

CHAPTER 1

Introduction

In the recent years, the complexity of modern embedded systems is increasing rapidly. Embedded systems exploited only a single or only a few micro-controllers in their infancy. Nowadays, these highly complex systems feature diverse multi-core processor architectures. Even more, embedded systems may exploit SoC¹ or NoC² platforms which arose lately on the market. Thus, today's embedded systems are truly heterogeneous systems with strict real-time requirements. Hence, the term *cyber physical systems* is gaining more and more momentum in favour of the term *embedded systems* for those architectures, because a system embedded in a physical world requires time as an inherent system property. Cyber physical systems demand for a design methodology which is aware of time in both hardware and software domains.

Traditional hardware / software co-design approaches require a lot of effort to get the different counterparts of the system working together seamlessly. Usually, these heterogeneous systems are specified by hardware description languages (e.g. VHDL³) and software languages (e.g. C), which are not well aware of their hardware or software counterpart. Even more, established software languages such as C, C++, or Java, which are usually exploited within embedded systems, have no appropriate means for defining time within. Thus, the system level design of such systems with traditional hardware and software languages is an error-prone process, which is bound tightly to the exploited architecture.

Intellectual property (e.g. VHDL source code) is often manually optimized for specific target architectures to increase performance and, thus, cannot be easily deployed into different architectures. A new target ar-

¹ System on a chip

² Network on a chip

³ Very High Speed Integrated Circuit Hardware Description Language

chitecture often requires a restructure of the system partitioning into its hardware and its software components to meet application requirements. For instance, software algorithms have to be shifted to hardware to meet timing constraints and, thus, the software which is presumably written in C source code has to be transformed to an appropriate hardware module specified using a hardware description language such as VHDL. Taken together, these common hardware / software co-design approaches are not suitable any more for the current demands of cyber physical systems. Therefore, abstraction of system level design has been raised by exploiting MoCs⁴ within the design flow.

1.1 Models of Computation

MoCs allow us to step back from the platform-dependent implementation of a cyber physical system to a more abstract view of the system and its designated behaviour. By exploiting a diversity of MoCs with each having specific properties, different system level design problems may be tackled. For instance, an algorithm which will be modelled with a Petri net may be verified for the Petri net property of liveness. This property implies that the algorithm always terminates and, thus, no deadlock will occur during the execution. Another benefit of using MoCs is the reusability: Once domain specific expertise is gained within a MoC, it will not be lost across different target-platforms. Hence, an embedded system level design flow should be built upon exploitation of diverse MoCs, but a solely use of a single MoC will be extremely beneficial compared to a common hardware / software co-design flow based on C and VHDL. Specifically, an exploitation of a timed MoC is the most promising approach regarding the system level design of cyber physical systems.

A lot of scientific effort has been put into the research of MoCs in respect of an embedded system level design flow. In the following, the most prominent ones are shortly discussed and reviewed in terms of their applicability for an hardware / software co-design flow based on MoCs.

ForSyDe [SJ04] is a functional programming framework with synthesizable code generation. The design process starts with an abstract formal specification model of the system which is required to be perfectly synchronous. Synthesizable code generation is a two step process: First, the

⁴ *Models of Computation*

initial abstract model will be successively refined by applying transformations until a more detailed implementable model is reached. Second, this implementation model will be mapped to the target architecture by exploiting hardware / software co-design algorithms like partitioning, resource allocation, and generation of C source code and VHDL source code. However, the modelling fidelity is limited in terms of the initial assumption of perfect synchronicity. This assumption may not be fulfilled by the computational models of a broad application field.

In contrast, Ptolemy II [BLL⁺05] is a genuine multi-MoC design environment. It is a pure simulation and modelling environment for heterogeneous systems exploiting a diversity of MoCs (e.g. FSM⁵, DE⁶, CT⁷, CSP⁸, or KPN⁹). In [FLN06], limited hardware synthesis of Ptolemy II model is introduced, which is primarily suitable for DSP applications because transformations of models are only allowed from the SR¹⁰ domain to VHDL.

HetSC [HV07] ease the software implementation of heterogeneous MoC designs. The employed models may be manually refined into synthesizable SystemC source code. Hence, it does not feature an automatic transformation of the whole MoC. However, together with ANDRES [HV07], it is strong in both, software transformation and hardware synthesis, but different MoCs will be exploited for the hardware and the software parts of the system under consideration. Thus, domain specific knowledge aggregated within one of these MoCs may not be easily transferred to the other. Therefore, MoC specific properties may be lost during design space exploration, model refinement, and especially during hardware / software co-design.

A comprehensive list of platform-based industrial and academic system level design tools is given in [DP06]. This survey results in the fact that, firstly, only a small subset of tools consider timed MoCs, and secondly, these cover only a small subset of the tasks necessary for synthesis. Thus, no complete system level design flow exploiting a single, timed MoC is covered.

In this thesis, a systematic approach of integrating diverse MoCs into a system level design flow in terms of model transformations will be pre-

⁵ *Finite State Machine*

⁶ *Discrete Event*

⁷ *Continuous Time*

⁸ *Communicating Sequential Processes*

⁹ *Kahn Process Network*

¹⁰ *Synchronous Reactive*

sented. Then, the advocated hardware/software co-design flow will exploit only a single timed MoC to accomplish hardware synthesis and software implementation. Therefore, the well-known timed DEVS¹¹ MoC from Zeigler et al. [ZKP00] will be exploited as a foundation for the system level design flow.

1.2 DEVS Model of Computation

The DEVS formalism is a strong mathematical foundation for specifying hierarchical, concurrently executed formal models. It covers both time-discrete and time-continuous models. However, this thesis focus on the time-discrete operation only.

A lot of scientific effort has been put into research the DEVS MoC. For instance, the generation of a reachability graph is given in [HZ09]. In [Wai09], plenty of real-world simulation application examples are given for the DEVS formalism which has its roots in the simulation domain. Interoperability with another other MoC (i.e. UML¹²) in terms of model transformations is presented in [RMZ⁺07] for UML state charts or in [SV11] for UML class diagrams, just to name a few.

However, the original time-discrete DEVS MoC has some inherent properties which impede direct synthesis of the models. For instance, an output port may emit an unbound amount of events within a single point in time. Such a behaviour is clearly not synthesizable because it would possibly require an infinite amount of memory at the sink to process these emitted events. Thus, the original formalism is refined in this thesis and presented in a much clearer behaviour definition named SynDEVS¹³ MoC which allows synthesis.

The refined version was carefully constructed with the following basic principles in mind:

- Substantiate the DEVS MoC behaviour to allow synthesis.
- If a DEVS model is synthesizable in terms of not exploiting the non-synthesizable DEVS properties, then it has to be possible to transform the original model easily into an equivalent synthesizable SynDEVS model.

¹¹ *Discrete Event System Specification*

¹² *Unified Modelling Language*

¹³ *Synthesizable DEVS*

- Preserving as much as possible of the original DEVS properties is of utmost importance.

Additionally, the transformation of SynDEVS models to hardware and software will be detailed in this thesis. Therefore, the models of the timed SynDEVS MoC will be transformed into synthesizable VHDL source code and embeddable C++ source code. By doing so, the system may be designed in an abstract manner by using MoCs, but existing synthesis software and compilers supplied by the target platform vendors may be still exploited. Hence, such an approach allows for an easy use of SynDEVS MoC within an existing embedded system design flow. Thus, there is no need of re-modelling the whole system with SynDEVS but instead the MoC may be only partially exploited where it is beneficial for the system. By doing so, once gained platform-dependent knowledge (e.g. a well-evaluated network stack source code library) is not lost. Thus, the transition to a solely use of MoCs within the design flow may be performed over time. This reflects the design process of embedded systems which is in general an evolutionary process.

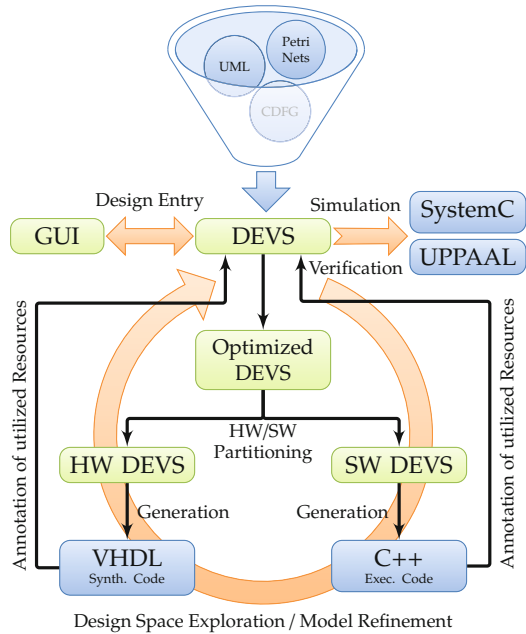
1.3 Hardware / Software Co-Design Flow

In this thesis, a hardware / software co-design flow based on the DEVS MoC will be presented. Figure 1.1 depicts this abstract system level design flow. The figure will be reused at each chapter relevant to the design flow, only highlighting the topics covered in the chapter.

The abstract DEVS-based design flow may be exploited to create implementable models. Thus, automatic transformations down to abstraction levels covering VHDL synthesis and embedded systems C++ executable source code generation are included. The designer may supply DEVS models either by creating them with a developed GUI¹⁴ and/or by the (automated) transformation of different domain-specific MoCs into DEVS models. Simulation and Verification may be exploited to test models for correct behaviour. Therefore, the DEVS models may be simulated with a non-introspective SystemC™ kernel extension. Verification may be done by applying the work from Madlener et al. [MWH10] which exploits an automatic model transformation into UPPAAL model-checker verification models.

¹⁴ *Graphical User Interface*

Figure 1.1 DEVS design flow covered by this thesis, highlighting different topics in terms of design entry, simulation, and hardware / software co-design. Throughout the thesis, at the beginning of each chapter relevant to the design flow, this figure will be shown, only highlighting the topics covered in the chapter.



At any time during the system level design, the designer may execute the following fully automated design flow process: First, the input model covering the whole system is optimized regarding different timing aspects and the model's component hierarchy is flattened. Second, the flattened and optimized system level model is split into a single SynDEVS model each for the hardware and software generation processes of the design flow. Communication interfaces in between the hardware and software instances may be automatically generated. Design space exploration and model refinement will be supported in terms of back-annotating the utilized resources to the initial model. Every intermediate model of the design flow will be a SynDEVS model, too. Thus, the designer has full control over each step in the design flow to further refine the model. This approach of a fully transparent design flow is very beneficial for more sophisticated design space exploration methods not covered in this thesis. These may be easily introduced by writing further (automated) tools which may operate on these intermediate SynDEVS models.

1.4 Remainder of this Thesis

The original DEVS formalism will be reviewed and discussed in terms of synthesis in Chapter 2. A revised version of the formalism in order to support synthesis will be introduced. This SynDEVS MoC will be defined and reviewed in detail in terms of synthesis aspects. Additionally, a visual notation of the SynDEVS model will be introduced.

Chapter 3 will present modelling and validation methods of SynDEVS MoC. First, it will be discussed how diverse MoCs may be transformed into SynDEVS MoC. Second, the validation of SynDEVS models in terms of SystemC simulation will be presented.

Afterwards, the main contribution of this thesis, the advocated hardware / software co-design flow based on SynDEVS MoC will be introduced. First, Hardware / Software partitioning will be detailed. Second, both model transformations to synthesizable VHDL source code and embeddable C++ source code will be thoroughly discussed. It will be detailed how SynDEVS timing annotations are implemented within the resulting VHDL and C++ source codes. An optimization algorithm will be introduced which generates a SynDEVS model with equivalent behaviour but with improved timing constraints. Furthermore, integration of legacy C / C++ source code software libraries into the SynDEVS software instances will be detailed.

Chapter 5 will shortly introduce the developed GUI for the SynDEVS MoC. The complete design flow may be centrally controlled using the GUI. Furthermore, all intermediate models of the design flow may be reviewed within the GUI.

In the following Chapter 6, three case studies will be presented. Each case study highlights different parts of the design flow: A SynDEVS model of a DVI¹⁵ controller with strict real-time requirements will be discussed highlighting the hardware part of the design flow. Then, this DVI controller will be extended to implement a network-based Pong game in order to demonstrate the software part of the design flow with legacy source code integration. Afterwards, a flexible cryptographic accelerator will be introduced to show the benefits of partially exploiting SynDEVS MoC within an existing design.

Finally, Chapter 7 concludes this thesis. Important points will be summarized and an outlook for further research challenges will be given.

¹⁵ *Digital Visual Interface*

SynDEVS Co-Design Flow

A Hardware / Software Co-Design Flow Based on the
Discrete Event System Specification Model of
Computation

Molter, H.G.

2012, XXVI, 198 p. 57 illus., 45 illus. in color., Softcover

ISBN: 978-3-658-00396-8