Arche**r**

PROJECT :	**Architecture Exploration (SPADE 2)**
REPORT :	1
SUBJECT :	*State of the Art in the Fields of System Specification and Architecture Exploration*
DATE :	June 19th, 2001
ABSTRACT :	*In this report we present an overview of system level design methodologies and tools. 8 different tools and their underlying methodologies were analysed. We found that existing tools are not mature enough to completely fulfill all design requests. In our opinion, more research has to be done in the area of mixed-level simulation, verification, and synthesis of complex heterogeneous systems.*
AUTHOR :	**Vladimir D. Živković**, LIACS - Leiden University
KEYWORDS :	Embedded system design; System level design methodologies and tools; Abstraction levels; Hardware/Software co-synthesis; Y-chart;

1 Introduction

The increasing interest in *E*mbedded *S*ystems (ES) has heightened the need for tools and methodologies suitable for ES modelling and design. Of particular interest are ES's that are defined as *heterogeneous* systems (i.e., those that mix different technologies and/or different operations). The heterogeneous signal processing systems are used as underlying platforms in multimedia and communication-oriented products, and due to the complexity of them (concurrency and often real-time requests), development of tools and methodologies for their design is not easy. In order to resolve this complexity, researchers usually introduce tools and methodologies that start with modelling of a ES at the system-level. In this document, we report on available research in the area of system-level *C*omputer *A*ided *D*esign (*CAD*) tools and methodologies for heterogeneous signal processing systems.

The remainder of the document is organized in following way: first, we give some general remarks about directions among today's methodologies and tools (Section 2), and after that we briefly describe UC Berkeley Ptolemy methodology (Sub-section 2.1) and related work (2.1.1), UC San Diego/NEC methodology (Sub-section 2.2), the POLIS system (Sub-section 2.3), Cadence VCC tool (Sub-section 2.4) and the derivative COSY flow (2.4.1), PAMELA modelling language (Sub-section 2.5), and SystemC - C/C++ language based modelling technique (Sub-section 2.6). The last section deals with SPADE and is subdivided into two subsections: the SPADE methodology (Sub-section 3.1) and the SPADE software structure (Sub-section 3.2). Section 4 is the conclusion of this document.

2 Methodologies and Tools

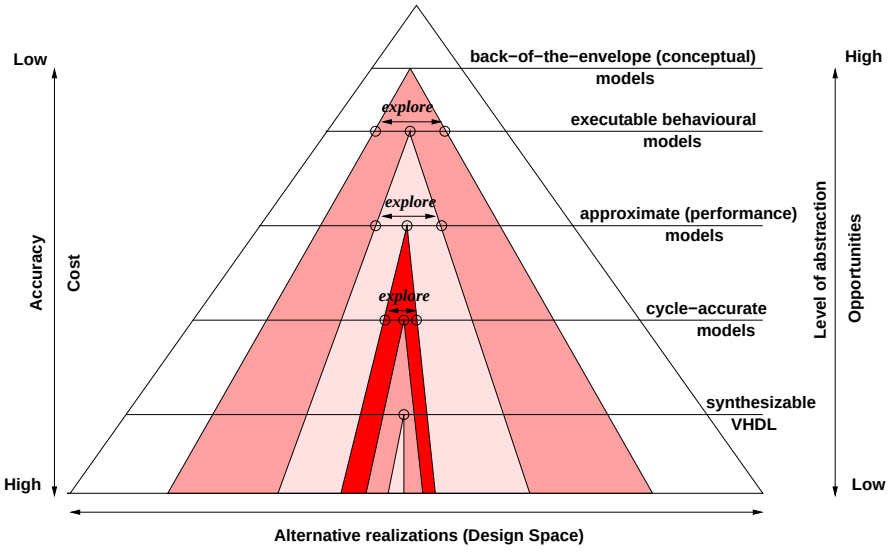


Figure 1: The abstraction pyramid

In this section we elaborate on a few directions of ES design that have emerged in the *E*lectronic *D*esign *A*utomation (EDA) community. In general, methodologies that deal with *h*ard*w*are/soft*w*are (HW/SW) co-synthesis of ES are developed to simplify the ES design as much as possible. They usually introduce abstraction levels, in order to hide the unnecessary complexity of ES design. One interesting way of illustrating the relations between accuracy, cost, and opportunities, when exploring

a design-space at various levels of abstraction, is shown in Figure 1.

An interesting level of abstraction is the *approximate-accuracy* level in Figure 1 - sometimes called *time-approximate* [15], or even *performance model* level [13]. The purpose of this level is to give more opportunities to the designer to explore alternative solutions, as well as, to bridge the gap between *behavioural* (*un-timed* [15]) level models and *cycle-accurate* level models. In Figure 2[1], the black dotted arrow represents the so-called "guru approach," where the ES designer jumps from the *conceptual* or *behavioural* model to the *cycle-accurate* or even *HDL* models. In order to cut time-to-market, ES design is now widely believed to benefit from a step-by-step design space exploration. This approach is represented by the white dotted arrow in Figure 2.

2

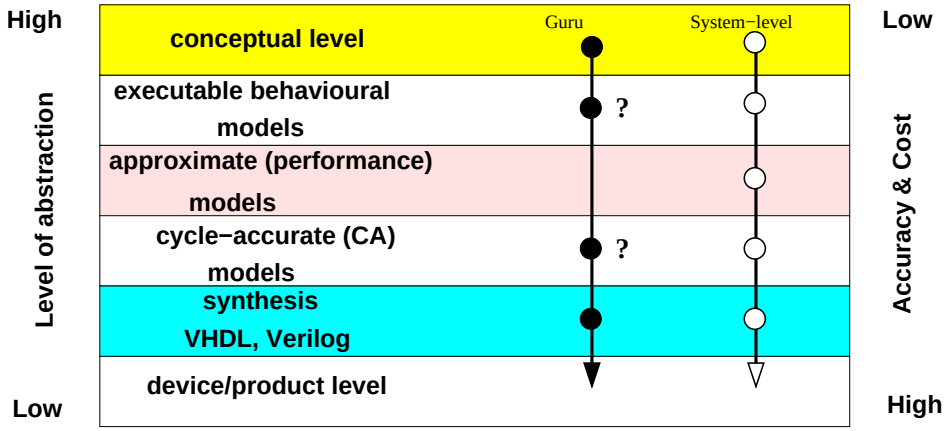


Figure 2: "Guru" vs. System-level approach

Different methodologies have a different view on the abstraction pyramid of Figure 1. Some are platform based [7], others are not. Some promote HW/SW co-design based on *M*odels *o*f *C*omputation (MoC) and *M*odels *o*f *A*rchitecture (MoA) (see [27],[2],[14]). Others do not distinguish between MoC and MoA but model both the application and the architecture with MoCs. Which MoC is chosen depends on the type of model being constructed[2]. Choosing a MoC can be daunting especially at the field of system level design. Since we are addressing ES design at different levels of abstraction, many MoC choices are available: *c*ommunicating *s*equential *p*rocesses - *CSP*, *c*ontinuous *t*ime - *CT*, *d*iscrete-*e*vents - *DE*, *d*istributed *d*iscrete-*e*vents - *DDE*, *d*iscrete-*t*ime - *DT*, *f*inite-state *m*achines - FSM, *p*rocess *n*etworks - *PN*, *s*ynchronious *d*ata*f*low - *SDF*, and *s*ynchronious/*r*eactive - *SR* models, or a mixture of these models. See the UC Berkeley Ptolemy II program [4].

While MoCs are very well formalised, MoAs have not received that much attention. However, many researchers favour the MoAs viewpoint ([1],[2],[6],[7],[9],[10],[14]). Whether there is a clear distinction between MoCs and MoAs may be a matter of taste. For example, in Figure3 are shown an application modelled as a *K*ahn *P*rocess *N*etwork (KPN)[21], and an architecture, onto which the application is to be mapped, modelled in terms of four MoCs: a KPN-like model (with blocking writes), a CSP-like model (with *rendezvous*), a *S*tream *B*ased *F*unction (SBF) MoC (which in itself uses the FSM model), and the FSM model. While the application is homogeneous, the architecture is not. Because MoA = MoC_1, MoC_2, MoC_3, MoC_4 one can say that both the application and the

[1]The original drawing was introduced by E. de Kock (Philips Natlab). See the C/C++ taskforce meetings minutes.

[2]The choice of MoC is problem domain specific.

architecture are specified in terms of MoCs. However, one can also classify architectures in terms of the number of MoCs it can be expressed in and call that classification models of architecture.

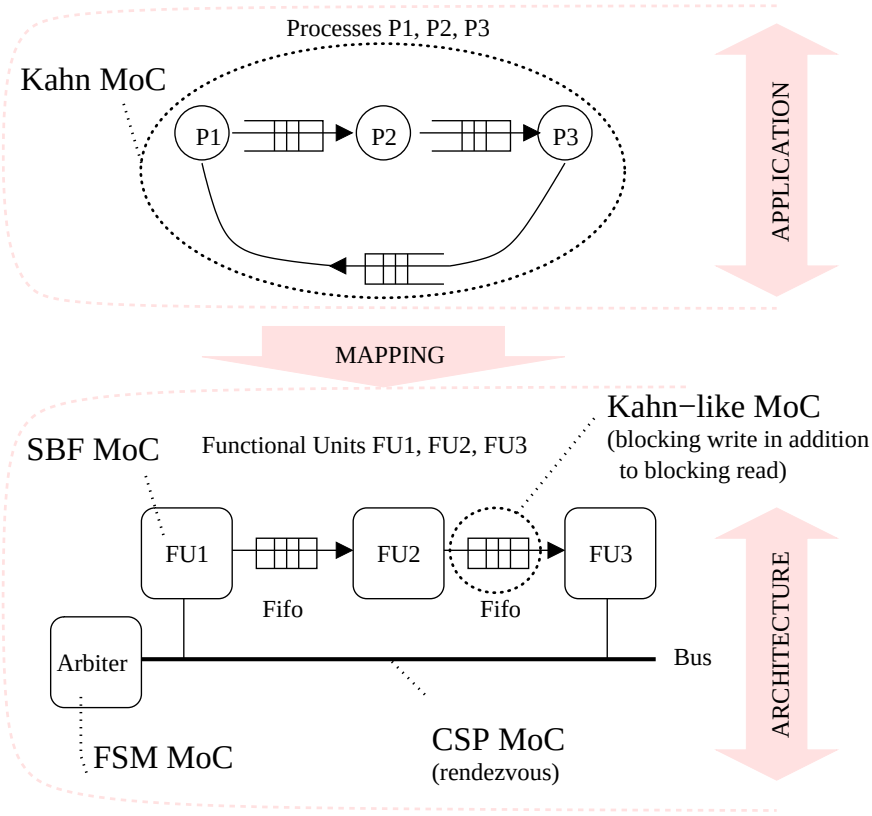


Figure 3: MoA as an union of different MoCs

Figure 3 opens a space for introducing the *Y-chart* approach [19]. The *Y-chart* approach permits multiple target applications to be mapped one after another onto candidate architectures in order to evaluate their performance. The resulting performance numbers may inspire an architecture designer to improve the architecture. He may also decide to restructure the application(s) or to modify the mapping of the application(s). The *Y-chart* approach is described in more detail later in Section 3, where we introduce the SPADE methodology.

In the following sub-sections we briefly present a subset of the available methodologies and tools in system-level design. Each methodology/tool is presented in a general form, and the presence or absence of particular features is pinpointed.

2.1 *Ptolemy*

The *Ptolemy* program studies the modelling, simulation, and design of complex computational systems [4]. It focuses on the design and use of MoCs in order to model both HW and SW. Hence, two very important features are supported in the Ptolemy methodology:

1. The construction and interoperability of a wide variety of executable models, and

2. Component-based design that involves disciplined interactions between components governed by a MoC.

An interesting claim is that Ptolemy (in fact *Ptolemy II*) might be called a software *a*rchitecture *d*escription *l*anguage (ADL) [4]. The rationale for this is that it does not have only the objective to describe existing interactions, but rather to impose some structure on those interactions. Thus, a new approach has been made: instead of verifying that a particular protocol in a single port-to-port interaction does not deadlock, Ptolemy tends to focus on whether an assemblage of components can deadlock.

Other important features of the *Ptolemy* methodology are:

- *Ptolemy II* components are *domain polymorphic*,
- *Ptolemy II* has the ability to mix different domains hierarchically, and
- Translations across MoCs are not needed.

The first two bullets are particularly interesting since they are dealing with *O*bject-*O*riented *D*esign (OOD) flavours, and it is good to see how these flavours fit into system-level EDA. The term *domain polymorphic components* means that components can interact with other components within a wide variety of domains. This feature is directly related to the interoperability between different MoCs. In this way, components do not have to have rigid interfaces, but they are designed to interact in a possible number of ways. Looking into the issue of mixing different domains hierarchically, we come to the problem of organising abstractions in OOD. Ptolemy tends to bring discipline among component interactions by forcing designers to think about an overall pattern of interactions, and to trade off uniformity for expressiveness. In short, this is the way in which Ptolemy exploits the power of OOD to master the complexity of concurrent ES design.

Ptolemy does not facilitate explicitly the *Y-chart approach*, and it does not strictly separate behaviour from architecture (there exists only a single implementation, which is on top of the Java Virtual Machine). Also, it does not contain a layered abstraction approach (like for example *VCC* that is described in one of the following sections). However, due to its good features (component-based design, wide variety of MoCs, interoperability among different component from different domains/MoCs), many projects that deal with deriving methodologies for system design use Ptolemy as a *kernel* for the implementation of the particular methodology into a tool-set. In the next subsection one particular way of architecture exploration that is based on the Ptolemy kernel is illustrated.

2.1.1 Architectural Trade Tool Based On Ptolemy Kernel

This tool is a *Y-chart* based tool, and it is a product of *Lockheed Martin Company*. It is described in [5], and its main idea is an "extension" of the Ptolemy kernel with:

- A capability to define an architecture,
- A functional mapping,
- A synthesis of an executable performance model, and
- A feedback on the results.

First, a functional representation of the algorithm (or "system behavioural description", in terms of other methodologies and tools) is produced using Ptolemy's SDF MoC. The reason for choosing SDF is that it fits the data-flow application domain well because of determinism and an explicit introduction of boundness.

Since the schedule does not depend on the data (this is a result of using SDF), we should verify an algorithm first, and then, after a successful verification of the algorithm, performance simulation can be done separately. The Ptolemy's DE MoC is used as an engine for performance simulation. Particularly, each architecture component (processor, bus, data source, or data sink) is modelled as a parameterised DE actor. In that way, designers have a basis for models of specific hardware devices.

After defining architecture entities, a designer can specify a functional mapping onto the architecture. All SDF actors have to be assigned to the processor entities in the architecture.

The following step is a performance level simulation of the architecture with the specified functional mapping. This is the main part of the architecture trade tool. The performance model that is synthesised here contains the already mentioned architecture entities and one or more functional actors. These functional actors are exactly in 1-to-1 relation with the SDF representation of the algorithm. They model the computational load (execution time) of the SDF actor, its memory usage, and data block sizes for inputs and outputs. Cost functions, expressed using arithmetic operations and transcendental functions are obtained by profiling the code. The actual accuracy depends strongly on these functions. It is important to mention that these functions are not easy to obtain and that they are unique for each different type of processor.

Finally, after a performance model is created, it can be executed. During the simulation various profile information is collected. Later, this information is used to identify bottlenecks and underutilised resources. Also, the tool gives some feedback on various system metrics, such as, power, cost, schedule, reliability, fault tolerance, etc. The steps described above can be characterised as a typical ES design flow (see Figure 4). The tool this section is dealing with, encapsulates this flow as its basic methodology.

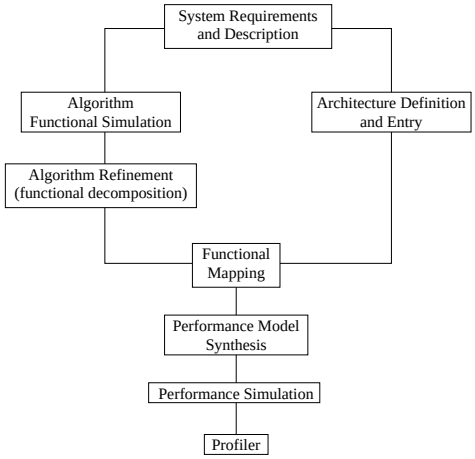


Figure 4: Step-based description of the typical system design flow (according to [5])

The goal of this tool is efficient architecture design space exploration, since the user (designer) can quickly evaluate various combinations of mappings and architectures in order to decide which combination fits best the given set of system requirements. The result is presented by means of *Gantt*

charts and thermometer scales; hence, an user can quickly notice whether some component is under-utilised or not, and whether some metrics are satisfied or not (respectively).

There are few aspects that are considered to be improved. They are primarily directed to (semi) automated determination of cost functions and functional mappings. Additionally, the performance component models (e.g., bus, processor) are limited in a sense that they support only FCFS (*F*irst *C*ome *F*irst *S*erve) arbitration/scheduling policy. Another issue is that more sophisticated system metrics calculus might be developed. Finally, MoC and MoA are still expressed in the same way, i.e., via Ptolemy MoCs (an application - that is a functional behaviour - via SDF MoC, and an architecture - that is "timed" model - via DE MoC). Mapping one onto the other creates a performance model, and this "architecture trade tool" can be seen as Y-chart oriented.

2.2 UC San Diego/NEC Methodology

An interesting design space exploration methodology has been presented in [6]. The aim is to obtain an optimal and automatical mapping of various communication mechanisms between system components onto a target communication architecture template. In addition, the configuration of communication protocols of all channels should be derived simultaneously (in order to optimise system performance). Most of the methodologies that deal with SoC (*S*ystem *o*n *C*hip) design space exploration tend to find an optimal mapping of application computation requirements (workload) onto a set of high-performance system components. Contrary, this methodology focuses on creating techniques and tools that have to map system's communication requirements onto a well optimised communication architecture (suited to an application) [6]. The reason for this focus on communication aspects is that the volume and diversity of data and control traffic exchanged among SoC components cause that on-chip communication (designed by "conventional" methodologies) could have severe impediment to optimal system performance and low power consumption. What a designer needs, however, is a SoC communication that efficiently transports large volumes of heterogeneous communication traffic. Unfortunately, most of the EDA-community work has been focused on an optimal mapping of system's functionality onto system components, and very little has been done in the area of mapping system's data and control communication onto a communication architecture. Furthermore, researchers spend more time on an architecture topology (which is also critical in a view of ES's), and less on the complementary problem of finding optima in communication protocols. This was the motivation for deriving a methodology that supports efficient performance analysis by means of:

- inter-component communication in a given system, and
- accurate modelling of potential contention for communication resources (channels).

2.2.1 Methodology Flow Description

In order to suffice designer's needs a "communication architecture design space exploration methodology" was presented in [6]. It assumes that an architecture topology is predetermined (i.e., there already exists a communication architecture template - see Figure 5), but communication protocols are not. The methodology consists of:

a) A constructive algorithm, that determines an initial architecture for mapping various SoC communications onto specific paths in a template communication topology.

b) An iterative improvement strategy, that generates optimised mappings, as well as carefully configured communication protocols.

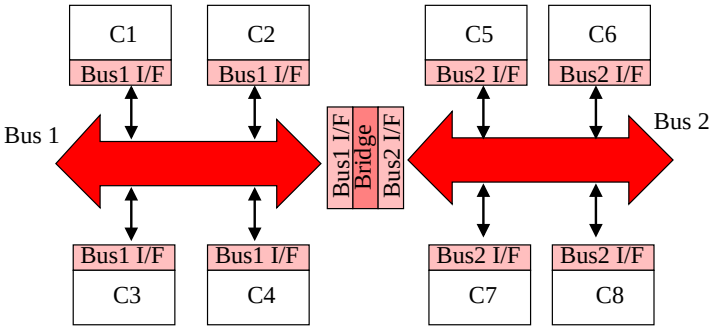


Figure 5: Predetrminated architecture topology in the "NEC" methodology [6])

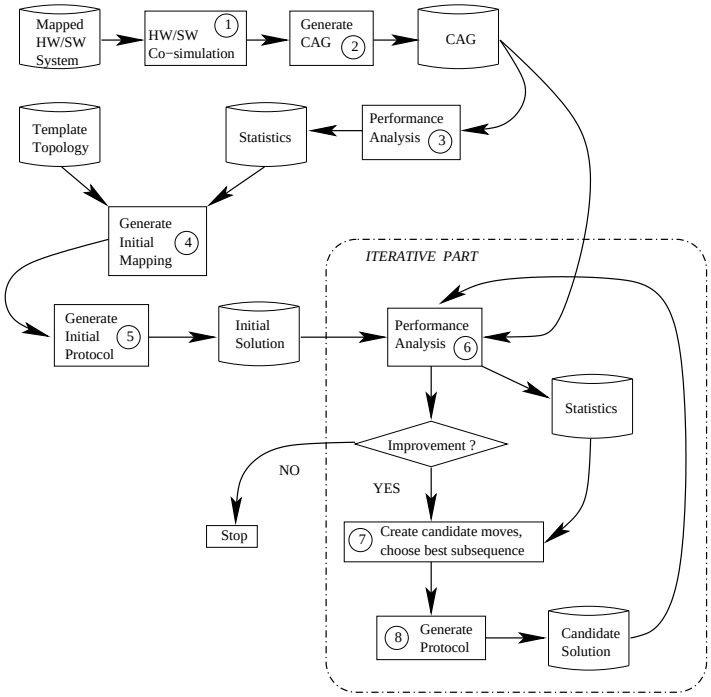


Figure 6: Overall communication architecture design methodology flow (according to [6])

As input, the methodology requests a system description that is already partitioned into HW/SW and mapped onto appropriate components. This corresponds to the first step in Figure 6. So HW/SW partitioning (using *POLIS*) and functional co-simulation (using *Ptolemy*) are performed in this step.

After a communication structure between components has been modelled, execution traces are collected and stored in a compact representation called *C*ommunication *A*nalysis *G*raph (CAG). The CAG captures the abstracted system behaviour in terms of computation, communication, and inter-component synchronisation, over the entire simulation trace (the second step in Figure 6).

Using different algorithms various statistics are generated to capture the system performance and the inter-component traffic. This is the third step in Figure 6.

The fourth and fifth steps in Figure 6 correspond to the initial mapping and communication protocol decisions. During the fourth step, a "constructive heuristic procedure" generates an initial distribution of components over the target communication structure, starting from the statistics generated in the third step and the input template communication architecture topology. The fifth step determines a set of protocol parameters for each communication channel.

Since the given initial solution is usually sub-optimal, a designer has to use the iterative part of the methodology. Thus, in the sixth step, the performance analysis tool is re-invocated. The focus is now on the proposed initial solution, i.e., on re-evaluation of its performance and communication statistics. Then, based on these statistics, alternative solutions are explored by calculating the potential performance gain from moving and re-mapping some components and channels (the seventh step). After choosing the best candidate for moving and re-mapping, the new mapping is generated. In the eighth step, it is important to choose optimal protocol parameters for mapping resulted from previous two steps. When a satisfactory improvement is achieved, the iteration over the step sequence $6 \rightarrow 7 \rightarrow 8$ can be terminated (for more details see [4]). The resulting candidate solution is optimal with respect to the criteria given as *Improvement?* in Figure 6.

2.2.2 Methodology Properties

The methodology described above deals with creating an optimal mapping onto a target communication topology. Moreover, it offers *automated* design space exploration for application specific SoCs. It provides efficient performance analysis to drive the exploration algorithms, which result in solutions that are characterised by a significant improvement over the initial solutions. Its trace-based approach supports more accurate modelling of dynamic effects (resource contention) than most static performance modelling techniques. Despite this methodology is narrowed to a particular architecture template, it could be used as a source of ideas for extending more general methodologies that have problems with communication mapping and optimisation.

2.3 The *POLIS* System

*C*o-design *F*inite *S*tate *M*achines (CFSM) is the underlying MoC in the *POLIS* system. Using the CFSM MoC, and its globally asynchronous/locally synchronous formalism, the POLIS system captures specifications of control-dominated systems. The input to POLIS is a system specification that uses a graphical language to describe links between the FSMs and the formal synchronous language combination (called ESTEREL) to describe FSMs behaviour. The analysis at the behavioural level can be carried out either with formal tools or by simulation in which case the Ptolemy simulation environment is used. The currently implemented design flow in the POLIS system is shown in Figure 7.

The complexity of the diagram in Figure 7 is caused by a variety of ES design requests such as safety, reliability, robustness, low cost in power dissipation and silicon area, etc. However, using 7 steps, described in [14], the POLIS system design flow can be "digested" easier. First, the ES specifications are translated (*Translators* in Figure 7) into CFSMs. Second, *Formal Verification* is supported. Regardless that today's formal verification tools still have problems with complexity of ES, POLIS deals with complexity via its formal verification methodology quite successful. The system-level HW/SW *Co-Simulation* ensures that designers get feedback on their design choices, even in early stages of the design. This is the step (step 3 in [14]) that involves Ptolemy kernel as a simulation engine, although, as stated in [14], other kernels can be used. The *Partitioning, Architecture Selection, and Scheduler Selection* step captures design decisions such as HW/SW partitioning, architecture se-

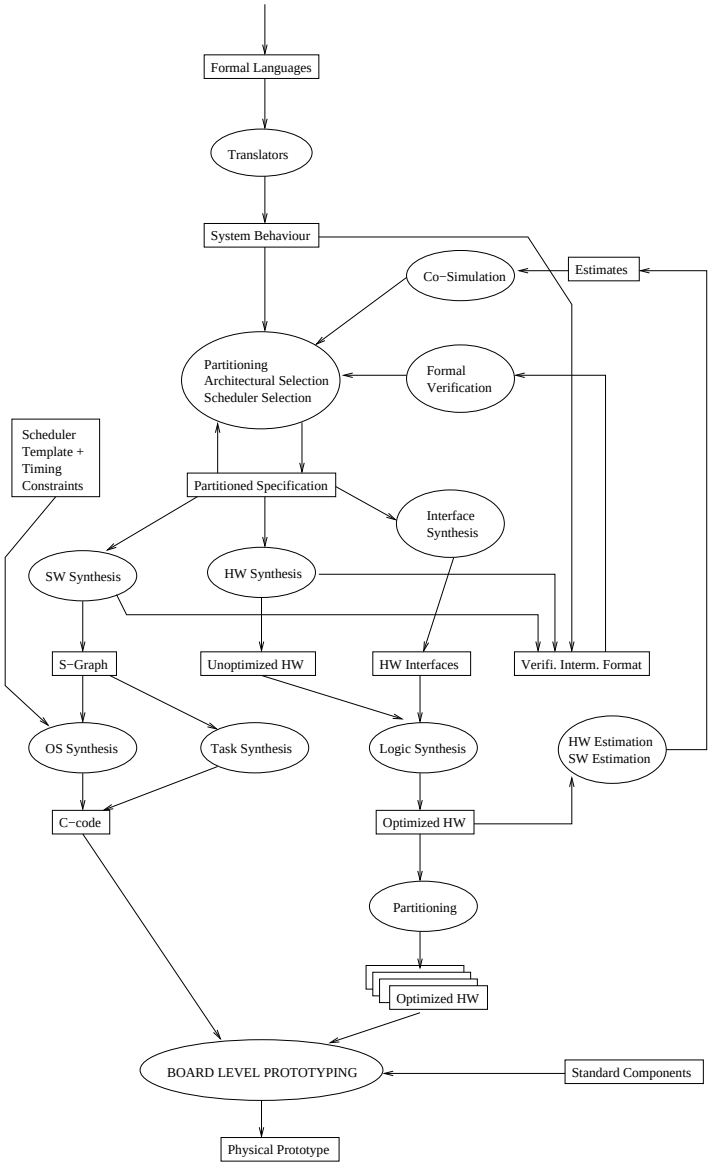


Figure 7: POLIS related design flow [14]

lection, and scheduler selection. These decisions are highly dependent on the designer's knowledge, so POLIS tries to give a design hint from the previous two steps and to evaluate the resulting architecture. The next steps are to map the chosen sub-networks of CSFMs into an abstract hardware description format (*HW Synthesis*) and software structure that includes a simple *R*eal-*T*ime *O*perating *S*ystem - RTOS (*SW Synthesis*). POLIS implements each transition function as a combinational network, optimised using *Logic Synthesis*, and latches some outputs from the network to implement state variables and thus ensures correctness of CFSM composition. Software CFSM's reactive behaviour is synthesised first as processor (platform) independent control/data flow graph (*S-Graph*), next it is implemented by means of portable and restricted *C-code*, and then implemented into a micro-controller specific instruction set. The RTOS is generated during the *OS Synthesis* part, and is an application

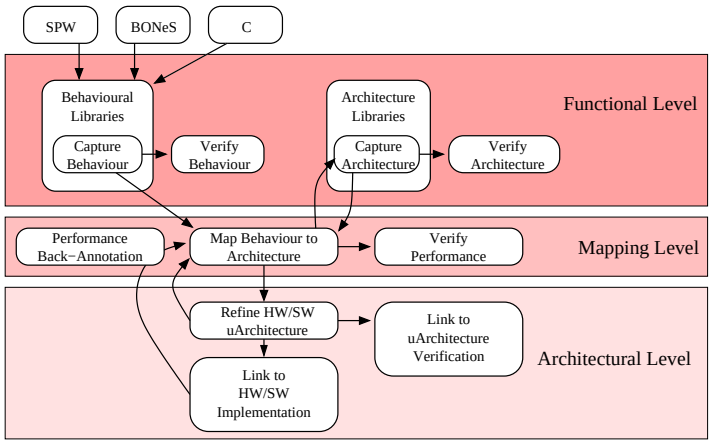
specific micro-kernel OS. It consists of a scheduler, to which the scheduling policy depends on the design, and I/O drivers, and is generated for each processor instance. Finally, it is necessary to interface different domains, i.e., SW procedures and co-operating circuits. Communication between software and hardware parts is achieved either through available I/O ports of micro-controller(s), or via memory-mapped I/O.

POLIS is mainly targeted for control-dominated ES's. Also, this is the methodology that is supported by a design environment and a set of tools (simulation, formal analysis, HW&SW synthesis)[14]. The work on POLIS is still in progress, and there is a commercial instance of the POLIS co-design methodology that will be described in the next section (see Figure 8).

The POLIS system does not accept Verilog or VHDL as input languages. Also, it does not support any other MoC except CFSM. Also, estimation techniques for more complex processor models, other than simple micro-controllers, are not supported. Architectures that can be synthesised with the POLIS system, consist typically of a single micro-controller unit and several ASIC's. This means that the POLIS system can not be used for designing ESs that rely on data flow as well.

2.4 *VCC* - Virtual Component Co-Design

Cadence *VCC* (*V*irtual *C*omponent *C*o-Design) is a toolset that is built based on the methodology presented in the prevoius subsection [7]. The POLIS project has highly influenced this methodology. The VCC toolset is based on communication refinement, as well as a segmentation of system verification requirements into distinct, complementary levels (see Figure 8). The general methodology is illustrated in Figure 8.



VCC by Cadence Design Systems

Figure 8: POLIS/VCC related methodology [7]

First, the functional behaviour of the entire system is captured and verified, thereby re-using elements from behavioural libraries and algorithmic fragments. In parallel or in sequence a target architecture is captured by re-using existing architecture IP components (DSP cores, micro-controllers, buses and RTOSs). After that, a manual mapping of behavioural functions and communication channels onto the appropriate architecture resources is done, and the system is evaluated via performance analysis in terms of constraints in the embedded system design (i.e., speed, power, and cost). The architecture, behaviour, and mapping for the given system specification are explored until an opti-

mal solution is obtained (back-annotations in Figure 8). Finally, the target architecture is refined into a detailed (HW and/or SW implementable) micro-architecture; thus, the refined target architecture can be passed to external environments for HW and SW implementations (the bottom in Figure 8). There is one important link which is not visible (at least not directly visible) in Figure 8, which is the co-verification back-annotation. It takes place between the architecture and the functional level, and serves for detecting timing and bug problems. The co-verification is not trivial, so, if exists, it has to be mentioned as a tool feature.

As with all commercial tools, *VCC* also tries to connect itself with the products of the same vendor, in this case Cadence. That is why in Figure 8 *SPW* (Cadence's *S*ignal *P*rocessing *W*orksystem) and *BONeS* (Cadence's system simulation kernel) are shown as the input behaviour and the simulation of performance models, respectively (for more information see [8]).

2.4.1 *COSY-VCC*

VCC is not confined to Cadence tools solely. As an example we will explain the flow introduced in the COSY (*CO*design *S*imulation and S*Y*nthesis) project [9]. In COSY, VCC is used in order to obtain an "infrastructure" for mixing and matching software and hardware components (IP's), while both the input behavioural specification and the refined target architecture synthesis are driven by non-Cadence tools (YAPI and TSS, respectively). Furthermore, the COSY-VCC approach substantially simplifies the design process. 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 9 shows the definition of the levels of communication in COSY-VCC.

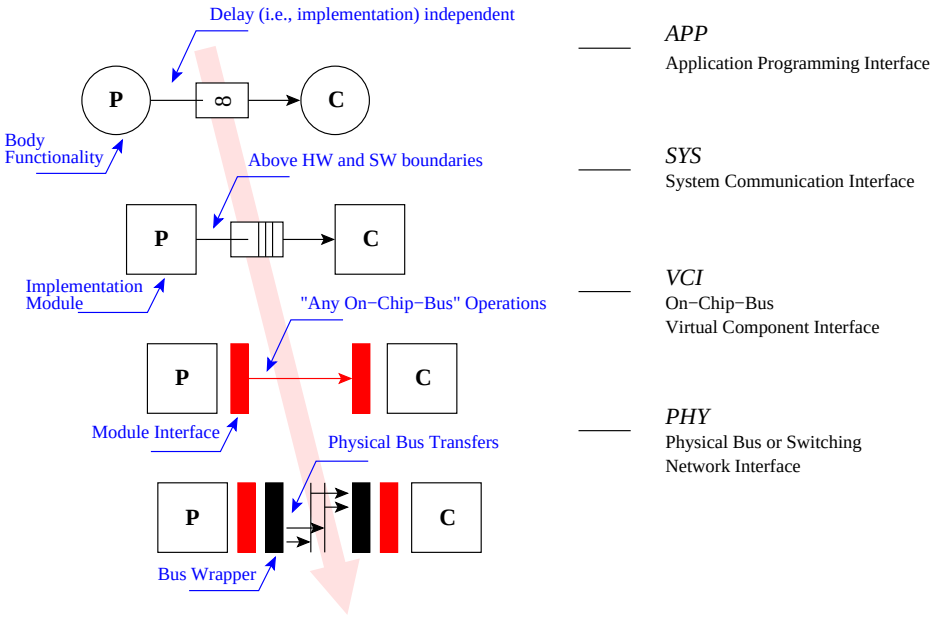


Figure 9: COSY-VCC related transaction levels [9]

Application level:
The COSY-VCC *APP* level in Figure 9 serves to implement application PN models. At the *APP*

level we are not concerned about communication and computation delays - we are only interested in the functional correctness of an application [9]. At this level we use an executable untimed specification, i.e., a YAPI model [10]. Untimed functional specification is given by means of the particular KPN MoC.

System level:

After mapping an application PN onto an architecture, *APP* transactions are refined into system transactions (the COSY-VCC *SYS* level in Figure 9). Functionally, *APP* and *SYS* transactions are equivalent (this is also true for all levels). However, at the *SYS* level we can implement certain functions either as software tasks on a programmable processor core, or as a dedicated coprocessor. That is why at this level we are able to perform a set of trade-offs (e.g., throughput vs. memory-size). On the other hand, at the *SYS* level transactions still operate on abstract data-types (e.g., video-frame and high-level I/O semantics), which makes them unrevealed into the more detailed interface [11]. Hence, the *SYS* level can be seen as "timed-functional" abstraction level.

Virtual interface level:

In order to deal with generic constraints related to on-chip communication between hardware components, a new level is required (the COSY-VCC *VCI* level in Figure 9) that obtains addresses and splits data in manageable chunks (manageable by bus or switching network). The COSY-VCC system integration flow relies on the simple *VCI wrappers* that translate the protocol used by *VCI* compliant interfaces into the physical protocol of the selected bus. This is effective by means of "IP behaviour from communication separation" [11].

Physical interface level:

This is the final level and deals with (real) physical bus size, signalling and arbitration protocol. This level is marked as *PHY* in Figure 9. This level should also produce a "solid ground reference" or a back-annotation link (see Figure 8) for calibrating and refining the estimation models relying on measurements at the "cycle-true/bit-true" level of existing systems.

The benefit of COSY-VCC is that the 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/she can effectively apply IP exchange because it is enabled by a clear separation of functionality and architecture, as well as the separation of IP behaviour and communication.

2.4.2 *VCC*'s Main Features

In comparison with other approaches, VCC has been innovative in a way that it integrates a number of technology developments. For instance, 80% of today's systems is software running on a general-purpose platform that is tightly coupled with unique dedicated hardware. Hence, a reasonable selection among programming languages produces results that make C/C++ an obvious candidate for a system level design language [12]. VCC provides several input formats for system behaviour, including for C/C++. Another example of technology development support is unifying heterogeneous control and data-flow models, as well as equal emphasis on system architecture and system behaviour. Looking at the heterogeneity of MoCs, it is important to mention that the system behaviour can be given with respect to many different MoCs, but all of them are actually executing on behalf of CFSMs that plays a dominant role of *meta* model of VCC. There are a few features that are crucial and should be stressed:

1. The abstraction level of architectural estimation models at a higher level than the current cycle and pin-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 (refinement).

The third feature is especially interesting and has been exploited by the COSY-VCC project. Also, VCC follows the *Y-chart* approach, which is a very powerful (high-level) design space exploration approach. However, independent functional simulation (independent of lower performance related levels of abstraction) is not possible. This means that a specification of system behaviour, governed by the MoC, can be simulated only jointly with a system performance model. **Remark:** This constraint is expected to be removed in the next *VCC* release; alternatively, in the VCC 1.2 release, independent functional simulation is possible when "artefact" behaviours are introduced (e.g., appropriate behavioural modules for FIFO).

2.5 *PAMELA* - Performance Modelling Language

Performance modelling of parallel computer systems can be done using a modelling language. One interesting representative among modelling languages is *PAMELA* [13]. It stands for *P*erformance *M*od*E*lling *LA*nguage and takes aim at two purposes:

1. More accurate - *dynamical* - model description (in comparison with *statical* models), and

2. A source language for a *static* performance modelling that yields an analytic (compile-time) model.

In order to achieve these conflicting goals (an explicit analytical form vs. a reasonable accuracy, and all of that with an extremely low cost solution), PAMELA has introduced some representation formalism restrictions with respect to the modelling of synchronisation. It uses "highly structured language operations"[13] to describe 4 factors that determine parallel system performance modelling: *c*onditional *s*ynchronisation - CS, *m*utual *e*xclusion - ME, *c*onditional *c*ontrol *f*low - CCF, and *b*asic *c*alibration - BC.

Another feature of interest is that PAMELA is a "procedure-oriented simulation language." "Simulation language" means:

- it naturally accounts for CS, ME and data-dependent CCF;
- it has inherited language advantage for providing simple compositional constructs for building large and parametric models;
- it offers dynamic performance evaluation near to actual system execution,

while "procedure oriented" provides notions of:

- structured synchronisation operators:
  - CS-related: *signal*(c) (signals appearance of the condition c) and *wait*(c) (waits for appearance of condition c),
  - ME-related: *use*(U, τ) (enters into a critical section, and acquires resource U for time τ), *using*(U) (generalisation of *use* construct, without time specifier), and *P/V* atomic semaphore operations (used for modelling and implementation of critical sections), and

– CCF: *par*(*i* = n,m) (parallel construct - i.e., executes each statement *i* in parallel with others) and *seq*(*i* = n,m) (sequential construct - i.e., executes each statement *i* according to the sequential order with others)

, and

- a time domain model, that is completely symbolic.

For instance, the execution time of program L is given as $T = \tau$. L = *delay*(τ) gives us a *delay* operation that increments virtual time τ (deterministic or stochastic according to some distribution). All together provide the time model.

Also, access to a shareable architectural resource from an application process can be described in three different ways, but all of them have the same meaning (time model with ME): *use*(U, τ) $\equiv$ *using*(U) *delay*(τ) $\equiv$ P(U) *delay*(τ) V(U).

An important reason why PAMELA was made as "procedure-oriented simulation language" is its "material-oriented" modelling approach. The main distinction between material-oriented modelling approach and its counterpart, "machine-oriented" modelling approach, is whether (shareable) resources, machines, of parallel computer systems are modelled as passive actors (i.e., **semaphores**) or not (server processes), respectively. Although both approaches are suitable for a parallel/concurrent system modelling, the machine-oriented one is less amenable to (compile-time) analysis, both with respect to CS and ME. The analysis is more complicated because the "message-passing" mechanism[3] that is implicated by machine-oriented modelling approach introduces non-determinism.

There are a few features that should be observed together with the tools and methodologies described:

1. Since PAMELA aims at doing static compile-time analysis, a trade-off has been made from accuracy point of view (although, the main target of PAMELA is not simulation, but estimation accuracy). That is why one might see PAMELA as an analysis and not as a performance modelling method. As an example, in comparison with other simulation languages PAMELA looses accuracy at the CCF-dealing part slightly, but gains more at the analysis part (In short: "static analysis with reasonable accuracy..."). An investigation was done [13] showing that the PAMELA model poses sufficient robustness in view of attractive cost features.

2. PAMELA does not distinguish separate formalisms to model programs and machines - there is no distinction between MoC and MoA. In that way, the *Y-chart* approach is not supported (at least not explicitly). One can have thoughts about finding some relationship between PAMELA and the *Y-chart* through the example given in Code 1. Notion of *delay* in Code 1 can be seen as an architecture model, and notion of *using* in Code 1 can be seen as a mapping. However, this is quite different from the *Y-chart* approach available in *VCC* or *SPADE*.

3. Although the material oriented approach brings a clear benefit for analysis, re-usability and extensiveness of models are not so clear as they are in the case of machine-oriented (object-oriented) modelling approach. This is sometimes not a sufficient argument, because it is very difficult to set some general patterns about concurrent systems anyway [3], and that implicitly means about MoA. Thus, the structural relationships between components can be captured easily, but describing (within a pattern) of how execution should occur is not trivial at all. At last, the main aim of PAMELA is not OOD, so looking into the support for re-usability and

[3] "message-passing" mechanism is the alternative to the PAMELA "procedure-oriented" mechanism

extensiveness of PAMELA models is probably more restrictive than SPADE models or Ptolemy models.

```
par (i=1, N)
{
using(R1) delay(τ1)
using(R2) delay(τ2)
...
using(RN) delay(τN)
}
```

Code 1: The pipeline model given in the PAMELA code; R_1, R_2, and R_N are pipeline cells that operate in parallel, and τ_1, τ_2, and τ_N give an architectural model - *delay*

2.6 *SystemC*

Originally intended for software design, neither C nor C++ programming language has the basic support required for accurately dealing with the hardware parts of a SoC design [17]. There are two ways to remedy this lack: either to build syntax extensions, or to introduce specific class libraries. *SystemC* follows the second path. Currently, SystemC has wide support both from commercial and academic side, and tends to become a standard as a language based modelling technique (probably because its C++ ability to manage ES complexity and re-use through OOD).

With SystemC, ES can be described by means of multiple concurrent processes. The underlying simulation kernel of SystemC is built ontop of a co-routine based multithreading library, which directly implies two benefits:

1. Each component is represented by means of the concurrent thread (that directly implies that concurrency of a parallel HW is captured in the thread model), and
2. Both properties "multithreading" and "co-routine based" are directed into a small simulation-time cost (meaning, faster simulation).

By providing a C++ class library based implementation of objects like processors, ports, signals, hardware data types, and many others, almost all kind of communication mechanisms are supported. Especially interesting is the *R*emote *P*rocedure *C*all (RPC[4]) mechanism, that is used to model "master-slave" communication at the high level abstraction models. Moreover, version 2.0, which is supposed to be available as β version at the end of April 2001, is going to support user defined types of communication[5].

Apart from the already mentioned SystemC object-types, there exist two additional and particularly important object-types: *modules* and *processes* (or *threads*). Modules usually encapsulate some component (either processing element or communication block), and can contain another module or process (meaning, OOD related hierarchy and containment). Processes serve to capture functionality, and can be reactive either to any input signal or to a clock. Also, processes can be synchronous (they include timing control statements or conditional synchronisation, like *wait*(*delay*)[6] and instructions between *wait*s are timeless), or asynchronous (instructions are timeless, and local variables are redefined each time the process is invoked) [15].

[4]RPC is well known in Operating Systems Community as one of the IPC (Inter-Process communication) mechanisms.
[5]Information was collected from [15]; unfortunately, the announced deadline is already broken.
[6]version 2.0 supports composite conditional statements, like *wait*(*C1* || *C2* ...)

Another benefit of the library based implementation of HW 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, as opposed to SPADE [1], where two different simulation environments are involved.

System C distinguishes several levels of abstraction:

L-1. *Untimed functional*: control/dataflow MoCs

L-2. *Timed functional*: behavioural processes,

L-3. *BCA/CA functional* - bus-cycle accurate/cycle accurate.

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 (components) at the instruction level, while others are at higher functional levels.

SystemC is useful as a support for methodologies and tools, because SystemC itself might not be considered a methodology; It is a modelling language. Specifications given within the SystemC context are executable (model verification is possible). It supports algorithmic and behavioural modelling of both HW and SW, and has support for low level HW models (RTL). Because of a broad support, SystemC tends to become a standard, which would empower re-use and sharing of models.

3 SPADE

In this section we present a concise description of the SPADE methodology and the SPADE software structure. The presentation of the software methodology is kept very short as a detailed expose of this aspect of SPADE is beyond the scope of this document. Moreover, the SPADE methodology is well documented elsewhere [1],[20],[23],[24]. In the part of the section that deals with the SPADE software structure, we have also avoided giving lengthy explanations: we use UML to describe the software structure and we assume that the reader is somehow familiar with that language. If not, we recommend you to find an introduction to UML[7].

The rest of the section is organized in two main sub-sections: The SPADE Methodology (Subsection 3.1) and The SPADE Software Structure (Sub-section 3.2). The last sub-section is subdivided into eight parts: Structure Layers (3.2.1), Top Layer (3.2.2), Application Subsystem (3.2.3), Architecture Subsystem (3.2.4), Mapping Subsystem (3.2.5), Trace Subsystem (3.2.6), TSS Subsystem (3.2.7), and Metrics Subsystem (3.2.8).

3.1 The SPADE Methodology

SPADE stands for "*S*ystem-level *P*erformance *A*nalysis and *D*esign-space *E*xploration". It is both a methodology and an implementation for high level system modelling and design. The SPADE methodology follows the *Y-chart* approach introduced in [19]. The Y-chart paradigm is summarized in Figure 10. The Y-chart approach allows for quantification of high-level design choices by measuring performance through simulation.

We now briefly comment on the various blocks in Figure 10.

Applications:

[7] We have used tutorial referenced as [18]

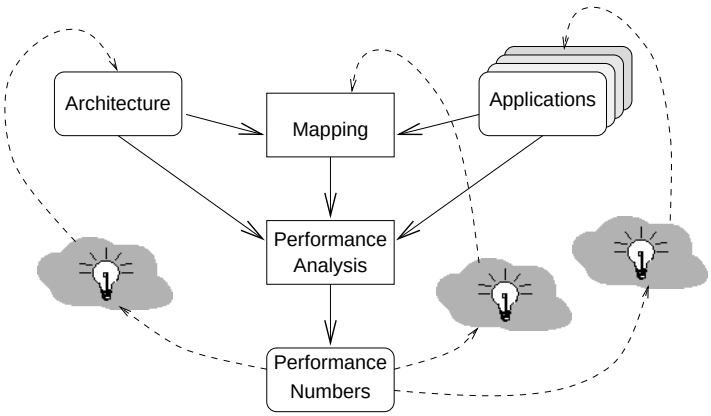


Figure 10: The Y-chart (picture is borrowed from [20])

SPADE application model is the Kahn Process Network - KPN (”Y-chart Application Programmer’s Interface”, or YAPI library [10], serves as KPN implementation in SPADE application modelling part). Thus, the application is modelled as a Process Network. Although processes can run in parallel, internally each of them is sequential. Processes communicate by token-exchange via unbounded channels. If a process tries to read from an empty channel it will block. It can continue only after the token it expects in a channel has arrived. On the other hand, any process can write to channel without blocking [21].

Architecture:

SPADE architecture model is based on network of generic building blocks. Since they are generic, they accept behaviour from an application model. Actually, an application model ”drives” the architecture model via Trace Driven Execution (TDE)[22] method (will be explained in the next paragraph). Additionally, architecture buffers (or ”FIFOs”) are bounded. This opens scheduling issue in the SPADE methodology. The reason is that wrong scheduling can result in deadlock. Due to previous, SPADE obtains scheduling of atomic architecture operations originated from traces. Each operation can be either communication load or computation load, and is visible in generic architecture block like delay, resource locking or both.

Mapping:

Each process that is mapped on a particular architecture block (unit) generates a trace, which will drive that block (earlier mentioned TDE). Traces are stamps of each read, write or execute activities of the mentioned process. In that way behaviour of the application KPN is mapped on architecture model.

Performance Analysis:

Performance analysis can take place at different level of detail and designers should exploit these different levels to narrow down the design space of architectures in a stepwise fashion [19]. This implies that evaluation of architectures is relatively inexpensive.

Performance Numbers:

After the execution of SPADE some performance numbers are available like: 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 designer what is the bottleneck or which parts of the architecture are under-utilised. Previous corresponds to the bottom of the Figure 10 from where feedback with light bulbs is coming.

Those main items in the Y-chart will show-up as subsystems in the SPADE software to be presented in the next section.

3.2 SPADE Software Structure

SPADE is implemented in a single program that uses different libraries. These libraries encapsulate the underlying methodology and allow an efficient multithreading mechanism. Furthermore, SPADE has been written in the C++ object oriented language. Previous should obtain easy extension, reusability and maintainability of the tool source code. For illustration and documenting the software structure and functionality, UML (Unified Modelling Language) will be used [18].

Before we say more about the SPADE software structure, let us first look in the specification requirements that are given if Figure 11. An UML use-case diagram is used for a specification documenting. Figure 11 shows that the required functionality of the SPADE software does not cover a really wide range. Also, the software does not involve a large amount of actors. It only identifies "user" as a *principle actor* and PAMELA as a *secondary actor*; there is no need for *external hardware* actors, as well as for actors representing *other systems* [18].

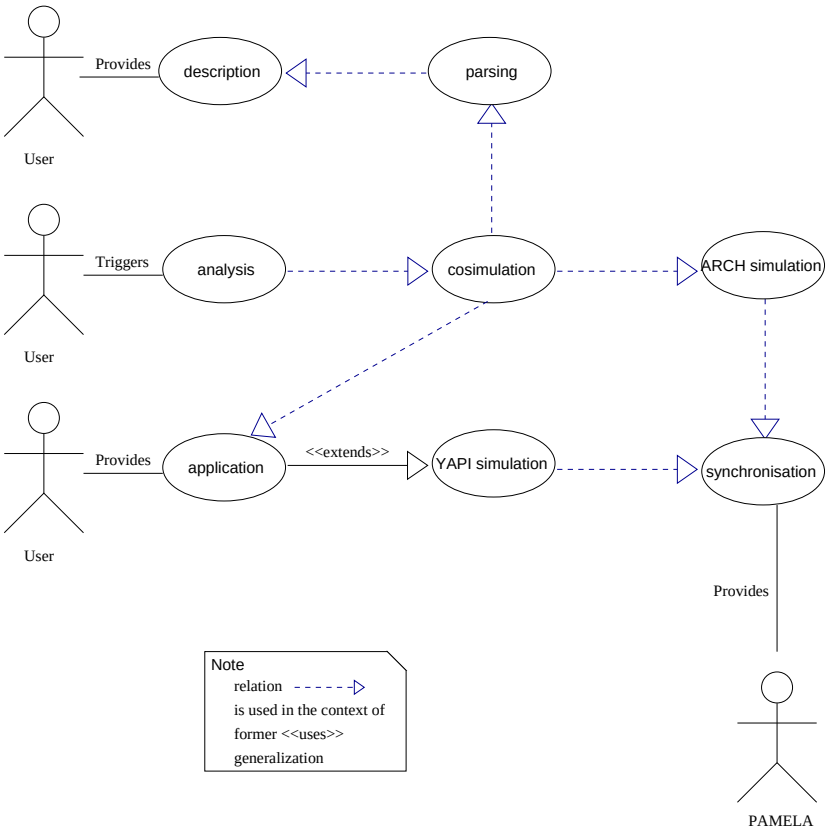


Figure 11: SPADE software from the user standpoint - actions & reactions

3.2.1 Structure Layers

The SPADE software structure can be examined in vertical and horizontal direction: The software has a layered structure (vertical), and each layer can be partitioned either in subsystems or classes/objects, (horizontal direction). See Figure 12.

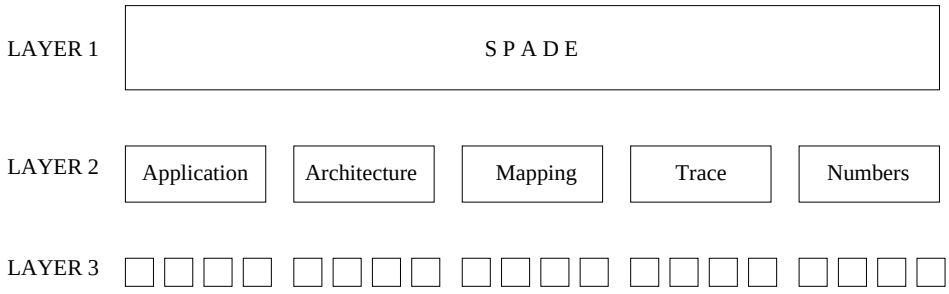


Figure 12: SPADE software architecture

3.2.2 The Highest Layer

The top layer (or L1 from Figure 12) gives the highest level of description of the SPADE software structure. This is presented in Figure 13 as the Spade subsystem (in UML notation). The executive program is represented as ”spade.exe”. Its source code (given in ”main.cc”) is straightforward sequence of activities:

1. Parsing command line options,
2. Running simulation,
3. Cleaning up.

During the first and partly the last task, singleton objects derived from classes SpadeOptions and SpadeEnvironment are involved. The main job (the second task) is delivered to singleton object derived from class Spade.

Spade object encompasses the Y-chart methodology. This is obvious from aggregation links to objects derived from classes SpadeApplication, SpadeArchitecture and SpadeMapping. In that way Y-chart entities (application, architecture, mapping) are in 1-to-1 fashion replicated into the implementation.

The Spade class is a singleton, meaning that there can be only one Spade object. This class has an *initialize* member-function that can initialise the object only once. Its job is to:

- Initialise the singleton Spade object.
- Initialise the PAMELA multithreading environment [13], and to set a new PAMELA trap handler.

The main method of the Spade object is the *run* function. Its job is to start a simulation as follows:

- First, local pointer to singleton object Environment is assigned.
- Second, application is constructed and instantiated according to the arguments that have been obtained via Environment object. At this point KPN processes are created (but not able to run).

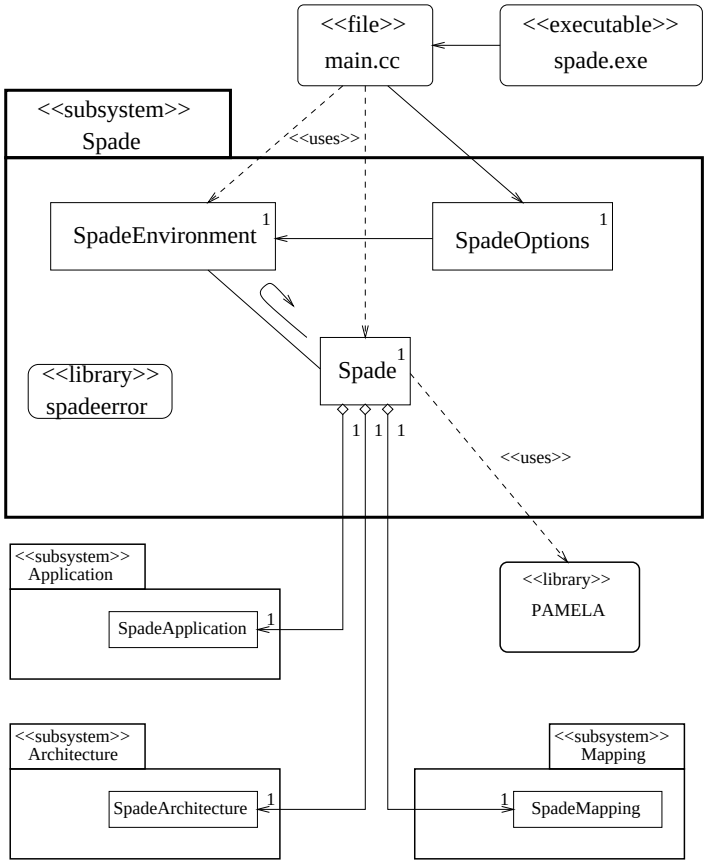


Figure 13: Top-level SPADE view on software architecture

- Then (depending on options) either a ”dotty graph” is put on standard output device, or the application is run without architecture model. Otherwise, the architecture and the mapping are constructed and instantiated. At this point the architecture and the mapping description files are parsed. The hardware-simulator thread (TSS) is in ready state.
- High-level system simulation is started. This has been done by the explicit call to the *start* member-functions from SpadeApplication (Figure 3.2.3) and SpadeArchitecture objects.
- SpadeArchitecture object’s member-function *wait* is called in order to block until hardware-simulator thread unlocks semaphore. This is done only when the user exits TSS simulator (**Remark:** Call of the SpadeApplication object’s member-function *wait* is avoided; the reason that there is no need to wait application to finish once user exits TSS simulator).
- Finally, the status of the YAPI FIFOs is printed.

So far, we have described the software structure and behaviour of SPADE at the top level of the layer hierarchy. The next subsections are about subsystems from the layer 2 (Figure 12).

3.2.3 Application Subsystem Structure and Behaviour

In SPADE application subsystem is implemented with full respected to the KPN [20]. Actually, most of the things are inherited from the YAPI library (except for the port-polling mechanism introduced via the *select* function). Unfortunately, the structure of application subsystem is not ”the clearest” and ”the simplest” one. It is shown in Figure 14. Figure 14 contains maybe more information than it should (e.g. STL - ”Standard Template Library” blocks like *vector*, *map*, *set* and *list*). However, in this way the actual software structure is better documented.

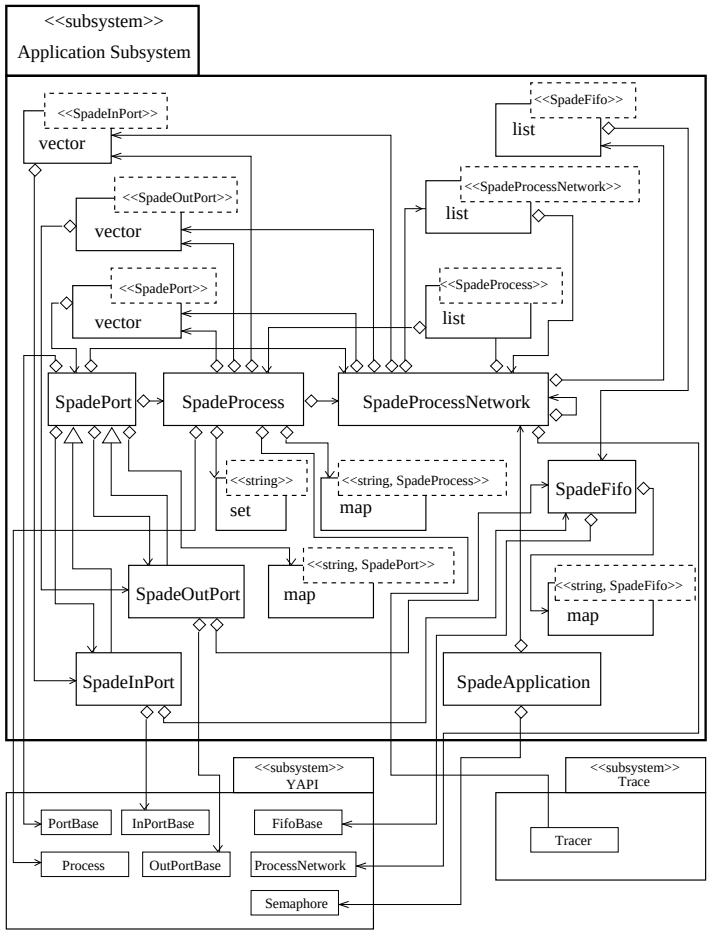


Figure 14: Application subsystem’s structure

It can be seen from Figure 14 that each process is implemented as a YAPI process. What is not explicit in the diagram is that each YAPI process is implemented as a PAMELA thread. From the UML activity diagram in Figure 3.2.3, one can see a chaining of activities (methods) during the execution of SpadeProcess’s *main* method. Another substantial thing that is visible in Figure 14 is that there exists mutual relation between class Tracer from the SPADE Trace subsystem and class SpadeProcess from the SPADE Application subsystem. These should suggest how architecture is driven from application - i.e. via traces (*ApplEvent*s in Figure 3.2.3).

From a simulation point of view each parallel process is represented by one object. This object is instantiated from class SpadeProcess. Class SpadeApplication is responsible for instantiation and

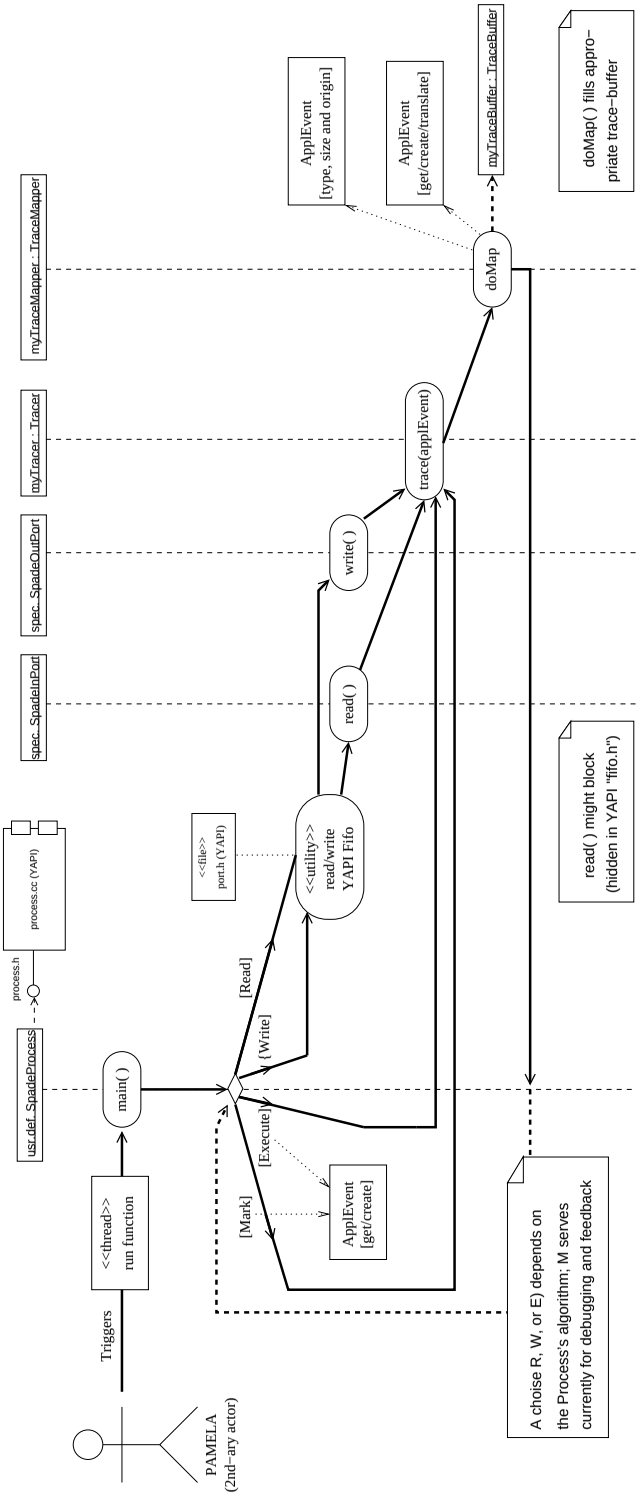


Figure 15: Behavioral description of class SpadeProcess *main* functionality: 1 iteration step only, which is to be repeated

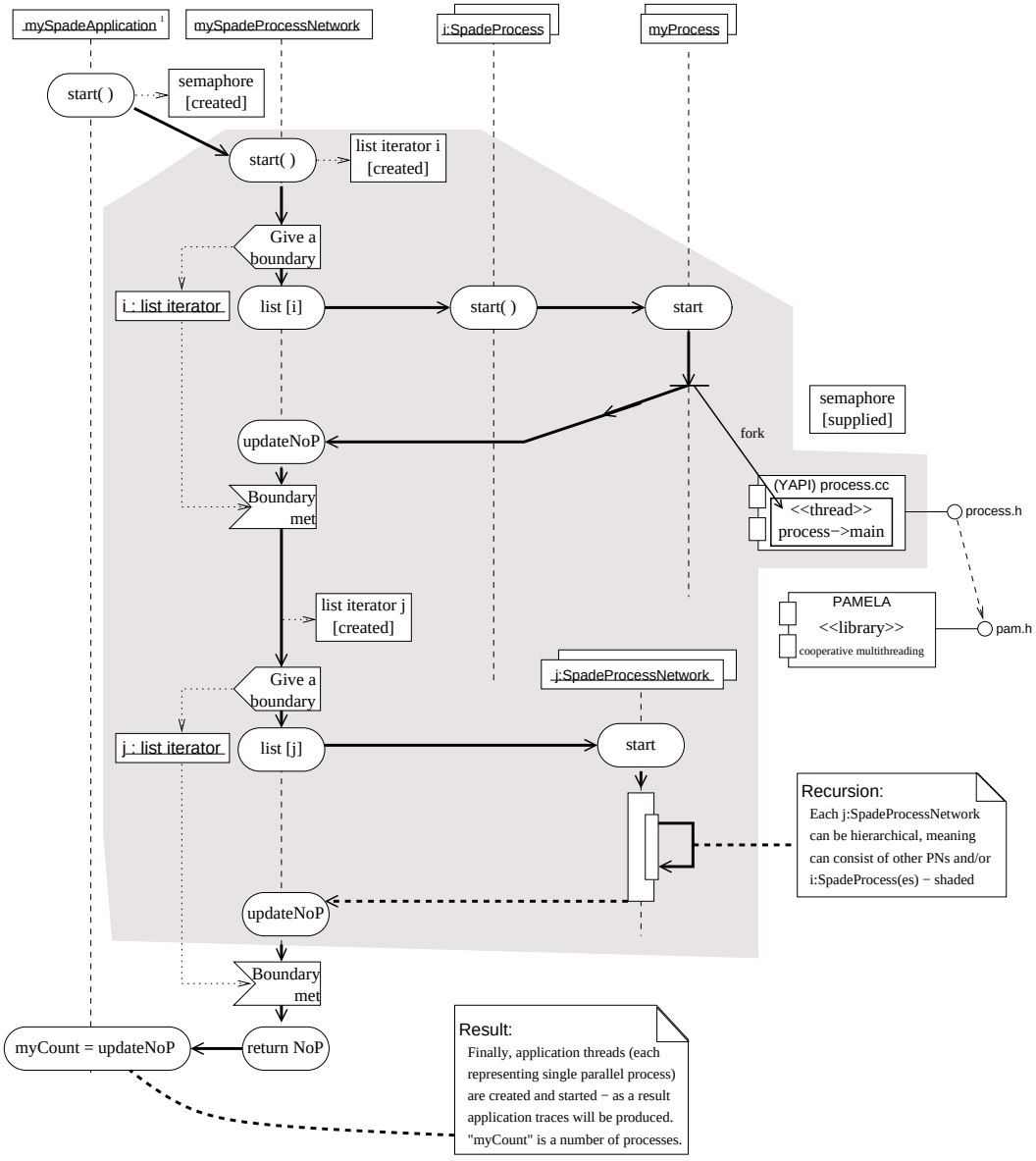


Figure 16: Behavioral description of class SpadeApplication *start* functionality

initialisation of KPN (*instantiate*), running (*start*), synchronisation (*wait*), and safe stopping of application simulation. Particulary interesting is how SPADE is going to run an application. Due to the previously mentioned, we focus on a behaviour of *start* method (Figure 16). By means of the shaded part of Figure 16 we have indicated the following recursion (see UML-Notes in Figure 16): a SpadeProcessNetwork object can be a network of SpadeProcess objects and SpadeProcessNetwork objects.

3.2.4 Architecture Subsystem Structure and Behaviour

The previous subsection has given a short overview of SPADE application subsystem. Here the software structure and behaviour of the SPADE architecture subsystem is discussed. The architecture subsystem has a structure as shown in Figure 17. The main object is the one derived from class SpadeArchitecture.

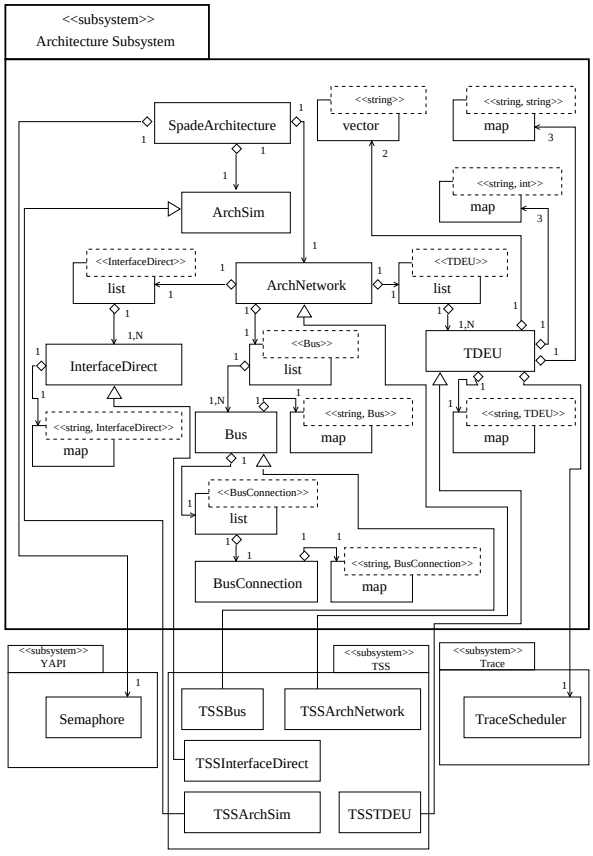


Figure 17: Architecture subsystem's structure

If SPADE is to be run with full simulation requests (and not only with "application stand alone" option), the architecture subsystem is involved. The function that is called first is the one that creates everything for architecture (*SpadeArchitecture::instantiate*). Its job is to first create the ArchNetwork object. It then calls the *parse* member-function from the ArchNetwork object which parses the architecture description (".arch") file and generates the corresponding data-structures and their linkage. Also, in the *instantiate* member-function of the SpadeArchitecture object the corresponding hardware-simulator object is created. This is done by the ArchSim object (More precisely, the TSSArchSim object). The UML activity diagram in Figure 18 illustrates previously elaborated behaviour of the *instantiate* member-function.

At this point, the architecture objects are created but not in an active state (the TSS PAMELA thread is not existent yet, but the TSS simulator object is; similary, the application objects are initialised, but the application threads are not available yet). The next thing that should be done is the parsing of mapping description (".map") file and generating the appropriate mappings/relations be-

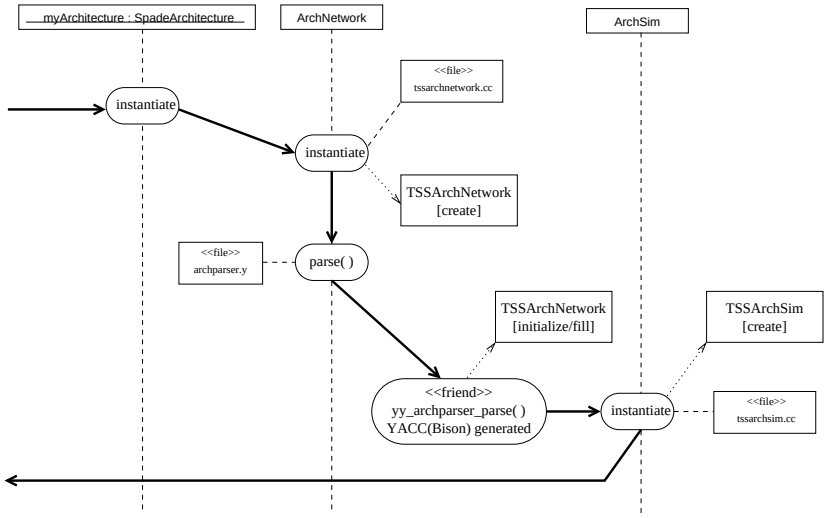


Figure 18: Behavioural description of class SpadeArchitecture *instantiate* method

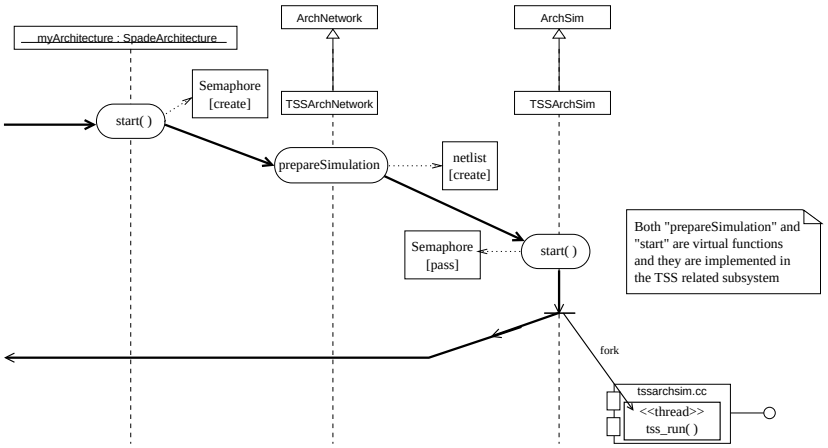


Figure 19: Behavioural description of class SpadeArchitecture *start* method

tween application process network and the corresponding hardware blocks (TDEUs - ”Trace Driven Execution Units”; FIFO buffers, busses). This too has been visualized in Figure 18. Once this is done, the simulator can be started (*SpadeArchitecture::start*). Just before the TSS simulator is brought in the running state, its ”netlist” input file will be generated. This is the responsibility of the *prepareSimulation* member-function from the ArchNetwork object (see Figure 19). Note that the various TSS objects which are being used, have been created in the initialization of Figure 18, as specializations of architecture objects. The mapping subsystem is the subject of the next subsection.

3.2.5 Mapping Subsystem Structure and Behaviour

This subsystem is exactly the translator of how and which parts of the hardware and software models are connected. The mapping convention is described in detail in [23]. The mapping subsystem

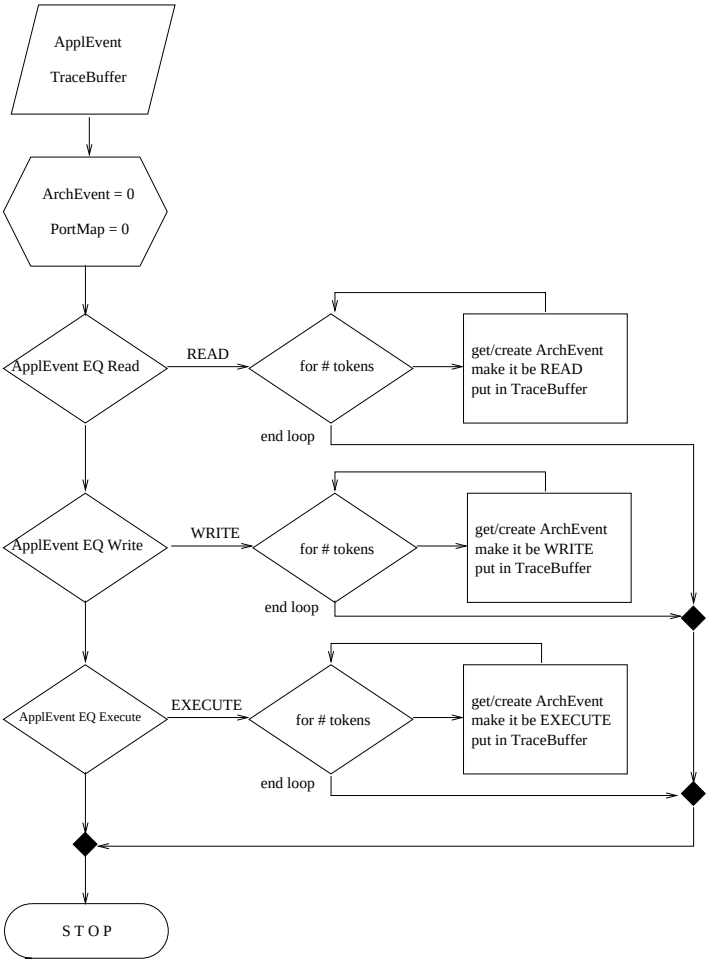


Figure 20: *doMap()* member-function of the TraceMapper object responsible for trace mapping

does not have the complex structure and functionality. It simply follows the Y-chart concept. The main object here is the one derived from class SpadeMapping. Another class from this subsystem that is particularly of interest for SPADE software structure and behaviour is the class TraceMapper that implements the specific linkage between application and architecture traces. For that purpose, TraceMapper object uses knowledge obtained from the mapping description (".map") file. Actually a member-function from this object, called *doMap*, is implementing the transformation from application trace (denoted in the SPADE implementation as a class ApplEvent) into the architecture trace (denoted in the SPADE implementation as a class ArchEvent). It fills the trace buffer that is associated with the process that has generated the trace. The *doMap* function is shown in Figure 20. As can be seen the current situation is that there is 1-to-1 correspondence between application traces and architecture traces[8].

Parsing of the mapping description (".map") file is done at creation time of the SpadeMapping object. The *parse* member-function of the SpadeMapping object will be executed as a last activity of the *instantiate* member-function of the same object. The *parse* member-function of SpadeMapping object

[8]This is only for the case of non-vector *read* and *write* application operations; for the vector situation, there will be more architecture operations resulting from a single application operation.

is the one that will make software ”prepare to work” according to underlying SPADE methodology (i.e. Y-chart + KPN + TDE).

3.2.6 Trace Subsystem Structure and Behaviour

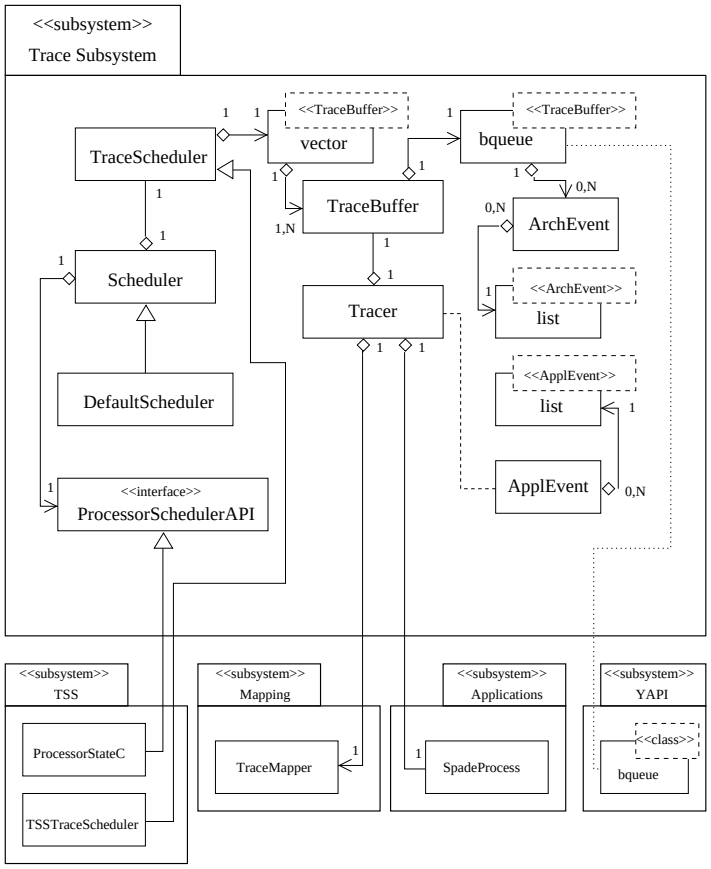


Figure 21: Trace Layer’s structure

The most interesting subsystem of SPADE is the trace subsystem. It implements the TDE (“Trace-Driven Execution”) concept described at the beginning of the document. This is the place where actual scheduling of the architecture traces is done. The trace subsystem represents the intermediate layer (“neither in application nor in architecture”). The structure of the trace subsystem is shown in Figure 21.

As can be seen from Figure 21, a Tracer object will accept all application trace entries, and, with the help of a TraceMapper object from the mapping subsystem, will fill the corresponding Trace-Buffer object (each parallel process has its own Tracer object and hence also TraceBuffer object). The TraceScheduler object that is called by the corresponding TSS thread, will ”empty” the Trace-Buffer object that it selects according to the scheduling policy (scheduling issues related to SPADE are described in [24]). Currently, only DefaultScheduler is implemented, and its policy is as follows:

- If the trace buffer that was active at the previous time instant contains an execute, use that trace buffer and select the execute.

- If the trace buffer that was active at the previous time instant contains write, and write is not blocking, select this write.
- If the trace buffer that was active at the previous time instant contains read, and the previous entry was not read, switch to another trace buffer if possible.
- Trace-buffer selection goes in Round-Robin (RR) fashion.

For more information related to scheduling, see [24].

3.2.7 TSS Subsystem Structure and Behaviour

Since SPADE has to obtain some hardware simulation performance numbers, a hardware simulator tool has been included. Currently, this simulator is TSS (”Tool for System Simulation”), which is a Philips in-house simulator tool. TSS has a complex API and it serves for low-level simulation (BCA - *B*us *C*ycle *A*ccurate); that is the reason why the TSS-related SPADE subsystem contains a lot of source code. For example, the conceptual level system description (given in the earlier mentioned architecture description “.arch” file) is mapped onto BCA level components-&-interfaces and communication network/bus. Each BCA level module involves appropriate FSM (*F*inite *S*tate *M*achine) behavioural description, as well as, the module structure description (see Figure 22).

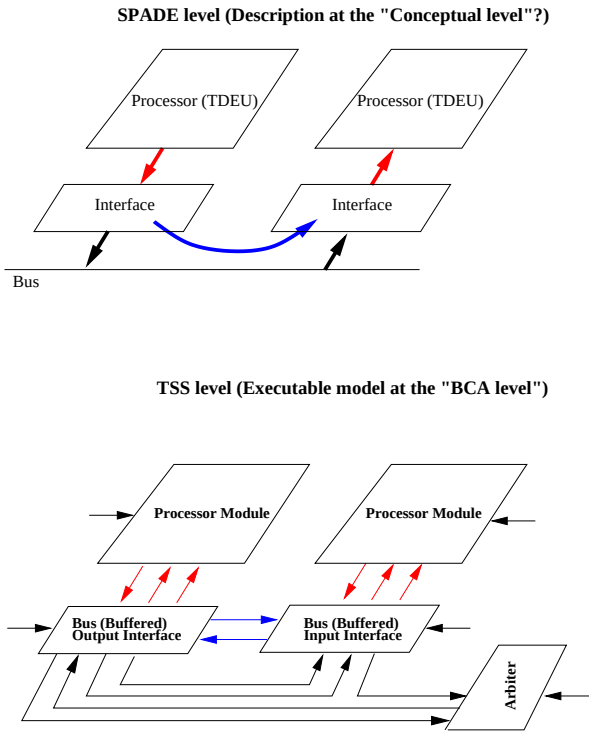


Figure 22: A conceptual architecture description and corresponding TSS model

Basically, the TSS software subsystem is also partitioned. From one side, there is TSS-library support for different target-platforms (operating systems). On another side, there are partitions with TSS specific SPADE-code, as well as library of TSS modules. These modules are the ones that will

be instantiated from parsed SPADE architecture. They will be driven by traces from the SPADE application. The mechanism of trace scheduling is inherited and implemented here. Figure 23 illustrates previously mentioned.

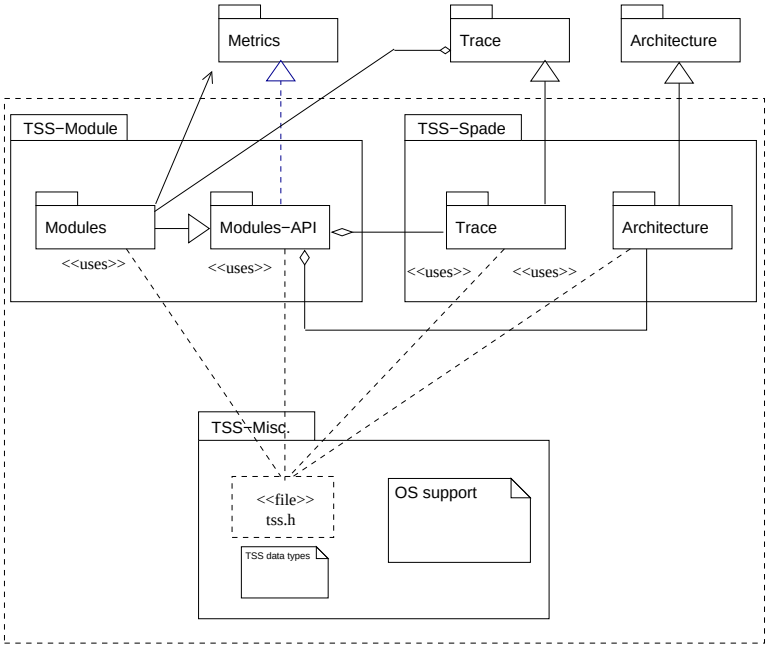


Figure 23: Pseudo-UML diagram of the TSS subsystem and its relations

TSS-Miscellaneous subsystem takes care of TSS-platform support. The TSS-Modules subsystem takes care of the implementation of the different module behaviours that are needed in SPADE. The TSS-Spade subsystem is responsible for two things: SPADE-to-TSS trace scheduler and SPADE-to-TSS architecture interfaces.

Due to the complexity of the TSS-part of the SPADE code a separate detailed description of it should be presented. However, to avoid detailed presentation in this document, we will suffice with the top level diagram shown in Figure 23. The important thing here is to observe what is the ”typical” way of interfacing the TSS simulator with the rest of the SPADE software. One interesting question is whether the TSS simulator could be “easily” replaced by another simulator, i.e., how tightly TSS is coupled with the rest of the SPADE software.

3.2.8 Metrics Software Subsystem

The name the software subsystem suggests that it should generate ”Performance Metrics”. However, the current SPADE output cannot be named metrics. That is why ”Performance Numbers” has been used as description title of corresponding step in the Y-chart representation in Figure 10.

From the implementation software point of view, the metrics subsystem has only three classes: class ArbiterMetrics, class ProcessorMetrics, and class ProcessorPipeMetrics (**Remark**: class ProcessorPipeMetrics does not work). These classes serve as super-classes of appropriate TSS-modules. Also, they implement methods for obtaining the performance numbers of a particular processing unit and its ports, as well as information about the bus arbiter. These methods are extensively used later on, during the hardware simulation with TSS.

The performance numbers can later on be visualized, analysed, and used for generating metrics information. Right now, software does not support this (metrics generation) step. Also, the accuracy information has not been made available yet.

3.3 The issue of TSS replacement

In section 3 we have been mainly introducing the SPADE methodology and the SPADE software structure in terms of UML diagrams. We have introduced all SPADE subsystems, although the details of the TSS subsystem have been left out. This is not to say that these details are of no importance; in fact they are of importance but only in relation with the question whether TSS - which is a Philips property - could be replaced with another tool, e.g., SystemC [15]. It is precisely the details of TSS, that enter the coupling issue of TSS with the rest of the SPADE software; YAPI/TSS is essentially a mixed level simulation environment. Although the objective was to perform system level design space exploration at a performance estimation level of abstraction, the implementation of this level is tightly coupled to the cycle-accurate lower level TSS simulator (see Figure 22). As a result, even at the higher level of abstraction a lot of details have to be incorporated because TSS requires so. These details are hidden for the higher level exploration, yet they should not be needed at a time performance estimation level, where DE simulators could run much faster. While this is a drawback, an advantage is that in this environment it is relatively easy to perform mixed level simulation and exploration.

Due to the previous, we have raised the "replacement" question in sub-section 3.2.7. The answer of thia question is not short and easy, and it is related with the following items:

- Looking into the SPADE sources and the related UML diagrams it is clear that the main objective is to re-write the TSS-related modules, interfaces, and APIs respecting features of another simulator.
- The "implementation issues" like the underlying OS, multithreading library, and the simulator specific APIs can highly influence "easiness" of the TSS replacement.
- Directions (extensions, improvements, etc.) of a future work on the SPADE environment are very tightly coupled with a simulator choice, meaning that some hardware simulators and their libraries do not support what we intend to do in the future.

First, coupling between TSS and SPADE can be recognized and followed relatively easy. However, the SPADE methodology relies on specific TSS modules, which are implemented within a large code (often not easy to read and understand), and that complicates things. Conclusion is that no matter which HW simulator we choose we will always have to re-write these TSS specific code.

Second, now both an application (a YAPI model) and an architecture (a TSS model) use a same multithreading library (PAMELA) during the simulation. In order to keep the things simple we have to follow the previous strategy. This means that if the chosen HW simulator is not implemented over the same multithreading library as YAPI is implemented, we will have to re-write YAPI in order to have it supported. This might become a non-trivial job. **Remark:** According to [25] and [26] YAPI on SystemC will be available soon and then the second judgement will not stand.

Finally, a further SPADE development flow is correlated with the heterogeneous signal processing system trends that are currently of interest. Particularly, the last bullet suggests that we need the results from the state-of-the-art exploration (Section 2.) in order to choose the right path. We will summarize these results in the next section.

4 Conclusion

A study was conducted to investigate different features of some ES design methodologies and tools that are available today. Although they are all created and used in the same field (system-level EDA), there exists wide diversity among them.

A total of 8 different methodologies and tools was studied[9]: UC Berkeley Ptolemy, Ptolemy-based co-simulation, UC San Diego/NEC methodology, POLIS, Cadence VCC and COSY/VCC, PAMELA, SystemC, and SPADE methodology. In Table 1 we summarise their most interesting properties. Most of the tools are created as non-commercial ones (first row in Table 1), or are available in one way or the other (can be bought or downloaded - second row in Table 1), and that was decisive to put them in focus of our exploration.

The study found that presented methodologies and tools differ from each other in their approach to HW/SW co-synthesis (Y-chart support - third row in Table 1), as well as in their use of MoCs (fourth row in Table 1). Consequently, there exists a ”fuzzy” space according to the MoA library support (fifth row in Table 1).

Although the differences among the seven tools are not absolute, the features shown in Table 1 indicate that the most preferable kind of methodology is *COSY/VCC*-like, because it has the largest amount of positive marks (”+”). The features shown in Table 1 also do suggest (1) that today’s ES-design methodologies and tools are not mature enough, and thus, there is (2) a plenty room for further research (a synthesis support, as well as for supporting all abstraction levels, and formal analysis and verification through all co-design flow). Furthermore, there exists a wide exploration space, especially according to the complex designs and mixed abstraction level support (rows 9 and 11 in Table 1).

[9]There are more methodologies and tools for system design available, but we have narrowed the scope to capture the most representative ones.

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

Methodologies & Tools ↦	*Ptolemy*	*Pt. based co-sim.*	*UC S.Diego/NEC*	*POLIS*	*VCC(COSY)*	*PAMELA*	*SystemC*	*SPADE*
Commercial	-	-	-	-	+	-	-	-
Available (can be bought or downloaded)	+	-	-	+	+	+	+	+
Y-chart supported	-	+	+	+	+	+	o	+
MoC variety supported	+	+	-	o	+	-	o	-
Library MoA variety supported	-	o	o	o	+	-	o	+
Dynamic performance models	+	+	+	+	+	+	+	+
Formal analysis and verification	-	-	o	+	o	+	o	-
Reusability supported	+	+	+	+	+	o	+	+
Complex designs supported	o	+	o	-	+	-	+	+
All abstraction levels supported	-	o	o	+	+	-	+	-
Mixed abstraction level simulation supported	-	o	-	o	-	-	+	o
Synthesis supported	-	o	+	+	o	-	-	o

Legend	
True statement (e.g., supported)	+
False statement (e.g., unsupported)	-
Fuzzy (either both + and -, or “possible”)	o

5 References

[1] P. Lieverse et al., ”A methodology for architecture exploration of heterogeneous signal processing systems”, 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,” 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 multi-channel 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] Jean-Yves Brunel et al., ”COSY: levels of interface for modules used to create a video system on chip,” *EMMSEC’99*, Stockholm, 21-23 June 1999.

[10] De Kock et al., ”Y-chart Application Programmer’s Interface”, Technical Report, Philips Research, January 2000.

[11] Jean-Yves Brunel et al., ”COSY Communication IP’s,”

[12] Guido Arnout, ”C for System Level Design,” white paper, CoWare, Inc.

[13] Arjan van Gemund, ”Performance Modeling of Parallel Systems,” PhD thesis, Technische Universiteit Delft, the Netherlands, 1996.

[14] L. Lavagno, A. Sangiovanni-Vincentelli, et al., ”Hardware/Software Co-Design Of Embedded Systems - The POLIS Approach,” Kluwer Academic Publishers, 2nd printing, 1999.

[15] Synopsys, Inc., CoWare, Inc., Frontier Design, Inc. and others, ”Functional Specification For SystemC 2.0 - Final”, January 17th, 2001.

[16] V. Živković, ”Memo2: Adaptive SPADE Case”, February 23rd, 2001.

[17] D. Verkest, J. Kunkel, F. Schirrmeister, ”System Level Design Using C++,” white paper, IMEC - Synopsys - Cadence.

[18] P.A. Muller, ”Instant UML”, ISBN 1-861000-87-1, Wrox Press Ltd., 1997.

[19] A.C.J. Kienhuis, ”Design Space Exploration of Stream-based Dataflow Architectures *Methods and Tools*”, PhD thesis, Technische Universiteit Delft, the Netherlands, 1999.

[20] Stefanov et al., ”Y-chart based system level performance analysis: an M-JPEG case study”, Progress Workshop, 2000.

[21] G. Kahn, ”The semantics of a simple language for parallel programming”, Information processing 74 - North-Holland Publishing Company, 1974.

[22] Uhlig et al., ”Trace-driven memory simulation: a survey”, ACM Computing Surveys, Vol.29, No. 2, June 1997.

[23] P. Lieverse, ”Memo17: Architecture and Mapping Description Languages”, March 22nd, 2000.

[24] P. Lieverse, ”Memo12: Many to one mapping and scheduling”, September 1999.

[25] Stuart Swan, ”An Introduction to System Level Modeling in SystemC 2.0”, White paper, Cadence Design Systems, Inc., OSCI, May 2001.

[26] De Kock et al., ”Proposal for Modeling Kahn Process Networks and Synchronous Dataflow in System-C”, White paper, Philips Research ft. Cadence Design Systems, Inc., April 2001.

[27] 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.