An Overview of Methodologies and Tools in the Field of System-Level Design

Vladimir D. Živković[1] and Paul Lieverse[2]

[1] Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Niels Bohrweg 1, 2333 CA Leiden, the Netherlands
`lale@liacs.nl`
`http://www.liacs.nl/~cserc/`

[2] Delft University of Technology, Information Technology and Systems, Delft, the Netherlands
`lieverse@ieee.org`

Abstract. In this paper we present an overview of system level design methodologies and tools. Eight tools and their underlying methodologies are analysed. We give a short description of each of them and point out some of their strengths and weaknesses. We conclude that there still is a lot of room for research on the design of embedded systems-on-a-chip, especially in the areas of mixed-level simulation, verification, and synthesis.

1 Introduction

The increasing interest in embedded systems has heightened the need for methodologies and tools suitable for modelling, simulation and design of embedded systems. We focus on heterogeneous embedded systems, i.e., those that mix programmable and dedicated components. These systems are of particular interest since they are used as underlying platforms in multimedia and communication-oriented products. In order to deal with the complexity of such systems, designers rely more and more on methodologies and tools that allow them to explore their designs at the system-level. In this document, we report on today's research in the area of system-level methodologies and tools for heterogeneous signal processing systems. We discuss eight tools and their underlying methodologies. Apart from a short description, we also point out some of their strengths and weaknesses. They are: Ptolemy, UC San Diego/NEC, POLIS, VCC, COSY, PAMELA, SystemC, and SPADE. Our choice of methodologies and tools was based on the availability of either the methodologies and tools or information about them.

This paper is organised as follows. First, we give some general remarks about directions in today's methodologies and tools in Section 2. Then we briefly describe each of the methodologies and tools in Sections 3 through 10. Finally, we draw some conclusions in Section 11.

2 General Observations

As we have indicated, modern embedded systems become increasingly complex and not easy to design. They typically have to meet real-time constraints, must be reliable and fault tolerant, and have a low power consumption. Designers have to verify each of these constraints in a model of an embedded system. Models become more accurate when more details are added. However, this also increases the time needed for system model development and simulation. In order to reduce the time needed for modelling and simulation, the evaluation of design choices should move to the early phases of the design process. This can be illustrated in terms of the abstraction pyramid [15] in Figure 1. The cost of model construction and model evaluation is higher at the more detailed levels of abstraction, while the opportunities to explore alternative realizations is significantly reduced at these levels. Therefore, methodologies to deal with the exploration of embedded systems at the system level are of interest.

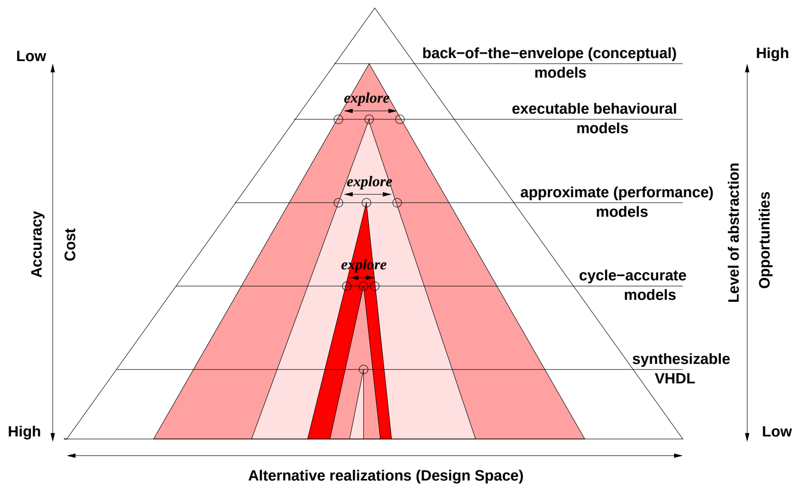
Low
High
Accuracy
Cost
back−of−the−envelope (conceptual) models
explore
executable behavioural models
explore
approximate (performance) models
explore
cycle−accurate models
synthesizable VHDL
High
Low
Level of abstraction
Opportunities
Alternative realizations (Design Space)

Fig. 1. The abstraction pyramid

In the past embedded system designers were almost exclusively operating at the VHDL level. In Figure 2 this is represented with the black dotted arrow. Skipping intermediate levels between high and low levels is only acceptable when a few low level parameters have to be explored. Otherwise, the lower levels are too detailed to explore larger design spaces. Indeed, the complexity of embedded systems grows constantly, and the design approach labelled as 'guru' in Figure 2 is no longer feasible for these complex system-on-a-chip designs. In order to cut time-to-market, embedded system design is now widely believed to benefit from a step-by-step design. This approach is represented by the white dotted arrow in Figure 2. At each level the alternatives are explored before going to the next level.

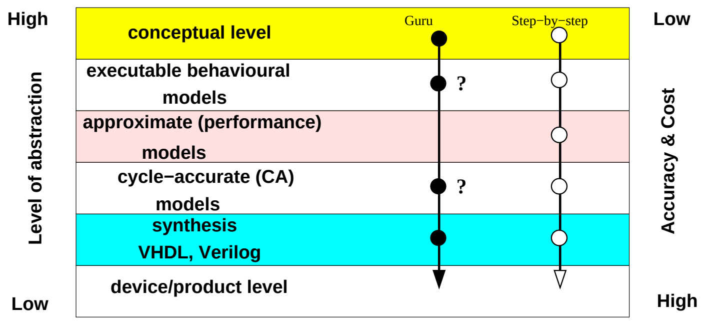

Fig. 2. Guru vs. Step-by-step approach

An interesting abstraction level is the approximate-accuracy level in Figure 1, otherwise known as time-approximate [13], or performance model level [11]. This level bridges the gap between models at behavioural or un-timed [13] level and models at cycle-accurate level.

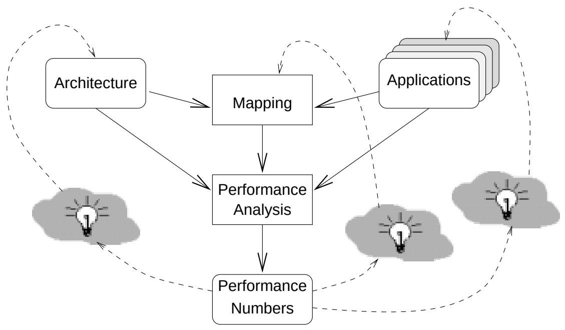

Fig. 3. The Y-chart [16]

Another approach to cut time-to-market is to allow reusability. The *Y-chart* approach [15], [18] is a general scheme that uses reusability for an efficient design-space exploration. It enables reuse by separating architecture and application modelling. This approach is illustrated in Figure 3. The Y-chart approach permits multiple target applications to be mapped one after another onto candidate architectures in order to evaluate performance. The resulting performance numbers may inspire an architecture designer to improve the architecture. The designer may also decide to restructure the application(s) or to modify the mapping of the application(s).

Different methodologies have a different view on how application modelling and architecture modelling in Figure 3 can be performed. Some promote hardware/software co-design based on Models of Computation (MoC) ([4], [12]) and Models of Architecture (MoA) ([18], [2], [12]). Others do not distinguish between MoCs and MoAs but model both the application and the architecture with MoCs. Which MoC or combination of MoCs to chose depends on the nature of the application domain at hand. Many MoC choices are available: Communicating Sequential Processes (CSP), Continuous Time (CT), Discrete Events (DE), Distributed Discrete Events (DDE), Discrete Time (DT), Finite State Machines (FSM), Process Networks (PN), Synchronous Data Flow (SDF), and Synchronous/Reactive (SR) models, or a mixture of these models. While MoCs are well formalised, MoAs have not received that much attention. Architecture features, such as time, types of resources, and resource contention are not easily captured in the formalisms of a single MoC. Figure 4 illustrates how multiple MoCs could be used to model an architecture. In this figure, the application is modelled as a Kahn Process Network (KPN) [17], and the architecture, onto which the application is to be mapped, is modelled in terms of three MoCs: a KPN-like model (with blocking writes in addition to blocking reads), a CSP-like model (with *rendezvous*), and the FSM model. While the application model is homogeneous, the architecture model is not. One can say that both the application and the architecture are specified in terms of MoCs. However, one can also call the combination of MoCs for modelling the architecture a MoA.

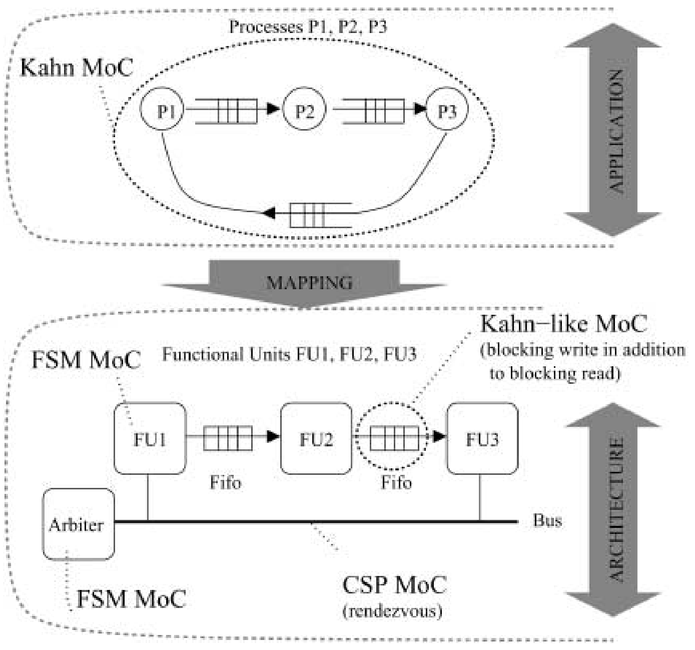

Fig. 4. MoA as an union of different MoCs

In the following sections we briefly present some of the available methodologies and tools in system-level design. A global overview is given for each methodology/tool, and the presence or absence of particular features is pinpointed.

3 Ptolemy

The Ptolemy framework provides methods and tools for the modelling, simulation, and design of complex computational systems [4]. It focuses on heterogeneous system design using MoCs for modelling both hardware and software. Important features are:

1. The ability to construct a single system model using multiple MoCs which are interoperable, and
2. The introduction of disciplined interactions between components, where each of them is governed by a MoC.

The interoperability between different MoCs is based on *domain polymorphism*, which means that components can interact with other components within a wide variety of domains (MoCs). Also, the Ptolemy methodology does not have the objective to describe existing interactions, but rather imposes structure on interactions that are being designed. Components do not need to have rigid interfaces, but they are designed to interact in a possible number of ways. Particularly, instead of verifying that a particular protocol in a single port-to-port interaction can not deadlock, Ptolemy tends to focus on whether an assemblage of components can deadlock. Designers are supposed to think about an overall pattern of interactions, and to trade off expressiveness for uniformity.

Ptolemy does not explicitly support the Y-chart approach, neither does it strictly separate application features and architecture features. There exists only a single implementation of a specified system, which is on top of a Java Virtual Machine. Also, it does not have a layered abstraction approach like, for example, VCC has (see Section 6). However, because of its excellent features, some projects that deal with deriving methodologies for system design use Ptolemy as a kernel for the implementation of a particular methodology into a tool-set. For example, an extension of the Ptolemy kernel in the direction of a Y-chart oriented tool for evaluation of architecture trade-offs, is described in [5].

4 UC San Diego/NEC Methodology

A design space exploration methodology of communication architectures is presented in [6]. The aim of the methodology is to obtain an optimal and automatic mapping of various communication mechanisms among system components onto a target communication architecture template. This is a sensible objective because the volume and diversity of data and control traffic exchanged among System-on-Chip (SoC) components imply that on-chip communication could have severe impediment to system performance and power consumption. What is needed, is SoC communication protocols that efficiently transport large volumes of heterogeneous communication traffic.

The methodology supports an efficient performance analysis of inter-component communication in a bus oriented system. It also supports accurate modelling of communication resources. The method assumes that a communication architecture template is given (see Figure 5), but communication protocols are not.

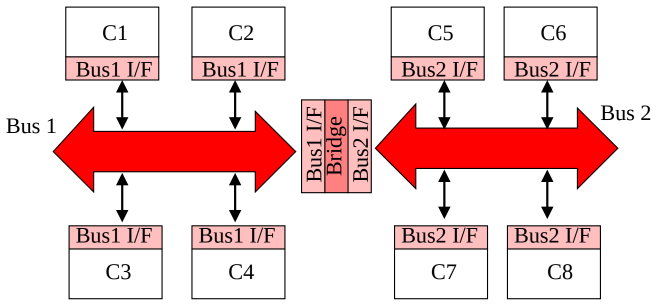

Fig. 5. Predetermined architecture topology in the methodology described in [6]

The methodology consists of two steps:

1. A constructive algorithm, that determines an initial architecture for mapping various SoC communications onto specific paths in a template communication topology.
2. An iterative improvement strategy, that generates optimised mappings, as well as carefully configured communication protocols.

In order to support accurate modelling of dynamic effects, e.g., resource contention, a trace-based approach is employed. First, all computational and communication events that originate from a hardware-software co-simulation of a specified system, are captured in traces. Second, these traces are converted into a communication analysis graph. This graph is the data-structure on which all algorithms used in the methodology are performed.

The methodology provides efficient performance analysis to drive the exploration algorithms of communication architectures. The resulting solutions are characterised by a significant improvement over the initial solutions. The methodology is narrowed to a particular architecture template, but it could be extended to more general architecture template. Also, it could be used as a source of ideas for extending more general methodologies that have problems with communication mapping and optimisation.

5 Polis

Polis is a design environment for *control-dominated* embedded systems [12]. It supports designers in the partitioning of a design and in the selection of a microcontroller and peripherals. It can generate C-code for the software part, including a simple Real-Time Operating System for the microcontroller, and HDL code for synthesis of hardware. It also provides an interface to verification and simulation tools, as well as an embedded (software) simulator.

The underlying MoC used for representation of applications in Polis is the Co-design Finite State Machines (CFSMs). A CFSM is a specialised FSM that incorporates the unbounded delay assumption: In a classic FSM, only the idle phase can have any duration between zero and infinity, the other phases have a duration of zero. The CFSM model can also be described as globally asynchronous/locally synchronous. A system is modelled as a network of interacting CFSMs communicating through events. The events are broadcasted to all connected CFSM.

A system specification is given either using a graphical FSM editor, or using the synchronous language Esterel. The specification is composed of separate modules. Each module is a CFSM. The analysis of a system at the behavioural level can be carried out either with formal tools or by simulation. In the latter case the Ptolemy simulation environment is used.

The Polis environment supports system partitioning by providing useful figures about the partitions. For example, for each module in the software partition, it provides estimations of the execution time and code size. For the hardware modules it gives the number of primary inputs and outputs and the number of latches.

Polis offers several options for simulation:

1. A basic CFSMs simulator that gives all signals and internal states, and that can track the sources of the signals,
2. A simulation of the software partition, where only external signals can be watched,
3. The Ptolemy DE[1] domain simulation of the software partition, and
4. A number of different output formats, such as behavioural VHDL, that can be used for simulation in other simulators.

On the other hand, Polis does not offer the following:

1. Representation of system specification in terms of any other MoC except CFSM,
2. Estimation techniques for more complex processor models, other than simple micro-controllers,
3. Non-dedicated type of communication among components, and
4. Multiple hardware and software partitions (there are only 2 partitions in Polis: one hardware and one software)

Items 1. and 2. imply that the POLIS system can not be used efficiently for designing embedded systems that are not control dominated.

[1] see Section 2

6 VCC – Virtual Component Co-design

The Cadence VCC (Virtual Component Co-design) toolset is built on top of the toolset in the Polis framework. The toolset is intended to support communication refinement. The VCC approach is illustrated in Figure 6. First, the functional behaviour of the entire system is captured and verified, thereby re-using elements from behavioural libraries and algorithmic fragments. Similarly, target architecture is captured by re-using existing architecture IP components (DSP cores, micro-controllers, buses, and RTOSs). The next step involves a manual mapping of behavioural functions and communication channels onto the appropriate architecture resources. The system is evaluated in terms of, e.g., speed, power consumption, and cost. This is a performance analysis step. Then, architecture, behaviour, and mapping for the given system specification are explored until an optimal solution is obtained. Finally, the target architecture is refined into a detailed hardware and/or software implementable architecture. The refined target architecture can be passed to external environments for hardware and software implementations. One important step which is not visible in Figure 6 is co-verification. Co-verification is performed between the architecture and the functional levels and is used for detecting timing problems and bugs.

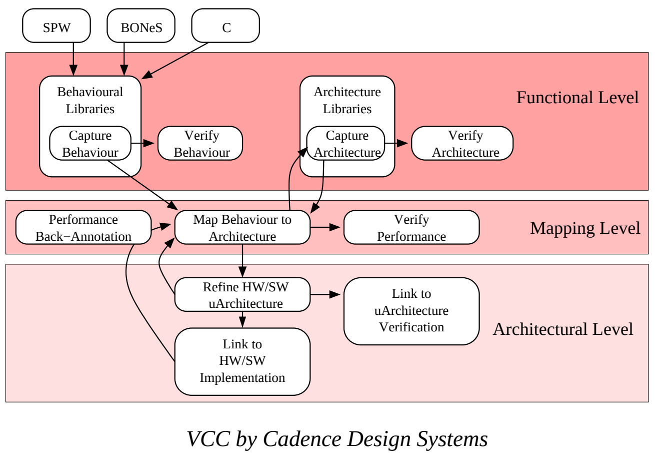

Fig. 6. POLIS/VCC related methodology [7]

VCC can be used together with SPW (Cadence's Signal Processing Worksystem) and BONeS (Cadence's system simulation kernel), as it is shown in Figure 6. SPW and BONeS are useful for specifying behaviour and simulating performance models, respectively. For more information, see [8].

VCC integrates a number of technologies. It provides several input formats for system behaviour, including C/C++. Also, it unifies heterogeneous control and data-flow models. The system behaviour can be given using many different

MoCs, but all of them are actually implemented on top of CFSMs, that plays a dominant role as meta-model in VCC. Moreover, system architecture and system behaviour are well separated. The main VCC features are:

1. Higher abstraction level of architectural estimation models than the current cycle-accurate simulation models,
2. Evaluation of an architecture via mapping of a system behaviour onto an architecture implementation, followed by performance analysis,
3. Interface based communication design.

The third feature has been exploited by the COSY project (see next Section). Also, VCC follows the Y-chart approach. However, a specification of a system behaviour can be simulated only jointly with a system performance model. This means that independent functional simulation, i.e., independent of lower performance related levels of abstraction, is not possible.

7 COSY

In the COSY (COdesign Simulation and SYnthesis) methodology [10], VCC is used in order to obtain an infrastructure for mixing and matching software and hardware components (IPs). Focus is on communication refinement. The input behavioural specification is given using YAPI, Y-chart Application Programmer's Interface [9]. TSS, Philips' internal Tool for System Simulation, is used for the simulation of a refined target architecture. Furthermore, the COSY approach substantially simplifies the design process compared to Polis or VCC. One reason for this is that the levels of abstraction for the communication mechanism that are introduced by the VSI Alliance, i.e., application level, system level, virtual component level, and physical transfer level, are adopted by this approach. Figure 7 shows the definition of the levels of communication in COSY.

The COSY APP[2] level in Figure 7 serves to implement application Process Network (PN) models. At this level an executable untimed specification is used, i.e., a YAPI model [9].

After mapping an application PN onto an architecture, APP communication transfers (or transactions) are refined into system transactions (the COSY SYS level in Figure 7). Functionally, APP, SYS, VCI, and PHY transactions are equivalent. In contrast with APP level the SYS level designers can implement certain functions either as software tasks on a programmable processor core, or as a dedicated coprocessor. However, at the SYS level, transactions still operate on abstract data-types, e.g., video-frames. Hence, the SYS level can be seen as the timed-functional abstraction level used in SystemC (see Section 9). In order to map the high-level I/O semantics and to refine them into the more detailed on-chip communication interfaces, a new level is required. At this VCI level, the interfaces work with addresses and split data in chunks managable by a bus or a switching network. The COSY system integration flow relies on the simple VCI

[2] see Figure 7 for the meaning of acronyms in this section

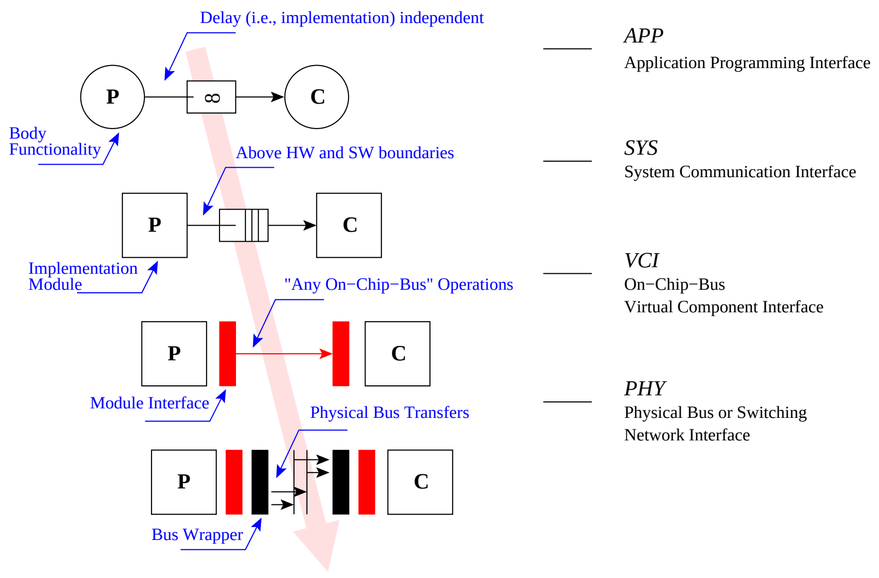
Delay (i.e., implementation) independent
P
∞
C
Body Functionality
Above HW and SW boundaries
P
C
Implementation Module
"Any On−Chip−Bus" Operations
P
C
Module Interface
Physical Bus Transfers
P
C
Bus Wrapper
APP
Application Programming Interface
SYS
System Communication Interface
VCI
On−Chip−Bus Virtual Component Interface
PHY
Physical Bus or Switching Network Interface

Fig. 7. COSY related transaction levels [10]

wrappers that translate the protocol used by VCI compliant interfaces into the physical protocol of the selected bus. The final level deals with physical bus size, signalling, and arbitration protocols. This level is marked as PHY in Figure 7. A simulation model at this level can be used to calibrate the estimation models at higher levels of abstraction.

The benefit of COSY is that a designer can, starting from pure behavioural/functional specification, do communication refinement and design-space exploration using generic performance models, and efficiently get to an optimal implementation. Furthermore, he can effectively exchange IPs because of clear separation of functionality and architecture, as well as the separation of IP behaviour and communication.

8 PAMELA

System performance modelling can be also done using a modelling language. In this section we discuss the tool PAMELA (Performance Modelling Language) [11]. The objectives of PAMELA are:

1. Improving model accuracy by allowing dynamic behaviour, and
2. Providing a source language for a static performance modelling that yields an analytic (compile-time) model.

In order to cope with these conflicting goals, PAMELA imposes some restrictions in particular with respect to the modelling of synchronisation, so that the objectives can at least partly be achieved. It uses highly structured language operations to describe four factors that determine parallel system performance modelling: conditional synchronisation (CS), mutual exclusion (ME), conditional control flow (CCF), and basic calibration of performance models (BC).

There are few features that should be observed:

1. PAMELA gives priority to static compile-time analysis, which goes at the cost of possible reduced accuracy. So, PAMELA could be seen as an analysis method and not as a performance modelling method. However, the PAMELA model does not fully exclude dynamic behaviour and does give more accurate performance measures than fully static models. See [11].
2. PAMELA does not distinguish separate formalisms to model programs and machines - there is no distinction between MoC and MoA. The Y-chart approach is thus not supported explicitly.
3. PAMELA does not support re-usability and extension of models clearly. In contrast to SPADE or Ptolemy, PAMELA was not designed to do so.

9 SystemC

Originally intended for software design, neither C nor C++ are suited to model hardware [14]. There are two ways to remedy this lack: either to build syntax extensions, or to introduce specific class libraries. The second approach has been taken by the developers of SystemC. Currently, SystemC has wide support both from commercial and academic side, and tends to become a standard as a language based modelling tool for System-level design.

With SystemC, embedded systems can be described by means of multiple concurrent processes. The underlying simulation kernel of SystemC is built on top of a co-routine based multithreading library.

By providing a C++ class library almost all kind of communication mechanisms can be modelled. The remote procedure call mechanism is used to model master-slave communication at the high abstraction level. Moreover, version 2.0, which is supposed to be available at the second half of 2001, is going to support user defined types of communication.

SystemC uses modules, which usually encapsulate some component, either a processing element or a communication block, and which can contain other modules or processes. Processes serve to capture functionality, and can be reactive either to any input signal or to a clock. Also, processes can be either synchronous, meaning they include timing control statements or conditional synchronisation, e.g., *wait*(*delay*), and instructions between *wait*s are timeless, or asynchronous, meaning instructions are timeless, and local variables are redefined each time the process is invoked [13].

Another benefit of the library based implementation of hardware objects is that SystemC is ANSI C++ compliant; hence, an application and an architecture (whole system) can be modelled within one and the same simulation environment.

System C distinguishes several levels of abstraction:

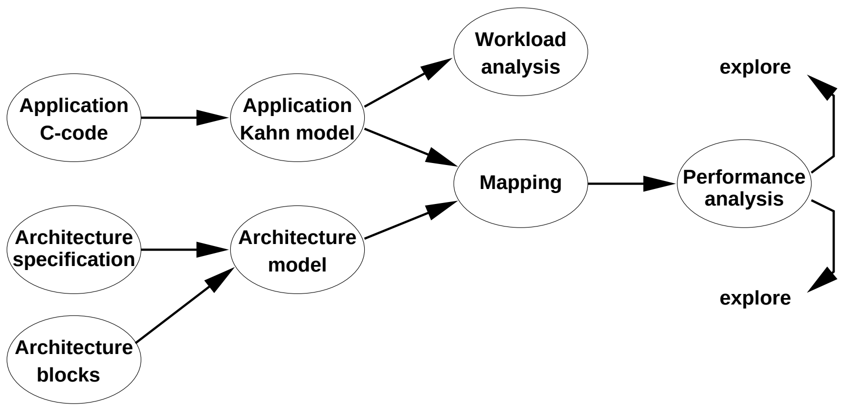

Fig. 8. The SPADE methodology flow

1. *Untimed functional*: control/dataflow MoCs
2. *Timed functional*: behavioural processes,
3. *BCA/CA functional*: bus-cycle-accurate/cycle-accurate architecture modules.

Modelling is possible at all levels of abstraction. Furthermore, SystemC provides interoperability among different levels of abstractions. That helps in dealing with large scale system complexity - simply we are allowed to model certain parts at the CA level, while others are at higher functional levels.

Specifications given within the SystemC context are executable. As a result, model verification is possible. It supports modelling of both hardware and software, and has support for low level hardware models. Because of a broad support, SystemC tends to become a standard, which would empower re-use and sharing of models.

SystemC itself should not be considered as a methodology; it is a modelling language. But as such, it may have impact on the exchange and interoperability of IP's at the various levels of abstraction.

10 Spade

In this section we present a concise description of the Spade methodology [1], [16]. Spade stands for System-level Performance Analysis and Design-space Exploration. It is both a methodology and an implementation for high level system modelling and evaluation. The Spade methodology follows the Y-chart approach introduced in Section 2. The methodology flow is illustrated in Figure 8. In this flow we recognise the aspects of Y-chart application modelling, architecture modelling, mapping and performance analysis. We now briefly comment on the various parts in Figure 8.

For application modelling Spade uses Kahn Process Network MoC [17] in order to make task level parallelism and communication explicit. The Kahn processes can run in parallel, but internally each of them is sequential. They communicate by token-exchange via unbounded channels. The application model represents the workload that is imposed on an architecture. The workload consists of two parts: communication workload and computation workload. Communication workload is seen through actions on channels, while computation workload is annotated explicitly. If an application model is run independently of an architecture model, a designer can do workload analysis.

Spade uses a building block approach for architecture modelling. Each architecture component is instantiated according to an architecture specification. Also, architecture components are generic building blocks. They describe only a timing behaviour, and not a functional behaviour. A set of parameters that is given in the architecture specification, is directly related to the timing behaviour of a particular component.

Spade supports an explicit mapping step, where each application process is mapped on a particular architecture component. Spade uses trace-driven execution to map the workload on an architecture model. In other words, application processes generate traces which drive architecture components. Traces contains stamps of communication and computation activities for each process.

When the application model, architecture model, and mapping are available, then Spade can perform simulation. After the execution, Spade provides some performance numbers: number of cycles processor was executing, was switching context, was busy with I/O or stalling or was idle, bus utilisation, and many others. These numbers should give guidelines to a designer of what are the bottlenecks or which parts of the architecture are under-utilised. The performance numbers can later on be visualised, analysed, and used for generating metrics information.

Since Spade has to obtain some hardware performance numbers, a hardware simulator tool has been included. Currently, this simulator is TSS (see Section 7), which is a BCA-level simulator. BCA-level simulators are typically slower than DE simulators at the higher levels of abstraction. While this is a drawback, an advantage is that in this environment it may be possible to easily perform mixed level simulation and exploration.

11 Conclusion

We have reported on a study that we conducted to evaluate and compare different features of some embedded system design methodologies and tools that are available today. Although they are all intended to be used in the field of system-level design, they are very diverse.

We studied eight different methodologies and tools: Ptolemy, UC San Diego/-NEC methodology, POLIS, Cadence VCC, COSY, PAMELA, SystemC, and Spade. In Table 1 we summarise their most interesting properties.

We conclude that the presented methodologies and tools differ from each other in their approach to hardware/software co-synthesis, e.g., Y-chart support

Table 1. Simultaneous overview of properties of presented methodologies & tools.

Methodologies & Tools ↦	*Ptolemy*	*UCSD/NEC*	*POLIS*	*VCC*	*COSY*	*PAMELA*	*SystemC*	*SPADE*
Commercial	-	-	-	+	+/-	-	-	-
Available	+	-	+	+	-	+	+	+/-
Y-chart supported	-	+	+	+	+	○	-	+
MoC variety supported	+	-	-	○/-	○	-	○	-
MoA support	-	○	○	+	+	-	○	+
Dynamic aspects modelling	+	+	+	+	+	+	+	+
Formal analysis & verification	-	○	+	○	○	+	-	-
Reusability	+	+	+	+	+	-	+	+
Complex designs	○	○	-	+	+	-	+	+
Multiple abstraction levels	-	○	+	+	+	-	+	○
Mixed level simulation	-	-	○	-	○	-	+	○
Support for Synthesis	-	○	+	○	○	-	-	-

in the third row in Table 1, as well as in their use of MoCs, e.g., in the fourth row in Table 1, and their use of MoAs, in the fifth row in Table 1.

Although most of the methodologies and tools share some common features, the overview shown in Table 1 indicates that from our selection of methodologies and tools COSY is the most complete one. The features shown in Table 1 also do suggest that (1) today's embedded system design methodologies and tools do not yet solve all design problems, and thus, (2) there is plenty of room for further research, e.g., support for synthesis, multiple abstraction levels, and formal analysis and verification. Specifically, there is more work to be done to master complexity in designs and to support mixed abstraction levels (rows 9 and 11 in Table 1).

Acknowledgements

This study was performed in part in the Archer project, funded by Philips Semiconductors. We want to thank Ed Deprettere (Leiden University), and Pieter van der Wolf (Philips Research Laboratories) for their contributions to this paper.

References

1. P. Lieverse et al.: A methodology for architecture exploration of heterogeneous signal processing systems. In Proc. 1999 Workshop on Signal Processing Systems, Taipei, Taiwan, Oct. 1999.
2. W. Wolf: Hardware/Software Co-Synthesis Algorithms. In System-Level Synthesis, A.A. Jerraya and J. Mermet (eds.), pp. 189-217, Kluwer Academic Publishers, 1999.
3. J.S. Davis II: Order and Containment in Concurrent System Design. In PhD thesis, University of California at Berkeley, Fall, 2000.
4. E.A. Lee et al.: Overview of The Ptolemy Project. Technical memorandum UCB/ERL M01/11, University of California at Berkeley, March, 2001.
5. E. Pauer, J.B. Prime: An Architectural Trade Capability Using The Ptolemy Kernel. In Proc. of the 1996 IEEE Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP), 1996.
6. K. Lahiri, A. Raghunathan, S. Dey: Performance analysis of systems with multichannel communication architectures. In Proc. Int. Conf. VLSI Design, pp. 530-537, January 2000.

7. G. Martin, B. Salefski: Methodology and Technology for Design of Communications and Multimedia Products via System-Level IP Integration. In Proc. of DATE 98, February 1998.
8. http://www.cadence.com/articles/vcc_meth_backgrounder.html
9. De Kock et al.: YAPI: Application modeling for signal processing systems. In Proceedings of DAC'2000, Los Angeles, USA, 2000, June.
10. Jean-Yves Brunel et al.: COSY Communication IP's. In Proceedings of DAC'2000, Los Angeles, USA, 2000, June.
11. Arjan van Gemund: Performance Modeling of Parallel Systems. In PhD thesis, Technische Universiteit Delft, the Netherlands, 1996.
12. F. Balarin, et al.,: Hardware/Software Co-Design Of Embedded Systems - The POLIS Approach. Kluwer Academic Publishers, 2nd printing, 1999.
13. Synopsys, Inc., CoWare, Inc., Frontier Design, Inc. and others: Functional Specification For SystemC 2.0 - Final. January 17th, 2001.
14. D. Verkest, J. Kunkel, F. Schirrmeister: System Level Design Using C++. white paper, IMEC - Synopsys - Cadence.
15. A.C.J. Kienhuis: Design Space Exploration of Stream-based Dataflow Architectures *Methods and Tools*. In PhD thesis, Technische Universiteit Delft, the Netherlands, 1999.
16. Paul Lieverse, Todor Stefanov, Pieter van der Wolf, Ed Deprettere,: System Level Design with Spade: an M-JPEG Case Study. In Proc. of Int. Conference on Computer Aided Design (ICCAD'01), San Jose CA, USA, Nov. 4-8, 2001.
17. G. Kahn: The semantics of a simple language for parallel programming. Information processing 74 - North-Holland Publishing Company, 1974.
18. B. Kienhuis, E. Deprettere, K. Vissers and P. van der Wolf: An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. In Proc. 11-th Int. Conf. on Application-specific Systems, Architectures and Processors, Zurich, Switzerland, July 14-16 1997.