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

G. Bosman

July 21, 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. 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. Too often, the validity of this decision is not validated before system integration. This is far too late with respect to project risk.

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[50].

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

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. Some parts of an embedded system are easiest to describe using modelling languages that do not fit nicely onto existing hardware. Typical examples are data-filtering or physical movement modelling. A filter can be easily described in a drawing, and ultimately it is possible to imagine a specialized chip that implements this drawing immediately in gates. However, more commonly the algorithm will first be translated into C code, which will then be compiled into something to run on a CPU. The fundamental difference between these two approaches is their dealing with concurrency: if the drawing is implemented onto hardware directly, concurrent processing comes naturally. A translation to C code looses this advantage.

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[43] [Guus: use more of Tessier on future developments [43]].

1.2.3 New design strategy

FPGA's shorten the development cycle dramatically and are much cheaper to use than ASICs. Their capacity has grown to a such a size that they can handle real application for a fair price for prototype series. This allows research in Co-Design to increase a lot and make it more worthwhile.

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. A reason to (still) use ASICs and not to implement all the functionality in microprocessors and FGPAs is the power-consumption. The flexibility of the FPGA comes with the price that they are in general power inefficent compared to dedicated custom hardware [41]. 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

Chapter **??** gives an overview of related work. In Chapter 3 we'll look into the field of Co-Design and the problems it tries to solve. Different types of Co-Design approaches are described, and the classifications that can be made. It turns out that understanding the paradigms, the computational models, is very

important to decide on selecting a design strategy to use. Therefore Chapter 4 describes computational models that are commonly used to model (parts of) systems.

A single paradigm approach has serious disadvantages so various modelling approaches that allows a mixture of 2 or more computational models have been proposed in literature. To help investigating such approaches the various aspects of system-level modelling will be discussed in in Chapter 5. This Chapter will serve as a guide for Chess in assessment of future developments. In Chapter 6 a number of existing projects and models will be described and analyzed using the classifications and ideas found in the first part. This thesis ends with conclusions on the type of approaches there are, and which of these can be valuable for Chess.

Chapter 2

Background on Co-Design

This chapter describes the field of Co-Design.

2.1 Co-Design research

The field of Co-Design is about 10 years old now. Two papers that give a broad overview of the field are those by Gajski [15] and De Michelli[31].

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[49].
- 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[35].
- Some authors made an broad overview of various Co-Design development methods, e.g. [51], [18].
- Edward Lee wrote a very readable introduction to the field of Embedded Software and various computational models[25] 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[45].

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"[44].

Within Chess this opportunities have been recognized, and reconfigurable FPGA's are used in projects like ITA. This is a payment terminal, that can remotely get upgraded by downloading new software.

The operation of reconfigurable systems can either involve a configuration phase followed by an execution phase or have concurrent (partial) configuration and execution.[31]. 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[33].

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, 42]. 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."[47]

The Co-Design process for embedded systems includes specification, modelling, validation, and implementation[31]. 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)[18].

That model may contain multiple models of computation and there is a trade-off between scalability and expressiveness in this IDR[46].

Modelling in the context of Co-Design is sometimes called *cospecification*. In Figure 3.1 the design process has been depicted, which will be described in this Chapter.

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 [32]. I believe this is a bit to superficial, as very often

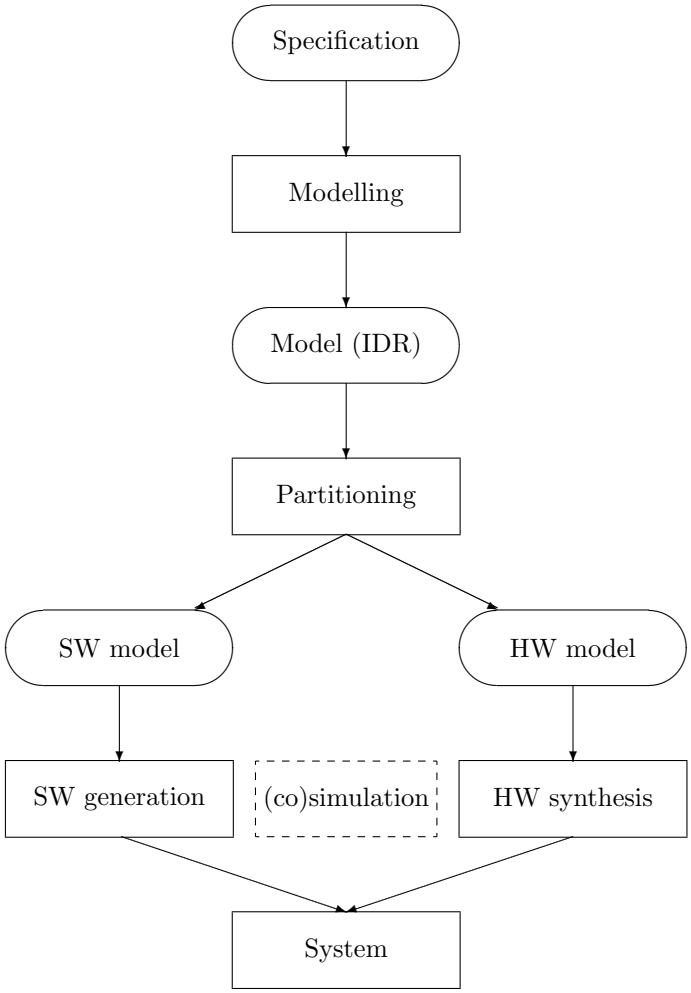

Figure 3.1: Co-Design design process

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[18].

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' [24]. 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[24].

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 [18]. 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 implemented using a cosimulation bus which is in charge of transferring data between the different simulators.

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[46]. There has been a lot of research in the simulation of heterogeneous hardware/software systems [6, 23, 46]. There are 3 three methods for validation:

1. Simulation
2. Prototyping
3. Formal Verification

Most design methods incorporate validation throughout the whole design process. Early in the design process simulation is mostly used. Some design methods support this explicitly. For example, some models are so detailed that they allow 'execution', on a very high level. More often, simulation is used after the partition step but before the hard- and software synthesis. In that context the term 'co-simulation' is often used.

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 that simulate the real target hardware[23].

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[40].

In the implementation phase architectural information is taken into account. Varea[46] 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

'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)'[31]. Partitioning is the a crucial step in Co-Design. The model must be split into parts: parts software, parts hardware. The most ideal would be to have *automatic partitioning*, in which tools will take the model and the target architecture into account, and make the split automatically. For this, it is important that the target architecture is known by the tools. In Chapter 5.3 there will be more about that.

The partitioning step is deliberately left vague here, as it differs strongly per design method. Often the partitioning step will be split into sub-steps, each dealing with a specific part of creating the split, such as scheduling and interface synthesis (see for example [15]. Other design methodologies don't have a single partitioning step. Instead before the final code and hardware synthesis, there is an iterative process of refining and transformation, such as in ForSyDe[39].

3.4.2 Synthesis

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. As an extra validation step the result of the implementation is often checked by creating a small amount of prototypes first.

3.4.3 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. In this Chapter common computational models are described and compared. In Chapter 4.2 a running example is introduced.

A computational model is a formal, abstract definition of a computer. It describes the components in a system and how they communicate and execute[28].

4.1 Properties of computational models

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

1. Modelling of time
2. Orientation
3. Main application

4.1.1 Modelling of time

An essential difference between concurrent models of computation is their modelling of time. [24] (page 11). Lui[27] states that the different notions of time make programming of embedded systems significantly different from programming in desktop, enterprize or Internet applications. Lee[26] 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 4.3.1)
- discrete time, there's a global clocktick
- partial ordered time, events are ordered
- No explicit notion of time

A very useful trick in dealing with time is assuming synchrenous operations.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.

'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].

Whether or not a language is synchronous has quite big implications. It is therefore also added in the matrix add the end of this Chapter.

4.1.2 Orientation

Gajski distinguises five categories of models[15]:

- State-oriented
- Activity-oriented
- Structural-oriented
- Data-oriented
- Heterogeneous

  Additionaly 'timed' models of computation can be recognized.

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 [19]. 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[28].

4.2 Example: internal combustion engine

A good illustration of the need for multiple computational models can be found in [27], where the working of an internal combustion engine is described using various models of computation:

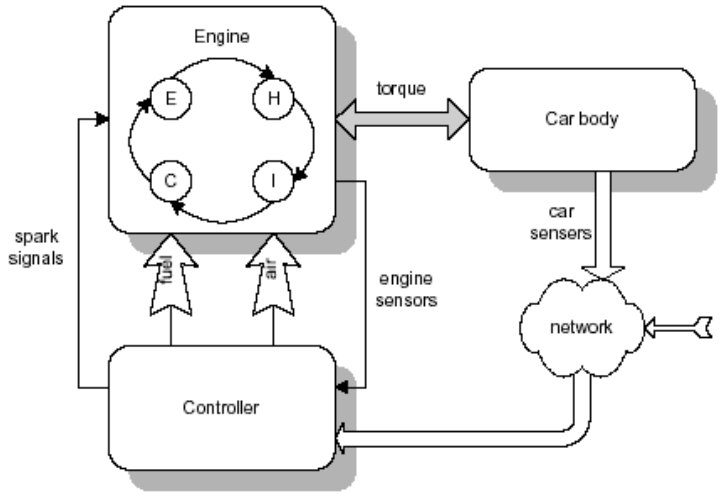

Figure 4.1: An internal combustion engine is made up of several parts that should be modelled by different MoCs. Adopted from Liu[27].

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. These models, and other, will be described in the rest of this Chapter.

4.3 Common computational models

Time	*Continuous, global*
Orientation	*Timed*
Synchronous?	-
Main app.	*Mechanical parts, analog circuits*

4.3.1 Continuous Time

As we saw above a computational model describes components and their interac-

tions. 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].

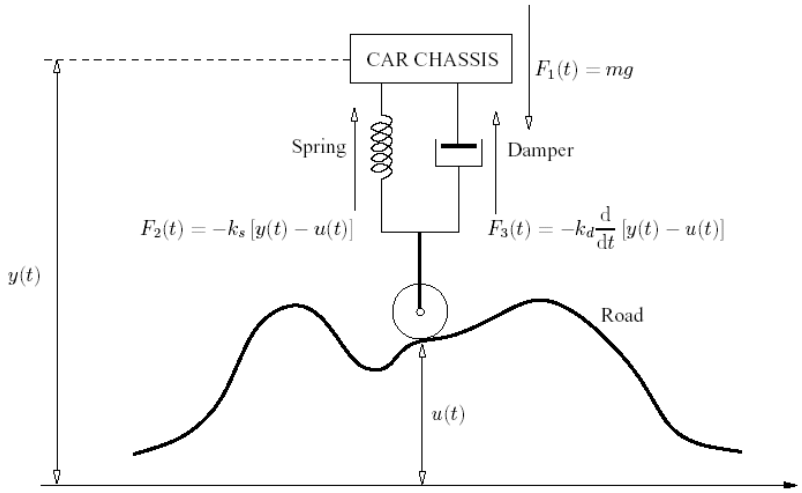

Figure 4.2: Forces on a car shock absorber that result in a differential equation relationship between road height signal $u(t)$ and the car height signal $y(t)$ [34].

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 [34] a relatively easy example is given. It shows how mechanical interactions quickly lead to differential equations (thus to a continuous model).

In Picture 4.3.1 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 to a differential equation (Equation 4.1).

$$\frac{d^2}{dt^2}y(t) + \frac{k_d}{m}\frac{d}{dt}y(t) + \frac{k_s}{m}y(t) = g + \frac{k_d}{m}\frac{d}{dt}u(t) + \frac{k_s}{m}u(t) \tag{4.1}$$

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"[42]. The 'execution' of a continuous time model 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.

4.3.2 Discrete Time

Time	*Discrete, global*
Orientation	*Timed*
Synchronous?	-
Main app.	*Filters, periodically sampled data*

Difference Equations are a discrete counterpart of differential equations. Where the latter works in the continuous time difference equations are discrete time based. Discrete-time systems can only react to their input and produce new output at distinct, equidistant time instances [42]. 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 modelling *filters* (that manipulate sound) and periodically sampled data streams. They can been seen as discretized version of differential equations.

4.3.3 Discrete-Event

Time	*Sorted events, global*
Orientation	*Timed*
Synchronous?	-
Main app.	*Digital circuits*

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].

4.3.4 (Co-Design) Finite-State Machines

Time	*Events with time-stamp*
Orientation	*State*
Synchronous?	*No*
Main app.	*Control-parts*

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].

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[24].

FSM can be visualized very well using a state transition graph (see Figure 4.3.4). FSM have a number of weaknesses. They are not very expressive, and the

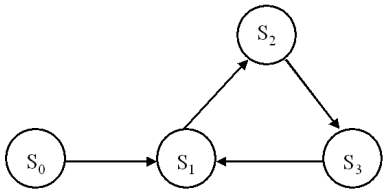

Figure 4.3: A state transition graph.

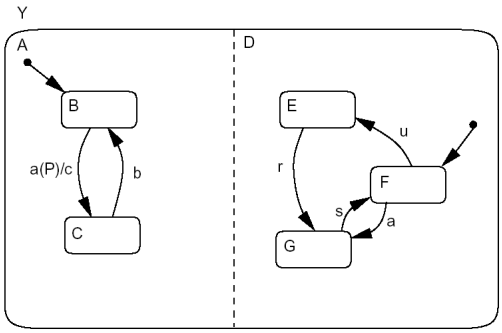

Figure 4.4: Hierarchical concurrent states[15].

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.

Extensions of the FSM

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[28]. He proposed StateCharts, which combine hierarchical FSMs and 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[15]; it could be used for computation-dominated systems.

An obvious extension to the FSM would be the addition of concurrency and hierarchy. Such a system is then called a Hierarchical Concurrent FSM (HCFSM). 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 a set of substates, thus modelling hierarchy. Each state can also be decomposed into concurrent substates which execute in parallel and communicate through global variables. An example can be seen in picture 4.3.4. 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.

Perhaps the most successful extension to the classical FSM is the Co-Design Finite State Machine (CSFM) which is used in Polis. In Chapter 6.4 we'll see more about this. Important in this extension is the hierarchy and concurrency that are added. The communication primitive in the CSFM system is the *event*. The behavior of the system is defined as sequences of events that can be observed when interact with the environment [8]. The notion of time in this model is thus: events with a time-stamp (ordered time).

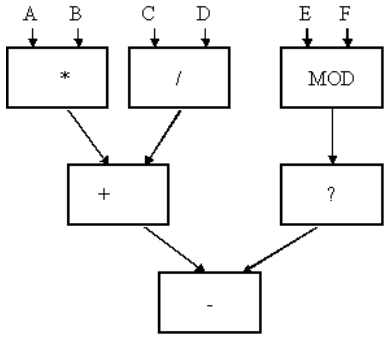

Figure 4.5: A dataflow graph for $(a * b) + (c/d) - \sqrt{(e \bmod f)}$.

4.3.5 Kahn Process Networks

Time	*No explicit timing*
Orientation	*Activity*
Synchronous?	*No*
Main app.	*Digital signal processing*

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.[24].

4.3.6 Asynchronous Data Flow

Time	*No explicit timing*
Orientation	*Activity*
Synchronous?	*No*
Main app.	*Digital signal processing*

It is a common representation formalism for modeling algorithms. The graph representation can be interpreted asynchronous (ADF) or synchronous (SDF). Special case of Kahn process networks.

4.3.7 Synchronous Data Flow

Time	*No explicit timing*
Orientation	*Activity*
Synchronous?	*Yes*
Main app.	*Digital signal processing*

4.3.8 Petri Nets

Time	*Order of transitions*
Orientation	*State or activity*
Synchronous?	*No*
Main app.	-

A Petri net is a well-known graphical and mathematical modeling tool. A Petri Net is a bipartite graph of transitions and places connected by directed edges. Each place has a number of tokens which are not ordered and at least in the basic Petri net model do not carry additional information.

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.

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. Transitions may fire if each of its input places contains at least as many tokens as there are edges from the input place to the transition. At firing, a transition removes one token via each incoming edge and produces a token via each outgoing edge.

Two important features of Petri nets are its concurrency and asynchronous nature.[8], this can be modelled quite naturally as seen in Picture 4.3.8.

It can be considered both activity- and state-based model, depending on the exact interpretation used in the modelling approach.

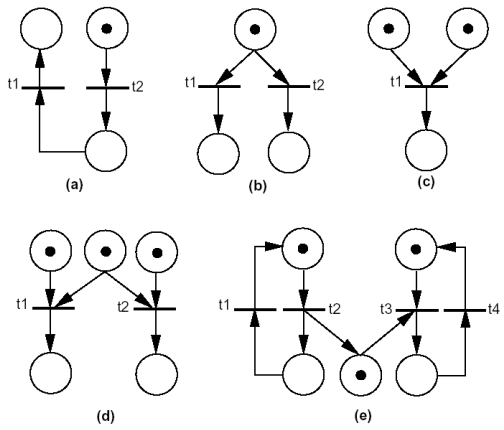

Figure 4.6: Petri net representing: (a) sequential order, (b) branching, (c) synchronization, (d) resource contention and (e) concurrency[15].

Time	*Atomic events on timeline*
Orientation	*State*
Synchronous?	*No*
Main app.	*Modelling resource management*

4.3.9 Rendez-vous

In a rendezvous model, the arcs represent sequences of atomic exchanges of data between sequential processes, and the bubbles presents the processes. [24]. 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.

An example of a design tool for embedded (control) systems based on this computational model is the CTJ (Java) library developed by Jovanovic et. al.[20].

Time	*No explicit timing*
Orientation	*Timed*
Synchronous?	*Yes*
Main app.	*Reactive real-time*

4.3.10 Synchronous/Reactive

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 [39].

4.3.11 Other Models of Computation

In this Chapter a number of computational models have been discussed. However, this list is not complete. There are many extension and variants of the computational models we've seen. Specifically Petri Nets are being researched a lot, such as Coloured Petri Nets and High Level Petri Nets. These can often be used well as Internal Design Representations. Varea proposed an IDR called Dual Flow Net (DFN), which provides tight control and data flow interaction[46]. Another mixed IDR is FunState, which are functions driven by state machines.

Another way that can be used to specify a model in a Co-Design method are imperative programming languages. The IDR would then be the syntax

graph of the language. This has the disadvantage that programming are too complex to use in formal verification, and that (imperative) programming languages do not explicitly model the system's state. The latter makes them harder to use in modeling embedded systems. Technically, such an IDR is often an 'Control/Data-Flow Graph', but in this paper it will be refered to a 'Syntax Graph'.

4.4 Comparison

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

Computational model	Time	Synchronous?	Orientation	Main application
Continuous Time	Continuous, global notion of time.	-	Timed	Continuous control laws, mechanical parts, analog circuits
Discrete Time	Global notion of time. Every signal has a value at every clock tick[28]	-	Timed	Periodically sampled data systems, cycle accurate modelling
Discrete-Event	Globally sorted events with time tag	-	Timed	Digital circuits
(Co-Design) Finite State Machine[a]	Events with time-stamp	no	State	Control-parts
Kahn Process Networks	No explicit timing	no	Activity	Digital Signal Processing
SDF	No explicit timing	yes	Activity	Digital Signal Processing
ADF	No explicit timing	no	Activity	Digital Signal Processing
Petri Nets	No explicit timing. Just order of transitions[8]	no	State- or activity	Scheduling, control, network protocols
Rendez-vous	Atomic events along line of time[8]	no	State	Modelling recource management problems
Synchronous/Reactive	No explicit timing	yes	Timed	Reactive real-time

[a]The Codesign Finite State Machine is chosen as a representative because the basic FSM is not sophisticated enough to be used as a Model of Computation (see 4.3.4).

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 (CSFM, synchronous/reactive). 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].

Next to the computational models discussed in this Chapter, there is a lot of research into new Internal Design Representations, such as Dual Flow Nets and FunState. These are still experimental, and there are no methodologies using these IDRs.

The formal computational models are not the only way a model can be specified; some design methods use the syntax graph of an imperative programming language as their IDR.

Because of the very different notions of time in the various MoC's, ranging from continuous time to no time-notion at all, 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 requirements for usable Co-Design methods are discussed. The result of this Chapter is a list of requirements and important aspects of Co-Design methodologies.

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.[22] 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 application's IDR and the target architecture 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:

How is integration of computational models dealt with?

- Using a single IDR (compositional approach)
- Using more than one IDR (co-simulation approach)

The main difference in the two approaches in my view is that the co-simulation approach allows for more MoCs in a methodology 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[33].

5.2 Origin of IDR

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 compositional 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 control-flow. We saw for example some extensions of the classic FSM in Chapter 4.3.4. Varea[46] 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 ($\mathcal{M}_{\overline{b}}$)

What is the origin of the IDR of the approach?

Such a classification can be useful to discriminate between various approaches.

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], or another example is SimuLink, a framework that is commonly used [27]. It's a modelling and simulation environment for continuous-time dynamic systems with discrete events that recently has been extended with statemachine modelling of discrete control[1].

5.3 Design-space exploration

Many parts in the design of embedded systems require manual decisions. Some Co-Design methods try to automate an important part of the design process: the decision what parts to implement in hardware, and what on software. This is called automatic partitioning.

Is automatic partitioning supported?

However, this is certainly not easy for a design methodology. In fact, most of them don't support it. Still, even with methodologies that don't have automatic partitioning, Co-Design is valuable. It makes validation and synthesis of code a lot easier, because of the shared model (IDR) that has been made.

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."[33].

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

5.4 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.

Does the approach allow IP reuse?

Not only the design process can be improved. Many suggestions brought up in this Chapter deal with the methodology itself. However, 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[35]. 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. So next to the design methodology and design process itself, there is also the matter of whether or not IP reuse is explicitly supported.

What type of target architectures are supported?

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"[21].

It is possible to generate a complete system using only FPGA's, but it's more common to use a combination of 1 (or more) processors with dedicated hardware (ASIC's) or with flexible hardware (FPGAs). This is combination of processor with extra hardware is called *coprocessing*[31], and the term *co-processor* is often used for the ASIC's.

The issue here is thus to create a useful *mapping* between an application and a target architecture. There is a design-space exploration method that is build on this premise, the Y-Chart approach[1] as proposed by Kienhuis[22]. This method follows a general scheme that can be visualized by an Y-shaped figure

[1]This term is rather unfortunate, as there exists another Y-Chart in Co-Design. The Gajski Y-Chart was introduced in 1983 for describing the taxonomy of electronic systems in 3 dimensions: behavioral, structural and physical.

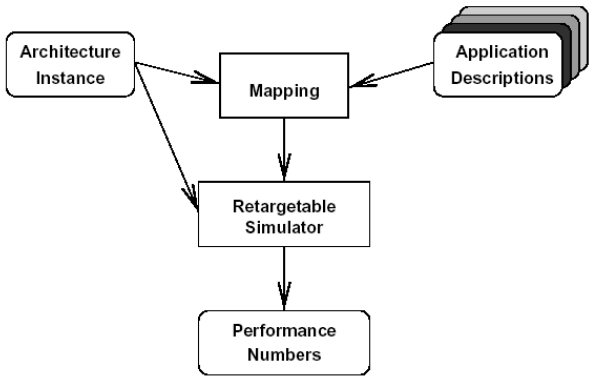

Figure 5.1: The Y-Chart approach[22].

(see Figure 5.4). In the upper right part in that Figure is the set of application descriptions that drives the design of the architecture. The effectiveness of the initial architecture is to be evaluated, using performance measurements. The various applications will be mapped on the architecture, and using quantitative numbers the performance will be determined. If this doesn't match the requirements, either the applications or architecture should be changed.

The power of this method is seen when these steps are supported by the development method. Therefore it is required to allow architecture to be defined within the same method, and to perform performance analysis. Newer Co-Design methods, such as Ptolemy II and SpecC have tools that help with design-space exploration.

5.5 Dealing with complexity

5.5.1 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 re-

search 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.

What is the abstraction level used in the methodology?

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 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[40].

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. There is of course no definition for "high level" or "lower level". Still, we'll try to get a view of a method by looking at the level of its specification language(s), and stay in line with Jerraya[18], who also used these words. In our result we'll make a difference in 3 levels: low, medium and high-level specification language.

5.5.2 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[27]. We're talking about behavioral hierarchy here – composition of child behaviors in time, as opposed to structural hierarchy.

"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 a 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[17] calls analytical slicing into domains?]

5.5.3 Implementation

A discrete-event model of computation is well suited for generating hardware. It is not very suitable to generate (sequential) software[6]. This is for example why VHDL simulation surprises the designer by taking so long. A model that heavily uses entities communicating through signals will burden the discrete-event scheduler and slow down the simulation. Thus, a specification built on discrete-event semantics is a poor match for implementation in 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 been shown that a single dataflow specification can be partitioned for combined hardware and software implementation.'[6]

What phases of the development process are supported in the approach?

[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.]

5.6 Conclusion

In this chapter various requirements for modern system-level design methods have been discussed. The following points should be covered by Co-Design approaches:

1. How is integration of various MoCs dealt with
2. What is the origin of the IDR of the approach
3. Is there a possibility of design-space exploration in the approach?
4. Does the approach allow IP re-use
5. What type of target-architectures are supported
6. The abstraction level of the approach
7. Is there a 'gap' in the development process; what phases of the process are supported
8. Is (automatic) partitioning supported (if the approach allows synthesis at all)

These aspects will be taking into account in the next Chapter, where a number of Co-Design methodologies will be analyzed.

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 [28].

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

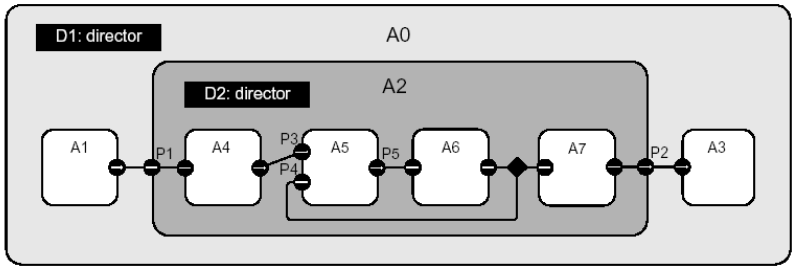

Figure 6.1: Example of a hierarchical specification of a systems using two (possibly different) Models of Computation. The directors controls the flow of control and data in such a MoC.

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.

Ptolemy II is a successor to Ptolemy Classic. It is written in Java, while the original version was made in C++.

The focus in Ptolemy is on simulation. However, Ptolemy is often used in combination with other tools, such as Polis (see Chapter 6.4)[30], to provide co-simulation functionalities. It is possible to use Ptolemy to generate software for specific processors (DSPs) too[37].

Ptolemy does not explicitly support the Y-chart approach[49]. However, Ptolemy is a very extensible system. There is an extension of the Ptolemy kernel in the direction of a Y-chart like tool for evaluation of architecture trade-offs, see [36].

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[35].

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. This is similar to a CDFG. The

ESG describes a sequence of declarations, definitions and statements and is overlayed 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.

"The chief advantage of this approach is the ability to utilize advanced software structures that result in enlarging the complexity of system designs. However, selective hardware extraction based on potential speedups makes this scheme relatively limited in exploiting potential use of hardware components. Further, the assumption that hardware and software components execute in an interleaved manner (and not concurrently) results in a system that under-utilizes its resources"[16].

Research using COSYMA has been discontinued in 1999.

6.3 ForSyDe

ForSyDe is a model based on the synchronous-assumption (see Chapter 4.3.10). It has been developed by Sander and Jantsch[38, 39].

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, which are Haskell functions or 'skeletons'.

Fundamental to the approach is the transformation process. The specification of the system is done in Haskell, and an iterative process of rewriting the specifications goes on until the requirements are met. ForSyDe does not allow for architectures do be described in the method, as such it does not support design space exploration directly.

Lu[29] 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.

IP reuse has not been specifically addressed.

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. Automatic partitioning is not supported in Polis.

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[49].

The Polis project has led to a set of commercial tools. For example, the Candence Virtual Component Co-Design toolkit (VCC) has been built op top of Polis. VCC allows better IP reuse than Polis, and follows the Y-chart approach[49]. Because the CSFM is not biased towards either hardware or software, it allows for a very late decision of what part to implement is software and what in hardware. This is also a great help for design space exploration: the options stay open.

6.5 SpecC

SpecC is a new language based on the C programming language. SpecC includes with it a methodology for system design, that allows a systematic design space exploration, called specify-explore-refine[14]. This methodology does not tend to support complex target platform[48].

The IDR used in SpecC (SpecC Internal Representation) is similar to ones used in software compilers, essentially a syntax graph.

6.6 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).

SystemC is a language, and doesn't have a complete methodology for system-level design[48].

6.6.1 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].

6.7 VULCAN

At the Stanford University the VULCAN system has been developed. The specification language used is called HardwareC. Although its syntax is C-like its semantics are that of a Hardware Design Language; thus rather low-level.

Initially, a system will be specified as a complete hardware solution (in HardwareC). When the timing and resource constraints are specified, an iterative automatic partitioning approach will be started. The basic idea of VULCAN is to move suitable parts of the system to software (that will then be run on a general purpose processor), thus making the dedicated hardware part smaller in each iteration. This all under the given performance constraints. The main purpose of this is cost reduction.

Internally Vulcan used a Flow Graph as its IDR. The design process Vulcan offers is complete: its support specification, (automatic) partitioning and synthesis of both software and hardware components. The 'price' paid in this approach is that the specification level is low.

6.8 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 (such as 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. Ptolemy is a bit of an outsider here. The main focus of Ptolemy II is (co-)simulation and the other phases get less attention. The fact that it's possible to use multiple Models of Computation in one system is very interesting. A tool similar in this aspect is Music[9].

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

Model	MoC based on	Specification language(s)
COSYMA	(Extended) Syntax Graph	C^x
ForSyDe	Functions ('skeletons')	Haskell
Polis	CFSM	Esterel, graphical
Ptolemy II	Multiple MoC's	Various graphical, others
SpecC	Syntax Graph	SpecC
SystemC	Syntax Graph	SystemC
Vulcan	Flow Graph	HardwareC

Table 6.1: Languages and Internal Design Representations of Co-Design methods.

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. It can be said that there are 2 ways to improve productivity: improving the design process, and reusing more IP[35]. SystemC is less complete in its methodology, but IP reuse has been an important feature. The same goes for SpecC, but SpecC's design methodology is much more mature. At the same time there are languages that have a complete methodology, such as COSYMA and Vulcan, but do not take IP reuse into account.

A recent trend in system design is platform based design, in which IP reuse takes a prominent place. Most of the methods here can be described as top-down methods, using a specification and than refining it. Platform based design works more ad hoc: existing components are 'glued' together. Extensions to Polis that take IP reuse into account [30] make it possible to use the best of two worlds. A commercial tools based on Polis that also deals with IP reuse is VCC, from Cadence.

It is clear that the choice of IDR is fundamental to a Co-Design methodology. Using a mathematically strong IDR such as CSFM, allow POLIS to reach a high level of abstraction and allow design-space exploration, with a broad variety of target architectures. SpecC and SystemC, using a more pragmatic IDRs based on syntax graphs (with extensions), also allow synthesis to many target architectures, while supporting IP reuse. However, they pay a 'price' in abstraction level, and the partitioning is not automated. Two other languages based on similar IDR, COSYMA and Vulcan, do allow automatic partitioning, but only for a very limited type of architecture. Their representation languages (C^x and HardwareC respectively) are on a low level of abstraction.

Table 6.2: Questions from Chapter 5.

	COSYMA	ForSyDe	Polis	Ptolemy II	SpecC	SystemC	Vulcan
Integration various MoCs	1 IDR ?	1 IDR	1 IDR	Multiple	1 IDR	1 IDR	1 IDR
What is the origin of the IDR of the approach	$\mathcal{M}_{CD}$	$\mathcal{M}_{\overline{b}}$	$\mathcal{M}_{CD}$[46]	-	$\mathcal{M}_{CD}$	$\mathcal{M}_{CD}$	$\mathcal{M}_{DC}$[46]
Design-space exploration supported?	no	no	Y-chart[49]	Y-chart like[36]	specify-explore-refine[14][b]	no	no
IP re-use	no	no	yes[a]	yes[d]	yes	yes	no
Target-architecture(s)	Single RISC processor (Sparc), with one ASIC (co-processor) that communicate using shared memory.	Single processor[e]	Multiple processors (MIPS), and CFSMs implemented in hardware (multiple co-processors).	Various[c]	Various	Various	Single processor, various ASICs
The abstraction level of the approach	-	++	++	varies	++	+	-
Is (automatic) partitioning supported	yes	no, manual	no, manual	no[c]	no, manual	no, manual	yes

[a]In Polis itself this has not specifically been addressed, but there are extensions[30] that take IP reuse into account. Also in tools based on Polis such as VCC it can be found.
[b]The design-space exploration of SpecC is less strong than a real Y-chart approach[48]
[c]No complete synthesis-trajectory exists for stand-alone Ptolemy, only for specific MoC's.
[d]Ptolemy can certainly be used together with other tools that support IP reuse specifically, such as in [30].
[e]Using ForSyDe either sequential C code for a processor can be generated, or (behavioral) VHDL to synthesize hardware.

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. [18]. 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.

[22] 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.

Hardware software partitioning is not the question, the question is how can you make the right trade-offs. This can only be explored by design-space exploration tools. This way various mappings from MoC to architecture can be compared to each other.

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] J. L. Diaz-Herrera. A survey of system-level design notations for embedded systems. draft, 2001.

[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, R. Dömer, and J. Zhu. IP-centric methodology and design with the SpecC language. Technical report, University of California, Irvine, 1998.

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

[16] R. K. Gupta. *CO-SYNTHESIS OF HARDWARE AND SOFTWARE FOR DIGITAL EMBEDDED SYSTEMS*. PhD thesis, Stanford University, 1993.

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

[18] 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.

[19] 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.

[20] D. S. Jovanovic, G. H. Hilderink, and J. F. Broenink. Integrated design tool for embedded control systems. In F. Karelse, editor, *Proceedings of the 2nd Progress Workshop on Embedded Systems*. Technology Foundation (STW), Oct. 2001.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

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

[28] 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.

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

[30] M. Meerwein, C. Baumgartner, and W. Glauert. Linking codesign and reuse in embedded systems design. In *Proceedings of the eighth international workshop on Hardware/software codesign*, pages 93–97. ACM Press, 2000.

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

[32] 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.

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

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

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

[36] E. K. Pauer and J. B. Prime. An architectural trade capability using the ptolemy kernel. Technical report, Sanders, a Lockheed Martin Company, Nashua, 1996.

[37] J. L. Pino. Software synthesis for single-processor dsp systems using ptolemy. Master's thesis, University of California, Berkeley, 1993.

[38] 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.

[39] 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.

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

[41] J. Smit, M. Stekelenburg, C. Klaassen, S. Mullender, G. J. M. Smit, and P. J. Havinga. Low cost & fast turnaround: Reconfigurable graph-based execution units. In *Proceedings Belsign Workshop*, 1998.

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

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

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

[45] 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.

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

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

[48] V. D. Živković, E. Deprettere, P. van der Wolfa, and E. Kock. From high level application specification to system-level architecture definition: Exploration, design and complitation. In *Proceedings of The International Workshop on Compilers for Parallel Computers (CPC03),Amsterdam, the Netherlands*, pages 39–49, January 2003.

[49] 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.

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

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

List of Figures

Appendix A

Vocabulary

ASIC	Application specific Integrated Circuit
FPGA	Field-Programmable Gate Array (a specific type of PLD)
FSM	Finite State Machine (see Chapter 4.3.4)
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