

January 21, 2003

## Co-Design for Chess-iT (draft)

by

**G. Bosman**

Supervisors:

Dr. Ir. A. M. Bos (Chess-iT)

Ing. P. G. C. Eussen (Chess-iT)

Dr. Ing. R. Lämmel (Vrije Universiteit)



*vrije* Universiteit

*amsterdam*

## Contents

### Chapter

<b>1</b>	Introduction	1
1.1	Traditional design . . . . .	1
1.2	Programmable logic . . . . .	2
1.3	Concurrency . . . . .	3
1.4	Outline . . . . .	3
<b>2</b>	Assignment	4
<b>3</b>	Co-Design	5
3.1	Related approaches . . . . .	5
3.1.1	Software in the loop . . . . .	5
3.1.2	Chunky function unit architecture . . . . .	5
3.1.3	Reconfigurable hardware . . . . .	6
3.2	Modelling and specification . . . . .	6
3.2.1	Modelling . . . . .	6
3.2.2	Validation . . . . .	7
3.2.3	Synthesis . . . . .	8
<b>4</b>	System Level Modelling	9
4.1	Homogenous modelling . . . . .	9
4.2	Heterogeneous modelling . . . . .	10

4.2.1	Co-simulation vs. compositioning . . . . .	10
4.3	Concurrency or: Dataflow vs. Control flow oriented . . . . .	11
4.4	Hierarchy . . . . .	11
4.5	Communication . . . . .	11
4.6	Synchronization . . . . .	11
<b>5</b>	<b>Computational Models</b>	<b>12</b>
5.1	Properties . . . . .	12
5.2	Imperative models . . . . .	12
5.3	Differential Equations . . . . .	12
5.4	Difference Equations . . . . .	13
5.5	Process networks and dataflow . . . . .	13
5.6	Discrete-event models . . . . .	13
5.7	Petri Nets . . . . .	13
5.8	Synchronous models . . . . .	14
5.9	Rendezvous models . . . . .	14
5.10	Finite-state machines . . . . .	15
5.11	Comparison . . . . .	15
<b>6</b>	<b>Hybrid systems</b>	<b>16</b>
6.1	Ptolemy II . . . . .	17
6.2	*charts . . . . .	17
6.3	Moses framework . . . . .	17
6.4	Solar/Music . . . . .	17
6.5	The SPI Workbench . . . . .	17
6.6	Emergent behavior . . . . .	17
6.7	On synthesizing code . . . . .	18

<b>7</b>	<b>Languages</b>	<b>19</b>
7.1	System C . . . . .	19
7.2	VULCAN/HardwareC . . . . .	20
7.3	Polis/Esterel . . . . .	20
7.4	SDL . . . . .	20
7.5	Haskell . . . . .	21
7.6	VHDL . . . . .	21
<b>8</b>	<b>Case Studies</b>	<b>22</b>
8.1	Method A: ForSyDe . . . . .	22
8.1.1	Digital Equalizer . . . . .	22
8.2	Method B: SDL and Matlab . . . . .	22
8.2.1	Digital Equalizer . . . . .	22
<b>9</b>	<b>Conclusions</b>	<b>23</b>
	<b>Bibliography</b>	<b>24</b>

## Chapter 1

### Introduction

[Guus: Chess or Progress vision on growth of market of embedded computing here.] A very important choice to be made when designing an embedded computer system is how to divide the functionality of the system in hardware and software parts. 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. Goal of co-design is to explore the whole design-space to be able to make well-informed decisions and to be able to make this decision in a later phase in the design process.

#### 1.1 Traditional design

Traditionally complex embedded systems are designed around a microprocessor in a Von Neumann architecture. A traditional Von Neumann system is fundamentally a sequential system. There is heavy optimisation inside the CPU itself (pipelining etc) but ultimately the commands are executed one by one. Systems based on microprocessors have several benefits. Microprocessors are very well optimized and they allow design of families of product that can be built. Such a family of product can provide various feature sets at a different price point and can be extended to keep up with rapidly changing markets[29].

However, the fact that each command is executed sequentially leads to a fundamental limitation. Backus [1978] calls this the "von Neumann bottleneck." He points out that this bottleneck is not only a physical limitation, but has also served as an "intellectual bottleneck"

in limiting the way we think about computation and how to program it.

ASICs are specialized chips, that are used for example for [...]. They are similar to processors in the sense that they are also 'hardwired' solutions. It is very expensive to design an ASIC, and a very slow process. Therefore, customizing an ASIC for a single application is often not feasible. Early approaches in Co-Design therefore started with a components of high granularity, such as ASIC and microprocessors.

If you'll design your whole system from the ground up (thus without using a microprocessor), or do this with a part of your system the optimizations found in existing microprocessors are not available. The designer will have to implement these. At the other side however there is a huge potential gain because the layout of the hardware can in theory be tailored exactly to the functional requirements. This can be extremely profitable, especially when the problem to be solved is mainly parallel in character. Therefore an integrated design approach is very interesting.

Often a mixture is used. Parts of the hardware are designed from scratch, but a microprocessor is also used. This allows the system to benefit from the microprocessor's specialization, and for the parts of the system that benefit most of this a parallel implementations.

## 1.2 Programmable logic

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 which may be reprogrammed as often as desired. FPGA's are a reconfigurable logic devices that are becoming very popular[7]. Three direct benefits of the reconfigurable approach can be recognized: specialization, reconfigurability and parallelism[26] [Guus: use more of Tessier on future developments [26]]. FPGA's shorten the development cycle dramatically and are much cheaper to use too. This allowed research to Co-Design to increase a lot, and also consider more fine-grained approaches.

Obviously designing hardware and software integrated 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.

In my thesis I'll investigate what methods exist that comply with the Chess business. I'll compare them with each other. I'll also investigate the problem of the paradigm shift (or the prevention of this) when designing parallelism in a high-level language. Ideally the resulting hard- and software description should be parallel in nature.

Many languages exist that are C-like, academic languages exist, each with a different focus. In this paper I'll give an overview of the field and indicate the open issues. I'll try to answer to question which languages is suitable for which problem.

### **1.3 Concurrency**

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.

### **1.4 Outline**

First we'll look at the definition of Co-Design and the problems it tries to solve. In Chapter 4 the different types of approaches are described, and which classifications can be made. It turns out that the paradigms, the computational models, are very important. Chapter 5 deals with this.

A single paradigm approach has serious disadvantages so various hybrid models have been proposed in literature. These will be described in Chapter 6, whereas in Chapter 7 existing projects and models will be described and analyzed using the classifications and ideas found in the first part. Finally we'll look into a few case studies in Chapter 8 to get a better view of how the multi-paradigm modelling works and how well it can be applied.

## Chapter 2

### Assignment

”To investigate various development methods and to investigate how an integrated approach of HW/SW design can improve system development at Chess.”

I’ll give a thorough introduction to the field of codesign. and how this could support the design of programmable hardware such as FPGA’s.

I will investigate the relevance and relationships between these specification and design languages in the track from specification to implementation. I’ll research to which degree existing development and implementation methods support compatible paradigms.

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



## Chapter 3

### Co-Design

”A design methodology supporting the concurrent development of hardware and software (co-specification, co-development, and co-verification) in order to achieve shared functionality and performance goals for a combined system.” [Guus: found it on the internet, where is it from exactly?]

#### 3.1 Related approaches

##### 3.1.1 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 eases the design of software for hardware it does not allow the full improvements made possible by Co-Design.

##### 3.1.2 Chunky function unit architecture

'The configurable computing community is divided into two camps, according to the level of abstraction provided by the programmable hardware. This thesis deals with the hardware on a very fine level of granularity: digital circuits are translated into netlists, which are composed of logic gates and flip-flop. However in the second camp are the architecture based on "chunky" function units such as complete ALU's and multipliers. These architectures limit the programmable hardware to the interconnect among the function units, but implement those units in much less IC area.' [22]

### 3.1.3 Reconfigurable hardware

Reconfigurable systems exploit FPGA technology, so that they can be personalized after manufacturing to fit a specific application. The operation of reconfigurable systems can either involve a configuration phase followed by an execution phase or have concurrent (partial) configuration and execution.[23]. The major co-design 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.

## 3.2 Modelling and specification

There are several tasks that must be performed to create a system-level design. To comprehend the benefits of various Co-Design technologies it is important to understand how the design process works.

The co-design system design process for embedded systems includes modelling, validation, and implementation[23]. These processes are fundamental steps in any methodology aimed to design an embedded system.

### 3.2.1 Modelling

'There is a subtle relationship between the specification of a system and the modelling of a system. An executable specification, for example, is also a model of an implementation. The difference is in emphasis. A specification describes the functionality of a system, and may also describe one or more implementations. A model of a system describes the functionality. In a specification it is important to avoid over-specifying the design, to leave implementation options open. In a model, often the key criteria are precision, simplicity and efficient simulation. A model should be the most abstract model that represents the details being tested.'

[4].

Specification is closer to the problem level, at a higher level of abstraction, and uses one or more models of computation. A Specification undergoes a synthesis process (which may be partly manual) that generates a model of an implementation. That model itself may contain

multiple models of computation.

The outcome of the modelling process is the internal design representation (IDR). There is a trade-off between scalability and expressiveness in this IDR[28]

#### 3.2.1.1 Hardware/software Partitioning

[Guus: should be somewhere else]. 'The partition of a system into hardware and software is of critical importance because it has a huge impact on the cost/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).'[23]. When considering general purpose computing systems, a partition represents a logical division of system functionality, where the underlying hardware is designed to support the software implementation of the complete system functionality. This division is elegantly captured by the instruction set. Thus instruction selection strongly affects the system hardware/software organization.

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. This is called **coprocessing**[23]

#### 3.2.2 Validation

Through the validation process, the designer achieves a reasonable level of confidence about how much of the original embedded system design will be in fact reflected in the final implementation.[28]. There are 3 three methods for validation:

- (1) Simulation
- (2) Prototyping
- (3) Formal Verification

There has been a lot of research in the simulation of heterogeneous hardware/software systems [28, 18, 4]. Formal verification allows for a more thorough test of the embedded system behavior (maximum behavioral coverage) by means of logics.

### 3.2.3 Synthesis

The final stage in the development of an embedded system is the synthesis process. In the phase architectural information is taken into account. Varea[28] calls this a merger between the IDR with the **technology library**. It is important that the intermediate IDR or specification is too operational (influenced by the current technology), it will bias the design towards a specific architecture.

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.

## Chapter 4

### System Level Modelling

Mooney et al.[24]: Approaches to hardware/software co-design of embedded systems can be differentiated in several ways. One way is to consider the system-level specification, which is either homogeneous (i.e., in a single specification language) or heterogeneous (i.e. involving multiple modelling paradigms). Another way is to distinguish how the CAD tool partitions the system specification: fine-grained or coarse-grained.

#### 4.1 Homogenous modelling

Homogeneous modelling implies the use of single specification language for the modelling of the overall system. Lee[19] calls this the 'grand unified method'. Co-design starts with a global specification given in a single language. This specification may be independent of the future implementation and the partitioning of the system into hardware and software parts. In this case co-design includes a partitioning step aimed to split this initial 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]. Lee[19] sees as a big problem that a homogenous approach imposes a model of computation which might be good for a subset of systems but bad for others.

## 4.2 Heterogeneous modelling

Heterogeneous modelling allows the use of specific languages for the hardware and software parts. The co-design starts with a virtual prototype where the hardware/software partitioning is already made. Here the emphasis is on the integral designing of the parts to make sure 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. [this sentence from the SPI Workbench].

### 4.2.1 Co-simulation vs. compositioning

[6] differences between 2 different types of heterogeneous modelling. The compositional approach 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. Examples are Polis, Javetime and SpecC.

The cosimulation-based approach consists in interconnecting the design environments associated to each of the partial specifications. Like its name suggests, with co-simulation the software parts and the hardware components of a system and their interactions are simulated in one simulation. It does not provide such a deep integration as compositioning does however it does allow for modular design. Communication is often done using a cosimulation bus, that is in charge of transferring data between the different simulators.

The Cosimulation-field is reasonably well established. Sometimes cosimulation is used to simulate the behaviour 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.

It is good to note that 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 FPGA's) that simulate the real target hardware[18].

### **4.3 Concurrency or: Dataflow vs. Control flow oriented**

There are languages that are designed to describe dataflow oriented systems (ie DSPs) and there are models more suitable for control-flow systems. [Guus: what exactly is the difference].

However this approach seems to be not too good as there are many systems that are not easily put in either one category. Most complex systems are a combination of data- and control-flow oriented parts.

### **4.4 Hierarchy**

Used for dealing with complexity. (Also called layering)

### **4.5 Communication**

There are two basic models for communication, message passing and shared memory. Or Remote Procedure Calls.

### **4.6 Synchronization**

There are 2 synchronization modes: synchronous and asynchronous.

## Chapter 5

### Computational Models

All Co-Design systems are based on a computational model, or combine a few of them.

A computational model is a formal, abstract definition of a computer. It describes the components in a system and how they communicate and execute. Several models exist. There are a number of authors who made an overview of various development methods, i.e. [30], [16].

#### 5.1 Properties

[Guus: here explain what I want to find about of the models]. I.e., timing, hierarchy.] An essential difference between concurrent models of computation is their modelling of time. [19] (page 11). Lee[20] proposed a mathematical framework to compare certain properties of models of computation. This allows for a precise definition of the various computational models.

#### 5.2 Imperative models

In an imperative model of computation, modules are executed sequentially to accomplish a task.[4].

#### 5.3 Differential Equations

These are often used to model mechanical dynamics, analog circuits, chemical processes and many other physical systems. [11].TBD. When using real numbers as time model,



continuous-time systems are active over the entire time axis processing their input and producing output.[25].

## 5.4 Difference Equations

Like differential equations, but discrete. TBD. These two are very important as they deal with a very common type of signal from the outside world. Discrete-time systems can only react to their input and produce new output at distinct, equidistant time instances. [25]

## 5.5 Process networks and dataflow

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.[19]. They are not suitable for control-logic. [Guus: uitwerken wat voor soort data-flow modellen er allemaal bestaan.]

## 5.6 Discrete-event models

In a discrete-event system, modules react to event that occur at a given time instant and produce other events either at the same time instant or at some future time instant. Execution is chronological.[4]. 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 DE simulator 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].

## 5.7 Petri Nets

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 transition to places, and an output function which is also a mapping from transition to places. This is a well-understood modelling tool. Two important features of Petri nets are its concurrency and asynchronous nature.[5].

## 5.8 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.[4]

Very often real-time systems are specified by means of concurrent processes, which communicate asynchronously [15].

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 of 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.'

This model serves as a good implementation model.

## 5.9 Rendezvous models

In a rendezvous model, the arcs represent sequences of atomic exchanges of data between sequential processes, and the bubbles presents the processes. [19]. 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?]

## 5.10 Finite-state machines

The classical 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.[5]. FSM are excellent for control logic in embedded systems[19]. They can very well be formally analyzed and it is relatively straightforward to synthesis code from this model. FSM have a number of weaknesses. They are not very expressive, and the 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. [Guus: SOLAR here[6]]. Another one is the CFSM model. It is based on FSMs and the communication primitive is called event.[5]. See also paragraph 6.2.

## 5.11 Comparison

[5] also made a comparison of various computational models.

Computational model	Clock	Property X
Differential Equations	?	?
Difference Equations	?	?
Process networks and dataflow	?	?
Discrete-event models	?	?
Petri Nets	Asynchronous	?
Synchronous/reactive models	Synchronous	?
Rendez-vous	?	?
CFSM	?	?
SOLAR	?	?

## Chapter 6

### Hybrid systems

Hybrid systems are systems that allow more than one computational model in a system to be used. Experience suggest that several MoC are required for the design of a complete system.[4].

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. An imperative language can be used to implement for example a dataflow MoC[4]. It is obvious that low-level languages such as VHDL are able to implement different models of computation. However, their lack of abstraction disqualifies them as candidates for modelling combined computational model-systems.

Many languages and tools that were developed based on a single model start to embrace other models [11]. The downside of such large languages with multiple MoC's is (according to [4]) that formal analysis may become very difficult. 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.

However also new frameworks have been developed that took multiple paradigms into their design from the beginning.

[Guus: this is an important chapter and I want to extend this a lot.]

## 6.1 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.[11]. 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.

## 6.2 \*charts

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

A recent development is \*Charts (pronounced Starcharts). TBD.

## 6.3 Moses framework

## 6.4 Solar/Music

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

## 6.5 The SPI Workbench

The SPI Workbench citeErnst is based on intervals of system properties and is specifically targeted to cosynthesis.

## 6.6 Emergent behavior

'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.'[11] '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 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.’ [4]

There are other ways to mix models of computation too. StateMate uses views.

## 6.7 On synthesizing code

A discrete-event model of computation is well suited for generating hardware. It is not very suitable to generate (sequential) software [4] (p. 131). This is for example why VHDL simulates 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 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 re-targetable. Hierarchical FMS 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.’ [4] [Guus: very interesting, expand this].

## Chapter 7

### Languages

In this chapter I'll take a look at various Co-Design methods. Not only at academical ones but also commercial ones.

A flat representation of a realistic embedded system would be too complicated to understand in many situations. A layered approach is therefore mandatory.

#### 7.1 System C

The SystemC language is becoming a new standard in the field and many designers are starting to use it to model complex systems. It is an industrially used language, not an academical one.

SystemC has been mainly adopted to define abstract models of hardware/software components, since they can be easily integrated for rapid prototyping. However it can also be used to describe modules at a higher level of detail, E.g., RT-level hardware descriptions and assembly software modules. Thus it would be possible to imagine a SystemC-based design flow where the system description is translated from one abstraction level to the following one by always using SystemC representations.[1].

SystemC is basically C++ with an extra class-library.

## 7.2 VULCAN/HardwareC

Another C-type language is HandelC. This is an extension of C though, with real new language constructs. It is influenced by the Occam language.

The input to the co-synthesis system is described in hardware description language called HardwareC, a subset of C defined for hardware description. The architecture is limited to a one-CPU-one-bus architecture. Type of the CPU must be a general purpose register machine with one-level memory. [30]

## 7.3 Polis/Esterel

The Polis system developed at the university of California, Berekely, implements a HW/SW Co-Design system using the CSFM model as its basis. The textual language used in the Polis project is called Esterel. There is work in progress on a system that uses StateMateTM, a graphical specification tool, to create specifications for the Polis system[14].

CSFM stands for 'Codesign Finite State Machine'. Is focussed on reactive systems or control oriented applications[15]. Esterel is a synchronous language.

## 7.4 SDL

SDL (specification and description language) is intended for the modelling and simulation of real-time, distributed and telecommunication systems and is standardized by the ITU. A system described in SDL is regarded as a set of concurrent processes that communicate with each other using signals. SDL supports different concepts for describing systems such as structure, behavior and communication. SDL is intended for describing large designs at the system level. There are two SDL formats, a textual and a graphical one.[8] In [8] a system is defined that allows the generation of VHDL from system level specifications in SDL.

In [3] a case-study is presented that uses SDL modelling to generate a VHDL description. It is then implemented in a FPGA.



## 7.5 Haskell

A Haskell program is a function, which consists of a composition of other functions. Functions produce only one result, although this results can be a tuple consisting of values multiple data-types.[15]. I.e. there are no side-effects.

A design method that uses Haskell is ForSyDe, see also Chapter8for a case study.

## 7.6 VHDL

[Guus: this section should be somewhere else, but where?] Hardware is usually described using a gate-level netlist. This can be used to lay-out a printed circuit board or to program a PLD (Programmable Logic Device). Within Chess FPGA's are an example of PLD's. FPGAs are typically based op Look-Up Tables implemented using high-speed SRAM cells. The permit the FPGA to implement any arbitrary function of the inputs. These netlists are very low-level descriptions. Usually to describe hardware a Hardware Definition Language is used. The two commonly used ones are VHDL and Verilog. Many methods for Co-Design focus on generating C for the software part and VHDL for the hardware part. Verilog is a somewhat easier but less rich in expression compared to VHDL.

VHDL was intended for use in the development, simulation, synthesis and testing of digital circuits. There is a behavioral and a procedural variant of VHDL. The first one is not deterministic when used to generate a netlist, the second one is.

## Chapter 8

### Case Studies

#### 8.1 Method A: ForSyDe

##### 8.1.1 Digital Equalizer

Lu[21] 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.

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.

#### 8.2 Method B: SDL and Matlab

##### 8.2.1 Digital Equalizer

[2] also investigated the design of a Digital Equalizer. They used a combination of SDL and Matlab as their design languages.

SDL is used to model the control parts, Matlab is used for the DSP parts.

## Chapter 9

### Conclusions

Most Co-Design tools make use of an internal representation for the refinement of the input specification and architectures[16]. Many specification languages exist.

Experiments with system specification languages have shown that there is not a unique universal specification language to support the whole design process for all kinds of applications. [16].

The fundamental computation model to use is first of all dependent on the type of problem to be solved. However there is a definite trend towards heterogenous modelling systems that allow more mixed types of systems to be modelled.

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[4]. In order to be sufficiently rich to encompass the varied semantic models for the competing approaches, they become unwieldy, too complex for formal analysis and high quality synthesis. He claims that generality can be achieved through heterogeneity, where more than model of computation is used.

Ptolemy II seems to be the most mature of the tools investigated, specifically the theoretical foundation of mixing various computational models is well thought-out.

[Guus: etc etc... this will take a lot of work!]

## Bibliography

- [1] A. Fin, F. Fummi, M. Martignano, and M. Signoretto. SystemC: A homogenous environment to test embedded systems. In Proceedings of the Ninth International Symposium on Hardware/Software Codesign (CODES-01), pages 17–22. ACM Press, Apr. 2001.
- [2] P. Bjurus and A. Jantsch. Mascot: a specification and cosimulation method integrating data and control flow. In Proceedings of the conference on Design, automation and test in Europe, pages 161–168. ACM Press, 2000.
- [3] C. A. Marcon, F. P. Hessel, A. M. Amory, L. H. L. Ries, F. G. Moraes, and N. L. V. Calazans. Prototyping of embedded digital systems from SDL language: a case study. In Proc. Seventh Annual IEEE International Workshop on High Level Design Validation and Test, 2002. To appear.
- [4] 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.
- [5] L. A. Cortés, P. Eles, and Z. Peng. A survey on hardware/software codesign representation models. Technical report, Linkping University, June 1999.
- [6] P. Coste, F. Hessel, and A. Jerraya. Multilanguage codesign using sdl and matlab, 2000.
- [7] S. Cotofana, S. Wong, and S. Vassiliadis. Embedded processors: Characteristics and trends. Technical report, Delft University of Technology, 2001.
- [8] J.-M. Daveau, G. F. Marchioro, and A. A. Jerraya. VHDL generation from SDL specification. In C. Delgado Kloos and E. Cerny, editors, Hardware Description Languages and their Applications (CHDL '97), Toledo, Spain, Apr. 1997. IFIP WG 10.5, Chapman and Hall.
- [9] S. A. Edwards. The Specification and Execution of Heterogeneous Synchronous Reactive Systems. PhD thesis, University of California, Berkeley, 1997.
- [10] S. A. Edwards. Design languages for embedded systems. Technical report, Synopsys, Inc., 2001.
- [11] 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.
- [12] F. Slomka, M. Dörfel, and R. Münzenberger. Generating mixed hardware/software systems from SDL specifications. In Proceedings of the Ninth International Symposium on Hardware/Software Codesign (CODES-01), pages 116–121. ACM Press, Apr. 2001.
- [13] D. Gajski and F. Vahid. Specification and design of embedded software-hardware systems. IEEE Design & Test of Computers, 12(1), 1995.

- [14] I. D. Bates, E. G. Chester, and D. J. Kinniment. A statechord based HW/SW codesign system. In Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES-99), pages 162–166. ACM Press, May 1999.
- [15] 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. ACM Press, May 1999.
- [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] 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.
- [19] E. A. Lee. System-level design methodology for embedded signal processors. Technical Report F33615-93-C-1317, University of California at Berkeley, 1997.
- [20] 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.
- [21] Z. Lu. Refinement of a system specification for a digital equalizer into HW and SW implementations. January, Royal Institute of Technology, 2002.
- [22] W. H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. K. Prasanna, and H. A. Spaanenburg. Seeking solutions in configurable computing. IEEE Computer, 30(12):38–43, December 1997.
- [23] G. D. Micheli and R. K. Gupta. Hardware/software co-design. In Proceedings of the IEEE, volume 85, pages 349–365, Mar. 1997.
- [24] 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.
- [25] T. M. Stauner. Systematic Development of Hybrid Systems. PhD thesis, Institut für Informatik der Technischen Universität München, 2001.
- [26] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. Journal of VLSI Signal Processing, 28(1):7–27, June 2001.
- [27] D. E. Thomas, J. K. Adams, and H. Schmit. A model and methodology for hardware-software codesign. IEEE Design & Test of Computers, Sept. 1993.
- [28] M. Varea. Mixed control/data-flow representation for modelling and verification of embedded systems. Technical report, University of Southampton, Mar. 2002.
- [29] W. Wolf. Computers as components: principles of embedded computing system design. Academic Press, 2001.
- [30] T.-Y. Yen and W. Wolf. Hardware-Software Co-Synthesis of Distributed Embedded Systems. Kluwer Academic Publishers, 1996.