February 2, 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. R. Lämmel (Vrije Universiteit)

vrije Universiteit *amsterdam*

Contents

Chapter

Chapter 1

Introduction

Due to the increasing size of integrated circuits and time-to-market requirements designing embedded systems is becoming more and more complex. This increasing complexity of embedded systems is the driving motivation for a high-level system design paradigm. A high-level system design notation could also offer substantial performance benefits as there can be a more optimal partitioning of the system's parts onto hardware and software. High-level system design is currently a major research area by academics world-wide and it is strongly rooted in traditional areas such as control theory and digital design theory. For Chess high-level system design could mean a faster and better design process. This paper investigates a number of aspects of system-level design and the current state of the art in existing tools and methods that allow system-level design.

Such a high-level system design paradigm should be at an abstract level that allows reasoning at a high-level of abstraction. However, to be useful it must also be possible to synthesis code from this high-level language for hardware and software. How to divide the functionality of the system in hardware and software parts is 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. 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. This would lead to a more optimal partitioning and more flexibility during the design

process.

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 we will mainly use "Co-Design." To really understand the improvement a high-level system design paradigm would be, it is important to see how traditional system design works and what the issues are with that.

1.1 Traditional design

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 (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 products 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[36].

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 been an "intellectual bottleneck" in limiting the way we think about computation and how to program.

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

Not all embedded systems are designed around microprocessors. It is possible to design embedded chips that compute in a parallel way. 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.

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, encryption etc. which are parallel 'in nature'.

An important motivation for the rising interest in Co-Design is that there now is another type of chip that is much more flexible than ASICs.

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 reconfigurable logic devices that are becoming very popular[7]. Three direct benefits of the reconfigurable approach can be recognized: specialization, reconfigurability and parallelism[33] [Guus: use more of Tessier on future developments [33]]. FPGA's shorten the development cycle dramatically and are much cheaper to use than ASICs. This allowed research to Co-Design to increase a lot. It also made researchers consider more fine-grained approaches.

Early approaches in Co-Design therefore started with components of high granularity, such as ASIC and microprocessors. 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 both the microprocessor's specialization, and for the parts of the system that benefit most of this a parallel implementations.

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.

In this thesis it is investigated what methods exists 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 where possible.

In this paper I'll give an overview of the field and indicate the open issues. I'll try to answer to question which type of language is suitable for which problem.

1.3 Outline

First we'll look at what Co-Design is and the problems it tries to solve. In Chapter 5 different types of Co-Design approaches are described, and the classifications that can be made. It turns out that the paradigms, the computational models, are very important. They are directly related to the paradigm shift. Chapter 6 describes computational models that are commonly used to model (parts of) systems.

A single paradigm approach has serious disadvantages so various hybrid models have been proposed in literature. In Chapter 7 existing projects and models will be described and analyzed using the classifications and ideas found in the first part. Per method we'll look at some case studies to get a better view of how the multi-paradigm modelling works and how well it can be applied. In Chapter 8 the pro's and contra's of the investigated heterogeneous methods will be discussed.

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 introduction to the field of Co-Design. 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 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."[**?**].

3.1 Abstraction

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 an much higher level than machine code allows. Analog have there has been a lot of research into finding languages to design hardware from a higher level. Nowadays it is very common to use a language as VHDL to define hardware. There are compilers available to generate netlists (hardware descriptions) languages like VHDL.

This looking for a higher level of abstraction 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.

3.2 Modelling and specification

Modelling is the process of conceptualizing and refining the specification. The result of the modelling phase is a model, which is specified in a internal design representation. There are several tasks that must be performed to create a system-level design model. 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[27]. 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[35]. In Chapter 5 we'll go deeper into the modelling process.

3.2.2 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).'[27]. 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**[27]

3.2.3 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 be reflected in the final implementation.[35]. 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 [35, 19, 4]. Formal verification allows for a more thorough test of the embedded system behavior (maximum behavioral coverage) by means of logics.

3.2.4 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[35] calls this a merger between the IDR with the **technology library**. It is important that the intermediate IDR or specification is not 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

Background and related work

4.1 History

[Guus: about Gupta paper etc]

In 1998 a paper was published that described a method to synthesize code for both hardware and software, for a specific type of data-flow programs[11].

4.2 Related work

[Guus: about other comparative papers]

4.3 Related approaches

4.3.1 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.

4.3.2 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 com-

posed of logic gates and flip-flop. However in the second camp are the architecture based on "chunky" function units suchs as complete ALU's and multipliers. There architectures limit the programmable hardware to the interconnect among the function units, but implement those units in much less IC area.' [25]

4.3.3 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.[27]. 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.

4.3.4 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 related engineering principles. Because of the way models are often treated (digital) the analog environment is often abstracted away (to a digital translation) by computer scientists. This way traditional Co-Design is unable to guarantee safety and/or performance of the embedded device as a whole.

To address this issue **hybrid** embedded system models have been designed[1, 32]. 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. In fact, the difference between the two is not always very sharp. [TBD: on differential equations].

Chapter 5

System Level Modelling

Mooney et al.[28]: 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 design methods deal with the SW/HW partitioning: fine-grained or coarse-grained. Modelling in the context of Co-Design is sometimes called **cospecification**.

5.1 Homogenous modelling

Homogeneous modelling implies the use of single specification language for the modelling of the overall system. Lee[20] 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. The is generally called a virtual prototype and may be given in a single language or different languages[16]. Lee 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[20].

5.2 Heterogeneous modelling

Heterogeneous 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. 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 tat enables system optimization across language boundaries. [this sentence from the SPI Workbench].

5.2.1 Co-simulation vs. compositioning

[6] differences between 2 different types of heterogenous 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, Javatime and SpecC.

The cosimulation-based approach consists in interconnecting the design environments associated to each of the partial specifications. Like its names suggests, with co-simulation the software parts and the hardware components of an 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 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. [Guus: really? This is interesting. Expand on this.]

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 FPGAs) that simulate the real target hardware[19].

It is clear that the choice of an IDR is a very important aspect of a Co-Design method, therefore in the next chapter we'll go into the various types of IDR's there are.

Chapter 6

Computational Models

Modelling is a the heart of development methods. The computational models can be found in the Immediate Representation Language. 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 exists. There are a number of authors who made an overview of various development methods, i.e. [37], [16].

[Guus: here explain what I want to find about of the models]. Ie, timing, hierarchy.] An essential difference between concurrent models of computation is their modelling of time. [20] (page 11). Lui[22] states that the different notions of time make programming of embedded systems significantly different from programming in desktop, enterprise or Internet applications. Lee[21] proposed a mathematical framework to compare certain properties of models of computation. This allows for a precise definition of the various computational models.

6.1 Concurreny or: Dataflow vs. Control flow oriented

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

There are models that are designed to describe dataflow oriented systems (ie DSPs) and

there are models more suitable for control-flow systems.However this approach lacks generality as most systems are not easily put in either one category. Most complex system are a combination of data- and control-flow oriented parts. Varea[35] proposes a classification according to the following taxonomy:

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

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

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

[Guus: use this classification!] 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[23].

6.2 Communication

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

6.3 Synchronization

There are 2 synchronization modes: synchronous and asynchronous.

6.4 Common computational models

6.4.1 Imperative models

In an imperative model of computation, modules are executed sequentially to accomplish a task. This is the most trivial computational model there is and most authors don't even

mention it. However in Chang's paper[4] it is mentioned because it can be used in combination with other models in a hierarchical system. See also Chapter 8.1.

6.4.2 Differential Equations

These are often used to model mechanical dynamics, analog circuits, chemical processes and many other physical systems. [12].TBD. When using real numbers as time model, continuous-time systems are active over the entire time axis processing their input and producing output.[32].

6.4.3 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. [32]

6.4.4 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.[20]. They are not suitable for control-logic. [Guus: uitwerken wat voor soort data-flow modellen er allemaal bestaan.]

6.4.5 Discrete-event models

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

6.4.6 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 transitions to places, and an output function which is also a mapping from transition to places. This is an well-understood modelling tool. Two important features of Petri nets are its concurrency and asynchronous nature.[5]. [Guus: expand a little bit: where are petri nets good for?]

6.4.7 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 [30].

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

'In synchronous languages, every signal is conceptually (or explicitly) accompanied by a clock signal. The clock signal has meaning relative to other clock signals. It defines the global ordering or events. Thus, when compariung 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.'[4].

This model serves as a good implementation model.

6.4.8 Rendezvous models

In a rendezvous model, the arcs represent sequences of atomic exchanges of data between sequential processes, and the bubbles presents the processes. [20]. 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?]

6.4.9 Finite-state machines

The finite-state machine model has been widely used in control theory and is the foundation for the development of several models for control-dominated embedded systems. The classical 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[20]. 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 7.7.

6.5 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	$\mathcal{M}_{CD}$
Synchronous/reactive models	Synchronous	?
Rendez-vous	?	?
CFSM	$\mathcal{M}_{CD}$	?
SOLAR	?	?

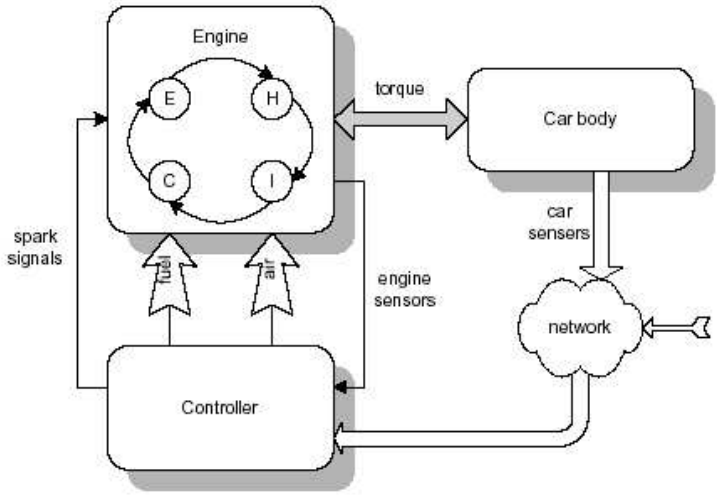
Chapter 7

Heterogeneous systems

Heterogeneous 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]. A nice illustration of the need for multiple computational models can be found in [22]:

[Guus: how can you determine system behaviour for these types of systems? Things like deadlocks, lifelock, timing...]

7.1 Example: internal combustion engine



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.

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 because it leaves programmers no freedom to make trade-offs between programmability, utilization of resources and silicon area.

7.2 Old tools revived

Many languages and tools that were developed based on a single model start to embrace other models [12]. 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.

7.3 New hybrid tools

However also new frameworks have been developed that took multiple paradigms into their design from the beginning. A framework is a software architecture that specifies the possible interactions of componenets, provides a set of services that components can use and may have a set of formal properties for the system.

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

7.4 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.[12]. 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.

7.5 HyCharts

7.6 Simulink

Existing (commercial) tool. [22] calls it a framework. A modelling and simulation environment for continuous-time dynamic systems with discrete events.

7.7 *charts

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

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

7.8 ForSyDe

An interesting method has been development by Sander and Jantsch[29, 30]. In their model events are totally-ordered by their tags. Every signal has the same set of tags. Events with the same tag are processed synchronously. There is a special value $\perp$ ("bottom") to indicate the absence of an event. These are necessary to establish a total ordering among events. A system is modelled by means of concurrent processes; it is a model based on the synchronous-assumption (see Chapter 6.4.7).

Lu[24] 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.

[3] 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.

7.9 Moses framework

7.10 Solar/Music

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

7.11 The SPI Workbench/Funstate

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

Chapter 8

Discussion of hybrid models

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

Because of the complexity of most systems, optimal manual decisions are sometimes not feasible. There are simply to many possibilities to consider. To use all of to 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 an there are various methods for systematic design space exploration cite(cassidy).

8.1 Hierarchy/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.'[12]

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[22].

'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 as subsystem specified using one MoC within a system specified using another. The wormhole must obey the semantics of the outer MoC at its boundaries and the semantics of the inner MoC internally. Information hiding insulates the outer MoC from the inner one.' [4]

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

8.2 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 simulate the the designer by taking so long. A model that heavily uses entities communicating through signals will burden the discrete-event scheduler and bog down the simulation. Thus, a specification built on discrete-event semantics is a poor match for implementation is software.

By contrast, VHDL that is written as sequential code runs relatively quickly but may not translate well into hardware. The same goes for C: it runs very quickly and is well suited for software, but not for specifying hardware. [Guus: Expand this. What are the problems with C?]

Dataflow and finite-state models of computation have been shown to be reasonably re-targettable. Hierarchical FMS such as Statecharts can be used effectively to design hardware or software. It has also be shown that a single dataflow specification can be partitioned for combined hardware and software implementation.'[4] [Guus: very interesting, expand this].

It is possible to conclude that the two issues found in this paper are closely related. If you want a single specification language you'll loose in the paradigm-shift. There is no (efficient) language that allows both control- and dataflow types to be presented and generate efficient

code for it. On the other had, 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.

[18] 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."

However, 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. A thing to investigate would be for what types of systems this kind of Co-Design approaches are suitable, and for which not.

The difference made between heterogeneous and homogenous modelling seems to be a bit to blunt. Some authors make it look like heterogenous modelling is the only answer possible because that's the only type of method that allows all kinds of systems to be modelled. However, some homogenous modelling tools allow for a subset of applications to be modelled.

8.3 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 all.[18] 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."

Chapter 9

Conclusions

Most Co-Design tools make use of an internal representation for the refinement of the input specification and architectures[16]. Many such internal representations exist. They are all based on one or more Computational Models.

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 of 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 investigate, 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] 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] I. D. Bates, E. G. Chester, and D. J. Kinniment. A statechart based HW/SW codesign system. In Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES-99), pages 162–166. ACMPress, May 1999.

[3] 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.

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

[12] 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.

[13] 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. ACMPress, Apr. 2001.

[14] D. Gajski and F. Vahid. Specification and design of embedded software-hardware systems. IEEE Design & Test of Computers, 12(1), 1995.

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

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

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

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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. ACMPress, Apr. 2001.

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

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

[34] D. E. Thomas, J. K. Adams, and H. Schmit. A model and methodology for hardware-software codesign. IEEE Design & Test of Computers, Sept. 1993.

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

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

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