subsystem specified using one MoC within a system specified using another. The wormhole must obey the semantics of the outer MoC at its boundaries and the semantics of the inner MoC internally. Information hiding insulates the outer MoC from the inner one." [6]. This approached of worm-holes was a bit biased towards data-flow computational models. In Ptolemy II it was replaced with opaque composite actors[11]. [Guus: explain the difference between composite actors and worm-holes!]

There are other ways to mix models of computation too. Statemate uses views. [Guus: are these views related to what Jantsch[15] calls analytical slicing into domains?]

5.5   Continuity gap

[Guus: to be done. Explain that not all tools support all phases of (co-) design process. I.e. some simulation only; others formal verification only.]

5.5.1   Partitioning

[Guus: On manual and automatic partitioning. Use [43].]

5.5.2   Implementation

A discrete-event model of computation is well suited for generating hardware. It is not very suitable to generate (sequential) software[6] (p. 131). This is for example why VHDL surprises the designer by taking so long. A model that

heavily uses entities communicating through signals will burden the discrete-event scheduler and bog down the simulation. Thus, a specification built on discrete-event semantics is a poor match for implementation is software.

By contrast, VHDL that is written as sequential code runs relatively quickly but may not translate well into hardware. The same goes for C: it runs very quickly and is well suited for software, but not for specifying hardware.

Dataflow and finite-state models of computation have been shown to be reasonably retargettable. Hierarchical FSMs such as Statecharts can be used effectively to design hardware or software. It has also be shown that a single dataflow specification can be partitioned for combined hardware and software implementation.'[6]

5.6 Varea's taxonomy

Many languages and tools that were developed based on a single model start to embrace other models [13]. The downside of such large languages that compose multiple MoC's in an ad hoc fashion is that formal analysis may become very difficult[6]. It compromises the ability to generate efficient implementations or simulations and makes it more difficult to ensure that a design is correct. It precludes such formal verification techniques as reachability analysis, safety analysis and liveness analysis.

This is not to be said that compositianal approach doesn't work. It does mean that a language should take the ability to integrate MoCs in it's design from the beginning. In this sense it's important to realize that it's not the language itself, but the IDR that a language will be converted in is fundamental. A lot of research goes into composite MoCs, suitable for both data- and controlflow. We saw a few examples of this already in Chapter **??**. Varea[40] proposes a classification for internal design representations according to the following taxonomy:

1. Models originally developed for control-dominated embedded systems and later expanded to include data-flow (these models will be called $\mathcal{M}_{CD}$).

2. Models developed in a data-dominated basis extended to support also control flow (referred to as $\mathcal{M}_{DC}$)

3. Unbiased model developed specifically to deal with combined control/data-flow interactions ($\mathcal{M}_{\bar{b}}$)

As noticed in Chapter 2.3.4 the modelling of hybrid digital-analog systems is a related field that is gaining more attention too. Also in this field there are existing tools that are extended with functionality to deal with hybrid modelling. An example of this is VHDL-AMS[7]. A good example is also SimuLink, a framework that is commonly used [24]. It's a modelling and simulation environment for continuous-time dynamic systems with discrete events that recently has been extended with statemachine modeling of discrete control[1].

5.7 Conclusion

In this chapter various requirements for modern system-level design tools have been discussed. The following points should be investigated for Co-Design ap-

proaches:

1. Integration of various MoCs (this relates to the paradigm shift)

2. What is the goal –the main application– of the approach

3. The abstraction level of the approach

4. How complexity of design is dealt with (support for hierarchical modelling of behavior)

5. Is there a 'gap' in the development process; what phases of the process are supported

6. If the approach allows synthesis of code is automatic partitioning supported

7. Does the approach allow IP re-use

8. What type of target-architectures are supported and

9. Is there a possibility of design-space exploration in the approach?

Chapter 6

Co-Design methodologies

In this Chapter we'll see a few of the most famous modelling methods and a few that have been selecting because they are special. [Yes, this should be a different sentence].

6.1 Ptolemy II

The Ptolemy project studies heterogeneous modelling, simulation and design of concurrent systems, where the focus is on embedded systems.[11].

The primary investigator of the Ptolemy project is Edward A. Lee. In 1991 he presented a paper that described the Ptolemy system[3]. This system has been in use for many years, and it's now succeeded by a new version, Ptolemy II.

The Ptolemy II software environment provides support for hierarchically combining a large variety of models of computation and allows hierarchical nesting of models[13]. It combines the wish for a homogeneous and thus predictable model with the desire to mix partial models of different kinds in a common heterogeneous model by hierarchically nesting sub-models of potentially different kinds.

A very good description of how this hierarchical mixed approach works in practice can be found in [25].

Ptolemy II is a component-based design methodology. The components in the model are called actors. A model is a hierarchial composition of actors. The atomic actors, such as A1, only appear at eh bottom of the hierarchy. Actors that contain other actors, such as A2, are composite. A composite actor can be contained by another composite actor.

Atomic actors contain basic computation, from as simple as an AND gate to more complex as an FFT. Through composition, actors that perform even more complex functions can be built. Actors have ports, which are their communication interfaces. For example in Figure 6.1, A5 receives data from input ports P3 and P4, performs its computation, and sends the result to output port P5. A port can be both an input and an output. Communication channels among actors are established by connecting ports.

The possibility to have various MoC's can be found in the *director*. A director controls the execution order of the actors in a composite, and mediates their
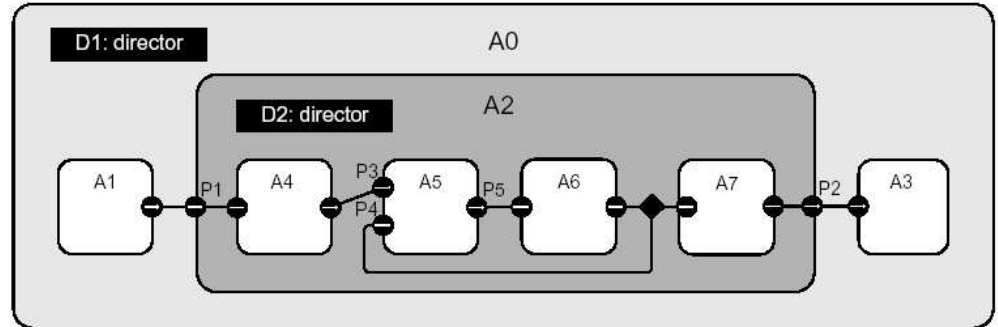
Figure 6.1: Example of a hierarchical specification of a systems using two (possibly different) Models of Computation. The directors controls the flow of control and data in such a MoC.

communication. In figure 1, D1 may choose to execute A1, A2 and A3 sequentially. Whenever A2 is executed, D2 takes over and executes A4-A7 accordingly. A director uses receivers to mediate actor communication. As shown in figure 2, one receiver is created for each communication channel; it is situated at the input ports, although this makes little difference. When the produces actors sends a piece of data (a token) to the output port, the receiver decides how the transaction is completed. Within a composite actor, the actors under the immediate control of a director interact homogeneously.

The focus in Ptolemy is on simulation.

6.2 COSYMA

An older design method is COSYMA, "CoSynthesis for Embedded Architectures". It was developed at the IDA, Germany. It covers the entire design flow from specification to synthesis. The target architecture consists of a standard RISC processor, a fast RAM for program and data with single clock cycle access time and an automatically generated application specific co-processors. Communication between processors and co-processor takes place through shared memory. The goal of Cosyma is basically speeding up existing programs by replacing parts in hardware[31][Guus: was this O'Nils?].

The system is designed in C^x. This is a C-extension with support for parallel processes and timing constraints. The C^x specification is then converted into an Extended Syntax Graph (ESG), the IDR. The ESG describes a sequence of declarations, definitions and statements and is overlayed [Guus: how?] with the Data Flow Graph (DFG) containing information about data dependencies. This shows the software-background of the tool, it is something like an extended compiler.

Research using COSYMA has been discontinued in 1999.

6.3 ForSyDe

An interesting method has been development by Sander and Jantsch[32, 33]. In their model events are totally-ordered by their tags. Every signal has the same set of tags. Events with the same tag are processed synchronously. There is a special value $\perp$ ("bottom") to indicate the absence of an event. These are necessary to establish a total ordering among events. A system is modelled by means of concurrent processes; it is a model based on the synchronous-assumption (see Chapter 4.3.8).

Lu[26] shows how to transform a system specification described in ForSyDe into its hardware and software counterparts. He does not provide a mixed implementation of HW and SW. [Guus: why not per module possible to make this decision?]

The hardware version of the Digital Equalizer that Lu makes is described using behavioral VHDL. The process are described using skeletons and these are then synthesized to VHDL code. The process described is manual. The Haskell code turns into behavioral VHDL quite easily. To generate (naturally sequential) C code an analysis phase is done to create a PASS.

6.4 Polis

The Polis research project started in 1988 by the UC Berkeley. It is a design environment for control-dominated embedded systems. It supports designers in the partitioning of a design and in the selection of a micro-controller and peripherals.

The system specification language is Esterel, but a graphical specification can also be given.

The generated software part consists of:

1. the application that has been modelled in CFSMs

2. a generated application-specific operating system for the selected processor

3. the I/O drivers

Hardware is synthesized as well. The Polis environment provides an interface for verification and simulation tools as well as an simulator.

A fundamental limitation of the Polis system is the MoC used, the Codesign Finite State Machine (CFSM). A CFSM is an extended finite state machine that communicates with other CFSMs asynchronously with unbounded delay and by means of events. The communication model between CFSMs is not efficient in representing systems with intensive data processing, since CFSMs communicate over channels with one-place buffers and have non-blocking write communication semantics. Therefore, a buffer is overwritten every time the sender emits an event before the receiver has consumed the previous event. This can be avoided either by means of scheduling constraints or with a blocking write protocol: however, both mechanisms often result in a loss of performance. This means that POLIS is mainly useful for control-dominated embedded systems. Although the POLIS method allows performance-estimation for the simple controller that is generated, estimation techniques for more complex processor models are lacking[42].

The Polis project has led to a set of commercial tools called Felix in 1998. It allows integration with Ptolemy, which makes it quite a powerful tool. [Guus: expand relationship Felix, Polis, VCC and Metropolis].

6.5   VULCAN

HardwareC. No behavioral hierarchy possible. Rather lowlevel.

6.6   Solar/Music

[9] describes a multi-language approach at the system level providing both system-level refinement and high-level interfaces synthesis.

6.7   SpecC

SpecC is a new language based on the C programming language.

6.8   SystemC

The SystemC language is a C++ language subset for specifying and simulating synchronous digital hardware. It's based on a class library of C++; it's not a new language by itself. It's a initiative by a group of vendors and embedded software companies to create a common (open source) standard.

In SystemC a complete system description consists of multiple concurrent processes. The system can be specified at various levels of abstraction (behavioral hierarchy).

6.8.1   SpecC vs. SystemC

SystemC and SpecC are two languages both coming from industry, only from different providers. Another difference is that SpecC works on a little higher abstraction level and it's process is more structurized. SystemC on the other hand is better suited towards RTL modelling of hardware design. It is possible and fruitful to mix the two approaches as Cai et. al. illustrated[4].

6.9   FunState/SPI Workbench

FunState is an internal design model based on functions driven by state machines[38]. It is an enhancement of the older SPI Workbench[45].

The SPI Workbench [45] is based on intervals of system properties and is specifically targeted to cosynthesis. Made for performance estimations. [Guus: Funstate = new version of SPI workbench?]

Table 6.1: The main characteristics of systems described in this Chapter.

Model	MoC based on	Main application	Origin	Target arch.	Specification languages
COSYMA					C^x
ForSyDe					
FunState	FSM with functions		$\mathcal{M}_{\bar{b}}$		
IRSYD	Flow Charts				
Polis	CFSM	Control-dominated	$\mathcal{M}_{CD}$		Esterel, graphical
Ptolemy II	Multiple MoC's				
SpecC					SpecC
SystemC					SystemC†
Vulcan	Flow...		$\mathcal{M}_{DC}$		HardwareC

†kVAd means distortion power.

6.10 Comparison

Some approaches don't try to incorporate many different models of computations. Polis for example is targetted towards control-oriented systems. It allows for a complete design process from high-level model to implementation. Because the computational model used in Polis is partly based on FSMs, the according state-space explosion causes the Polis approach to be only suitable for smaller systems.

Not all approaches described here are 'industry-ready'. There is a lot of research going on into new internal representation languages (FunState, ForSyDe). This shows that many researchers believe the current IDRs are not rich enough. A rough classification can be made of the direction research into IDRs is going:

- FSM based approaches (such as Polis)

- Petri-net based approaches

Not every approach that has been described support all the phases of Co-Design described in Chapter 3. In Table 6.10 the supported steps are indicated. Ptolemy II is a bit of an outsider here. The main focus of Ptolemy II is (co-)simulation and the other phases get less attention.

Two older approaches that do have a complete design process are Vulcan and Cosyma. They have a lower ambition though: the level of abstraction is rather low: they are both extensions of existing processes. Vulcan from a hardware side, Cosyma from a software side.

SystemC and Specc also have a lower level of abstraction than for example Polis offers. An important difference is that they do take IP integration into account.

[+automatic partioning? + IP reuse]

Table 6.2: The supported phases of the Co-Design development process.

Model	Simulation	Formal analysis and verification	Synthesis	Maturity	License
COSYMA					
ForSyDe					
FunState	FSM with functions				
Polis					
Ptolemy II	Yes	No[1]		Good	PD
SpecC					
SystemC					
Vulcan					

Chapter 7

Conclusions

A synergistic approach of hardware and software design and taking design to higher level are nowadays recognized as mandatory to keep up with the increasing complexity of embedded systems design. This observation made by Chess is shared by the whole industry. Academic work is done on this field too.

7.1 Internal design representation

Most Co-Design methodologies make use of an internal representation for the refinement of the input specification and architectures. Many such internal representations exist. They are all based on one or more Computational Models, and for separate parts of the system different MoC's are useful as we saw in Chapter 4.4. There is a 'natural mapping' between a MoC and certain system parts.

Experiments with system specification languages have shown that there is not a single universal specification language to support the whole design process for all kinds of applications. [16]. Some tools use 1, others use more than 1 internal design representation. There is a lot of research going into new (potential) IDRs.

7.2 Co-Design approaches

Ptolemy II seems to be a very mature tool, specifically the theoretical foundation of mixing various computational models is well thought-out. However, it is mainly focussed towards simulation.

[Guus: Which?!] The two issues found in this paper are closely related. If you want a single specification language you'll loose in the paradigm-shift. It is hard to imagine an (efficient) language that allows both control- and dataflow types to be presented and generate efficient code for it for all types of applications. On the other hand, if you don't mind taking the HW/SW decision earlier there are very good integrated tools and frameworks that allow working with both parts of your system in a systematic way. Code generation (or hardware generation) is easier in this style.

[19] also realizes this. He says: "the refinement approach has proven to be very effective for implementing a single algorithm into hardware. The approach

is, however, less effective for a set of applications. In general, the refinement approach lacks the ability to deal effectively with making trade-offs in favor of the set of applications."

There is also a mixed form possible. This mixed form would not be applicable for every type of system, but only for a subset. An example of such an approach is ForSyDe. They allow the specification of both control- and dataflow parts in a single language. They have shown to be able to generate (reasonably) efficient code.

7.3   Roadmap

Research in high-level system design will be concentrated in 4 directions:

- Special complete design flows for specific types of system (small control dominated: Polis, creation of DSPs ...)

- Research into internal design representations that allow a wider range of applications to be modelled later like above.

- Cosimulation based: allows for a wide range of applications, high level representation but does not offer code-generation.

- Platform based design, lower level representation, C-like languages. Allows code generation, integration of IP blocks.

Bibliography

[1] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1), January 2003.

[2] J. Backus. Can programming be liberated from the Von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[3] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: a mixed paradigm simulation/prototyping platform in C++. In *Conference Proceedings C++ At 163 Work*, 1991.

[4] L. Cai, D. D. Gajski, M. Olivarez, and P. Kritzingers. C/C++ based system design flow using specc, vcc and systemc. Technical report, University of California, Irvine, June 2002.

[5] A. Cassidy. High-level performance modelling and design exploration, –.

[6] W.-T. Chang, S. Ha, and E. A. Lee. Heterogeneous simulation – mixing discrete-event models with dataflow. *Journal of VLSI Signal Processing*, 15:127–144, 1997.

[7] E. Christen and K. Bakalar. VHDL 1076.1 - analog and mixed signal extensions to VHDL. In *Proceedings of the conference with EURO-VHDL'96 and exhibition on European Design Automation*, pages 556–561. IEEE Computer Society Press, 1996.

[8] L. A. Cortés, P. Eles, and Z. Peng. A survey on hardware/software codesign representation models. Technical report, Linköping University, June 1999.

[9] P. Coste, F. Hessel, and A. Jerraya. Multilanguage codesign using SDL and Matlab, 2000.

[10] S. Cotofana, S. Wong, and S. Vassiliadis. Embedded processors: Characteristics and trends. Technical report, Delft University of Technology, 2001.

[11] J. Davis II, C. Hylands, J. Janneck, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, S. Sachs, M. Stewart, K. Vissers, P. Whitaker, and Y. Xiong. Overview of the Ptolemy project. Technical report, University of California at Berkeley, Mar. 2001.

[12] M. Eisenring, J. Teich, and L. Thiele. Rapid prototyping of dataflow programs on hardware/software architectures. In *Proc. of HICSS-31, Proc. of the Hawai'i Int. Conf. on Syst. Sci.*, pages 187–196, Kona, Hawaii, January 1998.

[13] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. In *Proceedings of the IEEE*, 2002.

[14] D. Gajski, J. Zhu, and R. Dömer. Essential issues in codesign. Technical report, University of California, Irvine, 1997.

[15] A. Jantsch, S. Kumar, and A. Hemani. The Rugby meta-model. Technical report, Royal Institute of Technology, Mar. 2000.

[16] A. A. Jerraya, M. Romdhani, P. L. Marrec, F. Hessel, P. Coste, C. Valderrama, G. F. Marchioro, J. M. Daveau, and N.-E. Zergainoh. *Multilanguage Specification for System Design and Codesign*, chapter 5. Kluwer academic Publishers, 1999.

[17] Y. Jiang and R. K. Brayton. Software synthesis from synchronous specifications using logic simulation techniques. In *Proceedings of the 39th conference on Design automation*, pages 319–324. ACM Press, 2002.

[18] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. L. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform–based design. *IEEE Trans. on CAD*, 2000.

[19] B. Kienhuis, E. F. Deprettere, P. van der Wolf, and K. Vissers. A methodology to design programmable embedded systems — the Y-chart approach. *Lecture Notes in Computer Science*, 2268:18–??, 2002.

[20] Y. Kim, K. Kim, Y. Shin, T. Ahn, and K. Choi. An integrated cosimulation environment for heterogeneous systems prototyping. *Design Automation for Embedded Systems*, 3(2/3):163–186, Mar. 1998.

[21] E. A. Lee. System-level design methodology for embedded signal processors. Technical Report F33615-93-C-1317, University of California at Berkeley, 1997.

[22] E. A. Lee. Embedded software. Technical report, University of California at Berkeley, July 2001.

[23] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer Aided Design*, 17(12):1217–1229, Dec. 1998.

[24] J. Liu. *Responsible Frameworks for Heterogenous Modeling and Design of Embedded Systems*. PhD thesis, University of California at Berkeley, 2001.

[25] X. Liu, J. Liu, J. Eker, and E. A. Lee. Heterogeneous modeling and design of control systems. *Software-Enabled Control: Information Technology for Dynamical Systems*, 2002. To appear.

[26] Z. Lu. Refinement of a system specification for a digital equalizer into HW and SW implementations. January, Royal Institute of Technology, 2002.

[27] G. D. Micheli and R. K. Guipta. Hardware/software co-design. In *Proceedings of the IEEE*, volume 85, pages 349–365, Mar. 1997.

[28] V. J. Mooney III and G. De Micheli. Real time analysis and priority scheduler generation for hardware-software systems with a synthesized run-time system. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 605–612. IEEE Computer Society, 1997.

[29] S. Neema. System-level synthesis of adaptive computing systems, Mar. 2000.

[30] B. Ninness. *Fundamentals of Signals, Systems and Filtering.* –, 2002. To appear.

[31] M. O'Nils. *Specification, Synthesis and Validation of Hardware/Software Interfaces.* PhD thesis, Royal Institute of Technology, Sweden, 1999.

[32] I. Sander and A. Jantsch. System synthesis based on a formal computational model and skeletons. In *Proceedings of the IEEE Computer Society Annual Workshop on VLSI*, 1999.

[33] I. Sander and A. Jantsch. System synthesis utilizing a layered functional model. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES-99)*, pages 136–140. ACMPress, May 1999.

[34] S. Schulz and J. Rozenblit. Concepts for model compilation. *Proceedings of ICDA Conference*, 2000.

[35] T. M. Stauner. *Systematic Development of Hybrid Systems.* PhD thesis, Institut für Informatik der Technischen Universität München, 2001.

[36] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI Signal Processing*, 28(1):7–27, June 2001.

[37] The Economist. Bespoke chips for the common man. *The Economist*, Dec. 2002. 12th.

[38] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich. FunState - an internal design representation for codesign. In *ICCAD'99, the IEEE/ACM Int. Conf. on Computer-Aided Design*, pages 558–565, San Jose, U.S.A., Nov. 1999.

[39] J. van der Wateren and A. M. Bos. Real-time software testing throughout a projects life cycle using simulated hardware. In *Proceedings of the 5th International Workshop on Simulation for European Space Programmes*, Nov. 1998.

[40] M. Varea. Mixed control/data-flow representation for modelling and verification of embedded systems. Technical report, University of Southampton, Mar. 2002.

[41] Various. VSI alliance deliverables document. Technical report, VSI Alliance, 1999.

[42] V. D. Živković and P. Lieverse. An overview of methodologies and tools in the field of system-level design. *Lecture Notes in Computer Science*, 2268:74–??, 2002.

[43] W. Wolf. *Computers as components: principles of embedded computing system design.* Academic Press, 2001.

[44] T.-Y. Yen and W. Wolf. *Hardware-Software Co-Synthesis of Distributed Embedded Systems.* Kluwer Academic Publishers, 1996.

[45] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, and J. Teich. SPI – a system model for heterogeneously specified embedded systems. *IEEE Trans. on VLSI Systems*, 2002.

Appendix A

Vocabulary

ASIC	Application specific Integrated Circuit
FPGA	Field-Programmable Gate Array (a specific type of PLD)
FSM	Finite State Machine (see Chapter 4.3.1)
IDR	Internal Design Representation
IP	Intellectual Property. Used in the field of embedded systems to refer to existing modules (from other vendors) that can be used to build a system
MoC	Model of Computation, or computational model
PLD	Programmable Logic Device
VHDL	A language to describe layout and or behaviour of hardware. Comparable to a program-language for software