Specification and Modeling of HW/SW CO-Design for Heterogeneous Embedded Systems

Adnan Shaout, Ali H. El-Mousa., and Khalid Mattar

***Abstract*— The complexity of designing embedded systems is constantly increasing. Some of the factors contributing to the increase in complexity are: increasing complexities of hardware and software, increased pressure to deliver full-featured products with reduced time-to-market, and the fact that more embedded systems are using heterogeneous architectures consisting of dedicated hardware components (ASIC) and software running on processors. This ongoing increase in complexities motivates the usage of high-level system design approaches such as System Level Design tools and methodologies. In System Level Design, specification languages are used to build high level models of the entire system, at the System Level, to allow fast design space exploration. Models of Computations (MoC) are used as the underlying formal representation of a system. This paper investigates the co-design approach and its design activities. Specifically, specification and modeling computation process are investigated and evaluated. Popular models of computations are described and compared. Various specification languages for designing embedded are described and compared..**

***Index Terms*—System Level Design, Hardware / Software co-design, Heterogeneous embedded systems.**

I. Introduction

A. Embedded Systems

Embedded systems are special-purpose systems which are typically embedded within larger units providing a dedicated service to that unit [1]. In most embedded systems, a function-specific software application is provided by the product manufacturer, and end-users have limited access to altering the application running on the system. Embedded systems are found in consumer electronics products (i.e. cell phones, PDAs, microwaves, etc), transport control systems (i.e. cars, trains, and aircrafts), plant control systems and defense systems. These are only few examples of embedded systems.

Vahid et al. [2] describe the characteristic of embedded systems that differentiate them from other digital systems:

- *Single-functioned.* Embedded systems repeatedly perform a specific function.
- *Reactive and real time.* Many embedded systems, especially in the control domain, are reactive systems and must continually react to changes in the environment and meet timings constraints without delay.
- *Tightly constrained.* Embedded systems have tight constraints on design metrics. For example, embedded systems must have minimum design costs, must have small form factors and consume minimum power, especially for portable systems, must meet real time requirements, must be safe and reliable, and must have short time-to-market cycle.

The majority of embedded systems are implemented using heterogeneous systems consisting of dedicated parts and programmable processors [3]. A typical heterogeneous system consists of: dedicated hardware parts (ASIC), programmable processors such as microprocessor and ASIP[1] components (i.e. DSP and microcontrollers), memory for data and code, peripherals such A/D, D/A and I/O units, and buses connecting the above components.

Heterogeneous systems are implemented on a single board or a single chip. In single-board systems, processors, AISC components, memories and peripheral are integrated on a single board. Single-chip systems integrate on one ASIC processor cores, dedicated parts, memories and peripherals. Compared to single-board systems, single-chip system provide increased performance and reduced power consumption. However, single-chip systems are more complex which makes debugging these systems much harder.

Traditionally, hardware synthesis tools (logic synthesis and behavior synthesis) have been used to increase productivity. However, hardware synthesis is not sufficient since embedded systems use more software content [4]. In addition, hardware synthesis methods focus on designing a single hardware chip, where more embedded systems are using heterogeneous architectures.

The complexities in designing embedded systems motivate the need for using more efficient tools and design methodologies. *System Level Design* is a methodology to help address these complexities, and enable SoC designs.

II. System Level Design

System Level Design is concerned with addressing the challenges encountered in designing heterogeneous embedded systems. In System Level Design, complexities are managed by (1) starting the design process at the highest level

Manuscript received March 23, 2009.

Adnan Shaout is with the University of Michigan-Dearborn, the Electrical and Computer Engineering Department, shaout@umich.edu.

Ali H. El-Mousa is with the Computer Engineering Department, Faculty of Engineering & Technology, University of Jordan, Amman, Jordan. (Phone: ++96265355000 ext. 23000, fax: ++96265355588, e-mail: elmousa@ju.edu.jo)

Khalid Mattar is with the University of Michigan-Dearborn, the Electrical and Computer Engineering Department.

[1] Application Specific Instruction-Set Processor

of abstraction (System Level), (2) utilizing automated design methodologies to enable step-wise refinements during the design process (3) reusing Intellectual Property (IP) components when feasible [5].

The goal of System Level Design is to implement System Level specification on target architecture by refining the specification into a set of target-specific specifications.

Designing at a higher level of abstraction reduces the number of components with which the designer has to deal with, and thus increasing design productivity. This paradigm shift in design requires methodologies and automated tools to support design at higher levels abstractions.

The use of IP components allows the designer to use pre-designed components, instead of designing from scratch. IP use is gaining recognition in the design community. In order to fully leverage using IP components in the design of heterogeneous embedded system, IP components must allow seamless integration into the overall system, and design tools must support using IP components [6].

System Level Design is different than behavior synthesis since it (behavior synthesis) maps an algorithm into a ASIC, while System Level Design maps a high-level abstract specification model of an *entire* system onto a target architecture [4].

A. System level design approaches

There are three main system level design approaches: hardware/software co-design, platform-based design and component-based design [7].

- *Hardware/Software co-design* (also referred to *system synthesis*) is a top-down approach. Starts with system behavior, and generates the architecture from the behavior. It is performed by gradually adding implementation details to the design.
- *Platform-based design.* Rather than generating the architecture from the system behavior as in co-design, platform-based design maps the system behavior to predefined system architecture. An example of platform-based design is shown in [8].
- *Component-based design* is a bottom-up approach. In order to produce the predefined platform, it assembles existing heterogeneous components by inserting wrappers between these components. An example of component-based design is described in [9].

III. HARDWARE / SOFTWARE CO-DESIGN

Hardware/Software co-design can be defined as the cooperative design of hardware[2] and software[3] in order to achieve system-level objectives (functionality & constraints) by exploiting the synergism of hardware and software [6],[7]. Hardware/Software co-design research focuses on presenting a unified view of hardware and software, and the development of synthesis tools and simulators to address the problem of designing heterogeneous systems. While hardware implementation provides higher performance, software implementation is more cost effective and flexible since software can be reused and modified. The choice of hardware versus software in co-design is a trade-off among various design metrics like performance, cost, flexibility and time-to-market. This trade-off represents the optimization aspect of co-design. Fig. 1 shows the flow of a typical Hardware / Software co-design system.

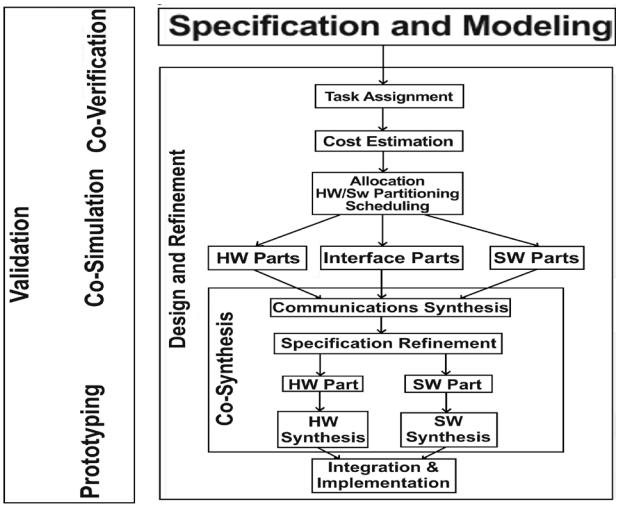

Fig. 1: Flow of a typical co-design system

Generally, Hardware / Software co-design consists of the following activities: *specification and modeling*, *design* and *validation* [6].

A. Specification and modeling

This is the first step in the co-design process. The system behavior at the system level is captured during the specification step [3]. Section IV provides details about specification and modeling, including Models of Computation.

B. Design and refinement

The design process follows a step-wise refinement approach using several steps to transform a specification into an implementation. Niemann [3] and O'Nils [6] define the following design steps:

- *Tasks assignment:* The system specification is divided into a set of tasks/basic blocks that perform the system functionality [3].
- *Cost estimation:* This step estimates cost parameters for implementing the system's basic blocks (output of task assignment) in hardware or software. Examples of hardware cost parameters are: gate count, chip area and power consumption, where execution time, code size and required code memory are examples of software cost parameters. Cost estimates are used to assist in making design decision to decrease the number of design iterations [3].
- *Allocation:* This step maps functional specification into a given architecture by determining the type and number of processing components required to implement the system's functionality. To make the allocation process manageable, co-design systems normally impose restrictions on target architectures. For example, allocation may be limited to a certain pre-defined components [10].
- *Hardware/Software partitioning:* This step partitions the specification into two parts: (1) a part that will be implemented in hardware and (2) a part that will be implemented in software.

[2] Hardware refers to dedicated hardware components (ASIC).
[3] Software refers to software executing on processor or ASIP

- *Scheduling:* This step is concerned with scheduling the tasks assigned to processors. If tasks information (i.e. execution time, deadline, delay) are known, scheduling is done statically at design time. Otherwise, scheduling is done dynamically at run time (i.e. using Real Time OS – RTOS). De Michell et al. [1] provide an overview of techniques and algorithms to address the scheduling problem.
- *Co-synthesis*: Niemann [3] classifies several design steps as part of co-synthesis:

  1) Communication synthesis: Implementing the partitioned system on heterogeneous target architecture requires interfacing between the ASIC components (HW) and the processors (SW) communication between the ASIC(s) and the processors. This is accomplished in communication synthesis step.

  2) *Specification refinement:* Once the system is partitioned into hardware and software, and the communication interfaces are defined, the system specification is refined into hardware specifications and software specifications, which include communication methods to allow interfacing between the hardware and software components.

  3) Hardware synthesis: AISC components are synthesized using behavior (high-level) synthesis and logic synthesis methods. Hardware synthesis is a mature field due to the extensive research done in this field. Details about hardware synthesis methods are provided in [11] and [12]

  4) Software synthesis: This step is related to generating, from high level specification, C or assembly code for the processor(s) that will be executing the software part of the heterogeneous system. Edwards et al. [10] provides an overview of software synthesis techniques.

C. Validation

Informally, validation is defined as the process of determining that the design, at different levels of abstractions, is correct. The validation of hardware/software systems is referred to as *co-validation*. Methods for co-validations are [9],[13] :

- *Formal verification* is the process of mathematically checking that the system behavior satisfies a specific property. Formal verification can be done at the specification or the implementation level. For example, formal verification can be used to check the presence of a deadlock condition in the specification model of a system. At the implementation level, formal verification can be used to check whether a hardware component correctly implements a given Finite State Machine (FSM). For heterogeneous systems (i.e. composed of ASIC components and software components), formal verification is called *co-verification*.
- *Simulation* validates that a system is functioning as intended by simulating a small set of inputs. Simulation of heterogeneous embedded systems requires simulating both hardware and software simultaneously, which is more complex than simulating hardware or software separately. Simulation of heterogeneous systems is referred to as *co-simulation*. A comparison of co-simulation methods is presented in [10].

IV. Specification and Modeling

Specification is the starting point of the co-design process, where the designer specifies the system's specification without specifying the implementations. Languages are used to capture system specifications. Modeling is the process of conceptualizing and refining the specifications. A model is different from the language used to specify the system. A model is a conceptual notation that describes the desired system behavior, while a language captured that concept in a concrete format. A model can be captured in a variety of languages, while a language can capture a variety of models [2].

In order to design systems that meet performance, cost and reliability requirements, the design process need to be based on formal computational models to enable step-wise refinements from specification to implementation during the design process [13]. Co-design tools use specification languages as their input. In order to allow refinement during the design process, the initial specifications are transformed to intermediate forms based on the Model of Computation (MOC) [14] used by the co-design systems. Two approaches are used for system specification, *homogeneous modeling* and *heterogeneous modeling* [6],[17]:

- *Homogeneous modeling* uses one specification language for specifying both hardware and software components of a heterogeneous system. The typical task of a co-design system using homogeneous approach is to analyze and split the initial specification into hardware and software parts. The key challenge in this approach is the mapping of high level concepts used in the initial specification onto low level languages (i.e. C and VHDL) to represent hardware/software parts. To address this challenge, most co-design tools that use the homogeneous modeling approach start with a low level specification language in order to reduce the gap between the system specification and the hardware/software models. For example, Lycos [15] uses a C-like language called C^x and Vulcan uses another C-like language called HardwareC. Only few co-design tools start with a high level specification language. For example, Polis [16] uses Esterel [17] for its specification language.
- *Heterogeneous modeling* uses specific languages for hardware (e.g. VDHL), and software (e.g. C). Heterogeneous modeling allows simple mapping to hardware and software, but this approach makes validation and interfacing much more difficult. CoWare [18] is an example of a co-design methodology that uses heterogeneous modeling.

A. Models of computation

A computational model is a conceptual formal notation that describes the system behavior [2]. Ideally, a Model of Computation (MOC) should comprehend *concurrency*, *sequential behavior* and *communication methods* [13]. Co-design systems use computational models as the underlying formal representation of a system. A variety of Models of Computation have been developed to represent heterogeneous systems. Researchers have classified MOCs according to different criteria.

Gajski [15] classifies Models of Computations according to their orientation into five classes [3]:

- *State-oriented models* use states to describe systems and events trigger transition between states.
- *Activity oriented models* do not use states for describing systems, but instead they use data or control activities.
- *Structural oriented models* are used to describe the physical aspects of systems. Examples are: block diagrams, RT netlists.
- *Data oriented models* describe the relations between data which are used by the systems. Entity Relationship Diagram (ERD) is an example of Data oriented models.
- *Heterogeneous models* merge features of different models into a heterogeneous model. Examples of heterogeneous models are Program State Machine (PSM), Control/Data flow graphs (CDFG).

In addition to the classes described above, Bosman [14] proposes a *time-oriented* class to capture the timing aspect of MOCs. Jantsch et al. [19] group MOCs based on their timing abstractions. They define the following groups of MOCs: c*ontinuous time models, discrete time models, synchronous models and un-timed models.* Continuous and discrete time models use events with a time stamp. In the case of continuous time models, time stamps correspond to a set of real numbers, while the time stamps correspond to a set of integer numbers in the case of discrete time models. Synchronous models are based on the *synchrony hypothesis*[4].

Cortes et al. [13] group MOCs based on common characteristics and the original model they are based on. The following is an overview of common MOCs based on the work in [13] and [14].

1) Finite State Machines (FSM)

The FSM model consists of a set of states, a set of inputs, a set of outputs, an output function , and a next-state function [20]. A system is described as set of states and input values can trigger a transition from one state to another. FSMs are commonly used for modeling control-flow dominated systems. The main disadvantage of FSMs is the exponential growth of the number of the states as the system complexity rises due the lack of hieracay and conccurency. To address the limitations of the classical FSM, researches have proposed several derivates of the classical FSM. Some of these extensions are described below.

- *SOLAR* [21] is based on the Extended FSM model (EFSM), which can support heirachy and conccurency. In addtion, SOLAR supports high level communication concepts inlcuding channels and global varaibles. It is used to represent high-level concepts in control-flow dominated systems, and it is mainly suited for synthesis purposes. The model provides an intermediate format that allows hardware/software designs at the system-level to be synthesized.
- *Hierarchical Concurrent FSM (HCFSM)* [3] solve the drawbacks of FSMs by decomposing states into a set of sub-states. These sub-states can be concurrent sub-states communicating via global variables. Therefore, HCFSMs supports hierarchy and concurrency. Statecharts is a graphical state machine language designed to capture the HCFSM MOC [2]. The communication mechanism in statecharts is instantaneous broadcast, where the receiver proceeds immediately in response to the sender message. The HCFSM model is suitable for control oriented / real time systems.
- *Codesign Finite State Machine (CFSM)* [16],[25] adds concurrency and hierarchy to the classical FSM, and can be used to model both hardware and software. It is commonly used for modeling control-flow dominated systems. The communication primitive between CFSMs is called an event, and the behavior of the system is defined as sequences of events. CFSMs are widely used as intermediate forms in co-design systems to map high-level languages, used to capture specifications, into CFSMs.

2) Discrete-Event Systems

In a Discrete Event system, the occurrence of discrete asynchronous events triggers the transitioning from one state to another. An event is defined as an instantaneous action, and has a time stamp representing when the event took place. Events are sorted globally according to their time of arrival. A signal is defined as set of events, and it is the main method of communication between processes [13]. Discrete Event modeling is often used for hardware simulation. For example, both Verilog and VHDL use Discrete Event modeling as the underlying Model of Computation [10]. Discrete Event modeling is expensive since it requires sorting all events according to their time stamp.

3) Petri Nets

Petri Nets are widely used for modeling systems. Petri Nets consist of places, tokens and transitions, where tokens are stored in places. Firing a transition causes tokens to be produces and consumed. Petri Nets supports concurrency and is asynchronous; however, they lack the ability to model hierarchy. Therefore, it can be difficult to use Petri Nets to model complex systems due to its lack of hierarchy. Variations of Petri Nets have been devised to address the lack of hierarchy. For example, the Hierarchical Petri Nets (HPNs) proposed by Dittrich [22]

- *Hierarchical Petri Nets* (*HPNs*) supports hierarchy in addition to maintaining the major Petri Nets features such as concurrency and asynchronously. HPNs use Bipartite [5] directed graphs as the underlying model. HPNs are suitable for modeling complex systems since they support both concurrency and hierarchy.

4) Data Flow Graphs

In Data Flow Graph (DFG), systems are specified using a directed graph where nodes (actors) represent inputs, outputs and operations, and edges represent data paths between nodes [3]. The main usage of Data Flow is for modeling data flow dominated systems. Computations are executed only where the operands are available. Communications between processes is done via unbounded FIFO buffering scheme [13]. Data Flow models support hierarchy since the nodes can represent complex functions or another Data Flow [6],[13].

Several variations of Data Flow Graphs have been proposed in the literature such as Synchronous Data Flow (SDF) and Asynchronous Data Flow (ADF) [22]. In SDF, a fixed number of tokens are consumed, where in ADF the number of tokens consumed is variable. Lee[23] provides an

[4] Outputs are produced instantly in reaction to inputs, and no observable delay in the outputs.

[5] A graph where the set of vertices can be divided into two disjoint sets *U* and *V* such that no edge has both end-points in the same set.

overview of Data flow models and its variations.

5) Synchronous/Reactive Models

Synchronous modeling is based on the synchrony hypothesis, which states that outputs are produced instantly in reaction to inputs and there is no observable delay in the outputs [14]. Synchronous models are used for modeling reactive real time systems. Cortes in [13] mentions two styles for modeling reactive real time systems: multiple clocked recurrent systems (MCRS) which is suitable for data dominates real time systems and state base formalisms which is suitable for control dominated real time systems. Synchronous languages such as Esterel [17] is used for capturing Synchronous/Reactive Model of Computation [13].

6) Heterogeneous Models

Heterogeneous Models combine features of different models of computation. Two examples of heterogeneous models are presented.

- *Programming languages* [15] provide a heterogonous model that can support data, activity and control modeling. Two types of programming languages: *imperative* such as C, and *declarative* languages such as LISP and PROLOG. In imperative languages, statements are executed in the same order specified in the specification. On the other hand, execution order is not explicitly specified in declarative languages since the sequence of execution is based on a set of logic rules or functions. The main disadvantage of using programming languages for modeling is that most languages do not have special constructs to specify a system's state [3]
- *Program State Machine (PSM)* is a merger between HCFSM and programming languages. A PSM model uses a programming language to capture a state's actions [15]. A PSM model supports hierarchy and concurrency inherited from HCFSM. The *Spec Charts* language, which was designed as an extension to VHDL, is capable of capturing the PSM model. The *Spec C* is another language capable of capturing the PSM model. *Spec C* was designed as an extension to C [2].

Table I: Comparison of Models of Computation [14] [13]

MOC	Origin MOC	Main Application	Clock Mechanism	Orientation	Time	Comm. Method	Hierarchy
SOLAR	FSM	Control Oriented	Synch	State	No Explicit Timings	Remote Procedure Call	Y
HCSFM/ State Charts	FSM	Control Oriented/ Reactive Real time	Synch	State	Min/Max Time spent in State	Instant Broadcast	Y
CFSM	FSM	Control Oriented	Async	State	Events w/t time stamp	Event Broadcast	Y
Discreet Event	N/A	Real time	Synch	Timed	Globally sorted events w/t time stamp	Wired Signals	N
HPN	Petri Net	Distributed	Async	Activity	No Explicit Timings	N/A	Y
SDF	DFG	Signal processing	Synch	Activity	No Explicit Timings	Unbounded FIFO	Y
ADF	DFC	Data Oriented	Async	Activity	No Explicit Timings	Bounded FIFO	Y

B. Comparison of Models of Computation

A comparison of various Models of Computation is presented by Bosman [14], and Cortes et al. [13]. Each author compares MOCs according to certain criteria. Table I compares MOCs based on the work done in [14] and [13].

C. Specification languages

The goal of a specification language is to describe the intended functionality of a system non-ambiguously. A large number of specification languages are currently being used in embedded system design since there is no language that is the best for all applications [3]. Below is a brief overview of the widely used specification languages [2, 6] :

1) Formal description languages

Examples of formal languages are *LOTOS* and *SDL*.

- *LOTOS* is based on process algebra, and used for the specification of concurrent and distributed system.
- *SDL* used for specifying distributed real time systems, and based on extended FSM.

2) Real time languages

Esterel & *StateCharts* are examples of real time languages.

- *Esterel* is a synchronous programming language based on the synchrony hypothesis. It is used for specifying real time reactive systems. Esterel is based on FSM, with constructs to support hierarchy and concurrency.
- *StateCharts* is graphical specification language used for specifying reactive system. StateCharts extend FSM by supporting hierarchy, concurrency and synchronization.

3) Hardware Description Languages (HDL)

Commonly used HDL are *VHDL*, *Verilog* and HardwareC.

- *VHDL* is IEEE standardized HW description language.
- *Verilog* is another hardware description language, which has been standardized by IEEE.
- *HardwareC* is a C based language designed for hardware synthesis. It extends C by supporting structural hierarchy, concurrency, communication and synchronization.

4) System Level Design Languages (SLDL)

System Level Design Languages (SLDL) are used to capture specification and model embedded system at the system abstraction level. With the increased time-to-market pressure, and to enable SoC designs, SLDS need to be able to specify and model all aspects of the system at higher abstraction level (at the System Level). This will allow early design space exploration to evaluate various design alternatives early in the design process. Most current SLDLs lack built-in support for specifying and modeling ALL aspects of a heterogeneous embedded system at the System Level. Some of these deficiencies are lack of support for:

- RTOS modeling at the System Level. This is important for modeling real time embedded system, and determining if the scheduling policy will meet time constraints and deadline at the System Level before committing to a specific RTOS implementation.
- Composing Heterogeneous models with multiple MoCs.
- Estimating power consumption at the System Level.

Examples of SLDL are SpecC and SystemC.

- *SpecC* [20] is system level design language based on ANSI C. It was developed at the University of California, Irvine to improve traditional HDL languages such as VHDL. The SpecC language models systems as a hierarchal network of behaviors and channels [3]. SpecC supports behavior and structural hierarchy, concurrency, state transition, exception handling, timing aspects and synchronization. Built on the

SpecC language is the SpecC design methodology.

- *SystemC* [24] is a C++ library based language designed the OpenSystemC Initiative (OCSI) group to improve traditional HDL languages. SystemC has common features with SpecC, and is widely used in system level modeling.

D. Requirements for specification languages

Gajski in [20] and Niemann in [3] describe the requirements for specification languages:

- *Hierarchy* is an important feature of a specification language. Two types of hierarchy: *(1) behavior hierarchy* which allows a behavior to be decomposed of sub-behaviors, *(2*) *structural behavior* which allows a system to be specified as a set of interconnected components, where these components can be specified as sub-components as well.
- *State transition* explicit support of state transition is important since state transition is important for modeling control and reactive embedded systems.
- *Concurrency* a large number of embedded systems consist of tasks that are working concurrently. Therefore, concurrency is a necessary feature of a specification language.
- *Synchronization* is needed when concurrent parts of a system exchange data.
- *Exception handling* exceptions such as reset and interrupts often occur in embedded systems. When an interrupt occurs, the system has to transition to a new state to handle the interrupt. Once the interrupt is serviced, the system has to go back to point prior to interrupt. Specification languages should be able to model exceptions.
- *Timing* is an important aspect of specifying real time embedded systems. Two timing aspects have to be specified when dealing with embedded systems: Functional timing which represents the time consumed for executing a behavior, and timing constraints which represent a range of time for executing a behavior.
- *Formal verification* is desirable for specification languages since it provides a mechanism to verify the specification using formal mathematical methods.
- *Support for RTOS modeling* is important for the specification of real time systems that will use a RTOS to implement dynamic scheduling. The ability to model the dynamic behavior of system, via RTOS modeling, during the specification phase provides the means to assess the real time behavior of the system early in the design process.

Table II shows a comparison of different well known specification languages.

Table II: Comparison of specification languages

	Formal Languages		Real-time Languages		HDL			SLDL	
	LOTOS	SDL	Esterel	State Charts	VHDL	Verilog	HardwareC	SpecC	SystemC
Structural Hierarchy	F	F	F	N	F	F	F	F	F
Behavior Hierarchy	F	P	F	F	P	P	P	F	F
Concurrency	F	F	F	F	F	F	F	F	F
Synchronization	F	F	F	F	F	F	F	F	F
Exception Handling	F	P	F	F	N	F	N	F	F
Timing	N	P	N	P	F	F	P	F	F
State Transition	F	F	N	F	N	N	N	F	F
Formal Verification	F	F	F	N	N	N	N	N	N
Model Executability	N	F	F	F	F	F	F	F	F
Full Support for RTOS Modeling	N	N	N	N	N	N	N	N	N

(F) Fully supported (P) Partially supported (N) Not supported

REFERENCES

[1] G. De Michell and R. K. Gupta, "Hardware/software co-design," *Proceedings of the IEEE,* vol. 85, no. 3, pp. 349-365, 1997.

[2] F. Vahid and T. Givargis, *Embedded system design: a unified hardware/software introduction*: Wiley, 2002.

[3] R. Niemann, *Hardware/software co-design for data flow dominated embedded systems*. Boston: Kluwer Academic Publishers, 1998.

[4] R. Domer, "System-level Modeling and Design with the SpecC Language," Ph. D. Dissertation, Department of Computer Science, University of Dortmund, Dortmund, Germany, 2000.

[5] R. Dömer, D. Gajski, and J. Zhu, "Specification and Design of Embedded Systems," *it+ ti magazine,* Oldenbourg Verlag, Munich, Germany, no. 3, June 1998.

[6] M. O'Nils, "Specification, Synthesis and Validation of Hardware/Software Interfaces," Doctoral thesis, Department of Electronics, Royal Institute of technology, Stockholm, 1999.

[7] L. Cai, "Estimation and Exploration Automation of System Level Design," Ph. D. Dissertation, Department of Information and Computer Science, University of California, Irvine, 2004.

[8] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS,* vol. 19, no. 12, pp. 1523 - 1543, 2000.

[9] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava, "Component-Based Design Approach for Multicore SoCs," in *Proceedings of 39th Design Automation Conference (DAC02)*, New Orleans, USA, pp. 789 - 794, 2002.

[10] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: formal models, validation, and synthesis," *Proceedings of the IEEE,* vol. 85, no. 3, pp. 366-390, 1997.

[11] R. Camposano and W. H. Wolf, *High-Level VLSI Synthesis*: Kluwer Academic Publishers Norwell, MA, USA, 1991.

[12] S. Devadas, A. Ghosh, and K. Keutzer, *Logic synthesis*: McGraw-Hill, Inc. New York, NY, USA, 1994.

[13] L. A. Cortes, P. Eles, and Z. Peng, "A survey on hardware/software codesign representation models," Dept. of Computer and Information Science, Linköping University, Linköping, Sweden, SAVE Project Report, June 1999.

[14] G. Bosman, I. A. M. Bos, P. G. C. Eussen, and I. R. Lammel, "A Survey of Co-Design Ideas and Methodologies," Master's Thesis at Vrije Universiteit Amsterdam, October 2003, 2003.

[15] D. D. Gajski, J. Zhu, and R. Dömer, "Essential Issues in Codesign," in *Hardware/Software Co-Design: Principles and Practice*, J. Staunstrup and W. Wolf, Eds.: Kluwer Academic Publishers, 1997.

[16] POLIS Group. (2009 April 5). POLIS, A Framework For Hardware-Software Co-Design Of Embedded Systems. Available: http://embedded.eecs.berkeley.edu/research/hsc/.

[17] F. Boussinot, R. de Simone, and V. Ensmp-Cma, "The ESTEREL language," *Proceedings of the IEEE,* vol. 79, no. 9, pp. 1293-1304, 1991.

[18] K. Van Rompaey, D. Verkest, I. Bolsens, H. De Man, and H. Imec, "CoWare-a design environment for heterogeneous hardware/software systems," in *Proceedings of the conference on European design automation* Geneva, Switzerland, pp. 252-257, 1996.

[19] A. Jantsch and I. Sander, "Models of Computation in the Design Process," in *SoC: Next Generation Electronics*, B. M. Al-Hashimi, Ed.: IEE, 2005.

[20] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC, Specification Language and [design] Methodology*: Kluwer Academic, 2000.

[21] A. A. Jerraya and K. O'Brien, "SOLAR: An Intermediate Format for System-Level Modeling and Synthesis," in *Computer Aided Software/Hardware Engineering*, K. Buchenrieder and J. Rozenblit, Eds.: IEEE Press, 1995.

[22] A. Agrawal, "Hardware Modeling and Simulation of Embedded Applications," M.S. Thesis, Department of Electrical Engineering, Vanderbilt University, Nashville, Tennessee, 2002.

[23] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE,* vol. 83, no. 5, pp. 773-801, 1995.

[24] Open SystemC Initiative. (2009, April 5). SystemC. Available: http://www.systemc.org/