JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 2, March-April 2009

UML for Modelling and Performance Estimation of Embedded Systems

Fateh Boutekkouk, University of Constantine. Algeria
Mohammed Benmohammed, University of Constantine. Algeria
Sebastien Bilavarn, University of NICE, Sophia Antipolis, 06000, France.
Michel Auguin, University of NICE, Sophia Antipolis, 06000, France.

Abstract

In this paper, we would like to present a new UML-based methodology for embedded applications design. Our approach starts from a pure sequential object paradigm model from which a task level model is extracted. The latter allows designer to expose all parallelism forms such as task parallelism, data parallelism, pipelining, while making control and communication over tasks explicit. Another particularity of our approach is hardware parameterization-based abstraction in which hardware platform is modelled as a set of generic components. Each component is parameterized by a set of abstract parameters matching the abstraction level of application. An estimation technique of performance is proposed. Since we are dealing with higher level of abstraction, the values of these metrics are not absolute, rather than, they are relative in the sense, we will use them to compare between possible alternatives.

1 INTRODUCTION

Embedded systems (ESs) are becoming increasingly important and ubiquitous. Most of recent Embedded Systems are capable of executing very complex algorithms ranging from control, telecommunication to multimedia high performance applications.

ESs are characterized by some common features. The most important ones are:

1. ESs are multi-disciplinary domain where different teams collaborate (e.g. clients, hardware designers, software designers, system designers, analysts, etc...).
2. ES is implemented in only one Chip (SOC) with an extra high integration density
3. The extensive reuse of IPs (Intellectual Properties) due to time-to-market constraints.
4. The extensive use of co-simulations at different level of abstractions.

5. ESs design is very difficult and may lead to design iterations.
6. High performance expectations, real time, and power constraints.

Despite the prevalence of ESs, one remark the scarcity of abstract and visual programming models supporting features related to theses systems. In order to manage complexity, ESs designer have resorting to software and system engineering and borrow from them many well practiced concepts like abstraction. Although, abstraction masters complexity by reducing the number of processed objects, improves simulation speed, and enables designers to explore larger design space, it comes on a price of inaccurate estimation, more refinements automatization and extensive use of formal techniques for validation. In addition, and contrary to software part which is a logical concept, abstracting hardware is more difficult. Finding an optimal abstraction is by itself a key problem and requires a deep analysis to extract pertinent parameters that have great impact on system performances.

On the other hand, the emergence of the Unified Modelling Language (UML) as a standard for object-oriented modelling may improve the design quality of ESs and helps designers to take decisions at early stages of development. Due to extension mechanisms offered by UML, UML can be tuned by definition of a set of stereotypes and constraints. Beyond visual modelling and documentation capabilities, UML can also be used in performances analysis like time and power consumption, and code generation. We note that UML does not replace the existing state of the art and practice ESs methodologies. Instead, it builds atop of them and furnishes a good support for visual modelling, fast design space exploration, and automatic code generation. The remarkable maturity of UML-based tools for code generation (e.g. Rhapsody [22]) will push designers to concentrate on higher level of abstraction rather than coding. Another advantage of using UML is the possibility of exploiting UML-based tools for formal verification. Since UML does not dictate any particular development process to be used, it is on designers to define a design flow. We think that the Y-chart approach is the most appropriate. The Y-chart approach puts strong emphasis on the Platform-Based Design (PBD). According to our knowledge, the PBD is one of the best-validated industrial approaches for achieving high reuse in SoC design. SoC can be defined as a complex IC that integrates the major functional elements of a complete end-product into a single chip or chipset. In general, SoC design incorporates at least one programmable processor, On-Chip memory, and accelerating functional modules implemented in hardware. It also interfaces with peripheral devices, and/or the real world, and encompass both hardware and software components. The rest of the paper is organized as follow: the second section reviews quickly related work, section three and four present the Platform-Based Design, and the Y-Chart approach respectively. The remained sections (from section five till section thirteen) are dedicated to discuss our proposed approach in some detail before concluding.

2 RELATED WORK

With regard to the application of UML to the ESs and SOCs domains, the literature is very rich. However we can mention some pertinent works.

MARTE [17] is an UML profile that targets real time embedded software-dominated systems, it offers a facility for modelling, and analyzing real time applications.

UML-SOC profile [18] intends to describe SOC specific information using UML. It integrates concepts from SOCs and allows automatic code generation for hardware.

UML-SystemC profile [16] captures both the structural and the behavioural features of the SystemC language and allows high level modelling of SOCs with straightforward translation to SystemC code. TUT profile [14] provides an automated path from UML design entry to FPGA prototyping including the functional verification and the automated architecture exploration focusing on automatic profiling and performance values back annotation. Gaspard2 [5] is an UML2.0 profile, targeting intensive signal processing (ISP) domain. It defines stereotypes for application, hardware platform, and mapping. Gaspard2 is based on the ISP profile. The latter allows the expression of task and data parallelisms using Array-OL language. The two main problems with these profiles are the lack of abstraction and reuse: they are all focused on the task paradigm, and the lack of formal support for analysis, refinement and validation.

UML-SOC and UML-SystemC target hardware related aspects. Consequently, they show limitations toward software part. With Gaspard2, designers deal with complex index expressions for multi-dimensional data, and finally TUT profile lacks expressiveness for data parallelism, and pipeline modelling.

3 PLATFORM-BASED DESIGN (PBD)

Is basically a specialization of hardware/software co-design that specifically addresses the challenges of cost effective development in the to days economical environments.

PBD [12] is focused on the idea of orthogonalization of system function and implementation architecture, design space exploration, synthesis, and analysis. Beyond the reuse of individual hardware/software components, PBD reuses complex architectures of hardware and software components (Platforms) organized for a specific application.

PBD can decrease the overall time-to-market for the first products and expand the considerably early-delivering opportunities of derivative products. Platforms are means of standardization and facilitate design reuse, portability, and flexibility. Pre-characterization of architectural components with their implementation characteristics supports high-level estimation and helps to avoid design iterations. Architecture platforms may be abstract, such as software platforms, operating systems and libraries, or physical, such as specific combinations of processing elements, storage, and peripherals.

Several platform types have emerged nowadays as a result of the evolution of platform-based design. Table 1 summarizes four types of platforms.

Platform type	Example
Full-application platforms	- Nexperia: Philips Semiconductors - Open Multimedia Applications Platform (OMAP): TI
Processor-centric platforms	- Micropack: ARM
Communication-centric platforms	- uNetwork: Sonics - AMBA bus architecture: ARM
Fully programmable platforms	- Virtex-II Pro: Xilinx

Table 1: Different types of Platforms

4 THE Y-CHART APPROACH

The core methodology of PBD is the Y-chart approach to system design [13], which is illustrated in Figure 1. In this approach, an application model (derived from a specific application domain) describes the functional behaviour of an application in a timing and architecture independent manner. A platform defines architecture resources and captures their performance constraints. The implementation of the function with the architecture platform is established in a dedicated mapping step. Mapping is done by relating system behaviour and structure to appropriate architectural elements. Transformations are required to explore design alternatives and to optimize results. The best design, with respect to some cost function, is chosen for synthesis. The synthesized result may be analyzed and refined by subsequent flows. During the traversal of the design flows the level of abstraction of both, the function specification and the architectures, is steadily lowered, and an increasing number of implementation parameters is fixed. To perform quantitative performance analysis, application models are first mapped onto and then co-simulated with the architecture model under investigation, after which the performance of each application / architecture combination can be evaluated. The resulting performance numbers may inspire the designer to improve the architecture, restructure/adapt the application(s), or modify the mapping of the application(s). These designer actions are illustrated by the light bulbs in Figure 1. During these iterations, designers can try different platform customizations and functional optimizations. This solution brings many benefits since it is more direct (with less refinement); it increases the production volume, and decreases the overall cost and development time. However, it can lead to a lack of flexibility and optimization: this is because system-level simulation is simply too slow for comprehensively exploring the design space, which is at its largest during the early stages of design. Finally, due to lack of a set of well defined APIs (Application

Programming Interface) for interacting with SOCs, there is an abstraction level gap between the system specification and available SOC platforms.

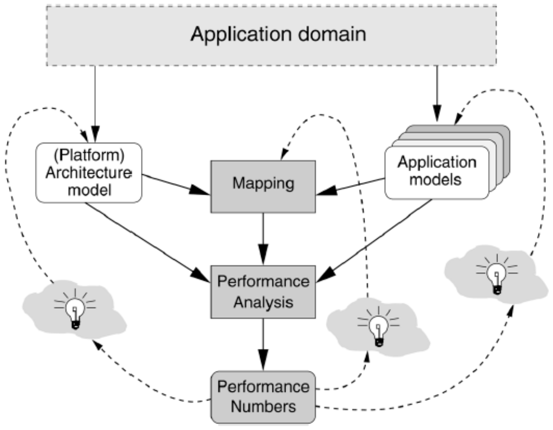


Figure 1: The Y-Chart approach [12]

5 OUR PROPOSED APPROACH

Our ultimate objective is to develop an UML-based environment for modelling, rapid design space exploration, and formal verification of ESs following the Y-chart approach. Using such environment, designers are able to validate a variety of ESs (control-dominated, data-dominated) at early stage of development. Our proposed methodology deals with two abstraction levels: the specification level where the application is described as a network of black box components, and behavioural level where components behaviour is refined. Indeed, our methodology starts at an early stage of development (the analysis stage) in which designer models the functional and Non-functional application requirements using UML sequence diagram annotated by temporal constraints. At this stage, the application is represented as collaboration between objects.

The internal behaviours of objects have not yet known only sequential interactions (possibly hierarchic interactions) with control information like conditions, loops and methods WCETs (Worst Case Execution Times). Hence, there is no explicit concurrency modelling and all messages are considered synchronous and executed sequentially. In other words, our initial model is a pure sequential object paradigm. We think that such a paradigm brings many benefits. First because, an object-oriented paradigm is preferable in term of abstraction and reuse. Secondly, we think that a sequential model facilitates the modelling task relieving the designer of the burden of concurrency modelling. Thirdly, starting from an existing sequential model (e.g. legacy C/C++ code) which is generally considered as the reference model, we can then extract many types of parallelism that exist in typical ESs, explore, and compare between different alternatives. In other word, the sequential model is strongly preferred from the system designer's perspective. The second step in our approach is the transformation of the analysis model (sequence diagram) to the design model comprising a set of communicating tasks with explicit task parallelism, data parallelism, pipelining, hierarchy and virtual communication channels.

The passage from the analysis model to the design model is done via a set of guidelines but we intent to automatize this passage in the near future. To support rapid design space exploration, the hardware architecture model should also be abstracted to match the abstraction level of the application. In our case, The SOC architecture is a set of abstract and generic components. Without abstracting the system, it is very difficult to perform a quick design space exploration for the intended system. It can be broadly said that the SOC architecture is composed of three types of architectural resources: computational resources such as CPUs, FPGA, IPs, communicational resources such as buses and memorization resources such as memories. Each architecture component is modelled as an abstraction of its fine grain model and they are generic components so that this whole architecture could potentially be used for modelling all types of SOCs. In order to estimate cost and exectution time, application is mapped to the SOC platform.

The mapping is done via a set of guidelines. These guidelines do not guarantee the optimal mapping, rather than, they help designers to find a good start solution. An important remark is that our approach does not address architecture exploration (as done by most of existing tools), rather than it targets application exploration. This tendency is justified by the fact that at higher level of abstraction, the application structure has the great impact on system performances. According to estimation results, designer can modify the application structure. When a convenient specification model is found, designer describes tasks internal behaviors using state charts or/and activity diagrams. We note that the internal behavior is still abstract (there is no implementation code). Only states for control dominated behavior, abstract operations (coarse grained) for data dominated behaviors or both in the case of mixed behaviors. Similarly, the architectural model is refined by adding interfaces and protocols communication between bus and other hardware components, and enrich the model by low level parameters. At this level of abstraction, more accurate time estimation is done. We introduce here the power consumption estimation. We think that the introduction of power consumption at specification level is useless because in general, the amount of power consumption is not proportional to cycles number: we can have two tasks with the same cycles number but differ largely in power consumption. for this reason we prefer to do it on the second level of abstraction where actions type is identified and blocking time is more predictable than specification level. According to estimation results, an optimization step is performed on both refined application and architecture models. A further step will be the formal verification of some properties such as deadlock and Bus congestion. The focus of this paper is on the specification level.

6 APPLICATION MODELING

Computation modeling

1. Leaf Behaviour

A Leaf Behavior (LB) represents the elementary schedulable computation of an application. LB inputs and outputs are defined as required and provided interfaces respectively. Each interface comprises a set of signals that carry data. The size of the data to be transmitted or received is expressed in term of tokens number. One of the novelties of our approach is data abstraction. So instead of specifying data width (e.g. 8 bits) or data type (e.g. float), at higher level of abstractions, it is preferable to express data as abstract tokens. An abstract token is the elementary datum communicated or processed by LBs. The actually width of data is known in further refinement steps. The advantage of such abstraction is the large opportunity to apply static formal analysis and fast simulations can be easily performed. We define a new stereotype named ***"behavior"*** with the following tagged values:

- The relative WCET (Worst Case Execution Time) expressed in term of cycles number when it is executed sequentially.
- Iterations number. The number of iterations can be either constant or input-dependant. In the latter case, we introduce the min and the max values.
- The min and the max of read access number to a shared data
- The min and the max of write access number to a shared data
- The dominated behavior (control, data, or a mixture). This information is very useful when mapping application on hardware platform.

Figure 2 shows an example of a leaf behavior called B.

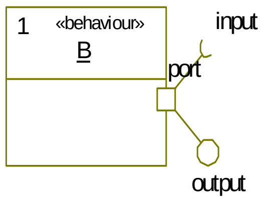


Figure 2 : Leaf behaviour

2. Behaviors Sequencing

In order to model behaviours executing in a sequence fashion, we introduce a new stereotype called ***"sequence"***. In figure 3, behaviours B1 and B2 are executed in sequence. For simplicity we do not specify interfaces associated with ports. Behaviours communicate via abstract channels.

3. Behaviours Pipelining

We introduce a new stereotype called ***"pipeline"***. This stereotype composes behaviours in sequence, with the output of one connected to the input of the next. The "pipeline" stereotype contains two tagged values the "PipeDepth" and the number of iterations. The latter specifies the number of the pipe stages. Each behaviour included in the "pipeline" stereotype is allocated to a different CPU. Pipeline execution implies the iterative execution of children.

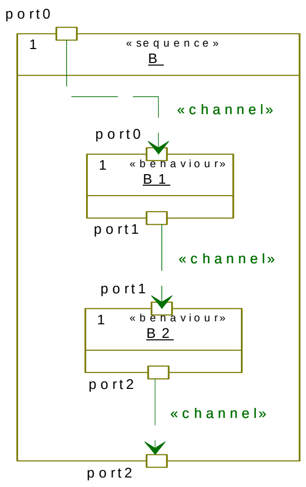


Figure 3 : Sequence behaviors

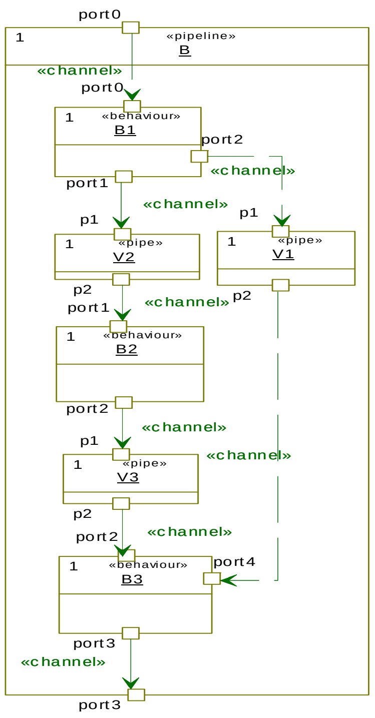


Figure 4 : Pipelined behaviors

In the example shown in Figure 4, the child behaviors B1, B2 and B3 form a three-stage pipeline of behaviors. When the pipeline is started, only B1 is executed. When B1 completes, the second iteration starts and B1 and B2 are executed in parallel. Finally, in the third and every following iteration, all three child behaviors are executed in parallel. The pipeline stereotype also supports communication buffering modelling between the pipeline stages. The example shown in Figure 13 includes three variables for communication between the pipeline stages (V1,V2 and V3). Each variable is stereotyped by a new stereotype ***"pipe"***. A variable stereotyped by "pipe" can be thought of as a variable with two storages. A write access to such a variable always writes to the first storage. A read access, on the other hand, always reads from the second storage. In addition, the contents of the first storage are shifted to the second storage whenever the pipeline starts a new iteration. In other words, the data produced by B1 will be accessable by B2 only in the second iteration, and by B3 only in the third iteration [10].

4. Data parallelism

To satisfy streaming applications needs, we introduce a new stereotype called ***“datapartition”***. This stereotype distributes data to a set of parallel streams, which are then joined together. Each data stream is executed by the same code and requires a memory buffer. Hence this stereotype duplicates behaviour into many synchronized behaviours. The “datapartition” stereotype contains three tagged values: *PartitionsNumber* to specify the number of data partitions, *PartitioningMecanism* to specify how incoming data are scattered over behaviours, and *CollectionMechanism* to specify the mechanism of data gathering. Each data partition is stereotyped by ***“partition”*** with one tagged value: *PartitionSize* in term of tokens number. A behaviour stereotyped by “datapartition” plays the role of a controller (master). It is responsible of creating, synchronization between the slave behaviours, data scattering and collecting. The controller executes concurrently with its slaves. Figure 5 shows a behaviour stereotyped by “datapartition”. In this example, we suppose that the data flow is distributed into two partitions. For each partition, a behaviour is created. Hence we have three leaf behaviours in parallel: B (master), B1, and B2 (slaves) and two partitions: data1, and data2.

5. Hierarchy

We introduce a new stereotype called ***“structure”***. This stereotype defines a hierarchical behaviour. We use this stereotype to manage complexity and to enable hierarchic descriptions. It has no tagged values. Figure 6 shows an example of a hierarchic behaviour called B. The latter contains four behaviours B1 (sequence), B2 (pipeline), B3 (datapartition), and B4 (leaf behaviour). B1, B2, B3, and B4 execute concurrently.

6. Mutually exclusive behaviours

In some cases, it is preferable to expose behaviours which are IF-dependant. In this case, we have to define a new stereotype called ***"exclusive"***. The latter is applied on UML constraints. In figure 7, behaviours B1 and B2 are mutually exclusive (only one behaviour is executed at a time). Exposing this kind of dependence leads to better estimations.

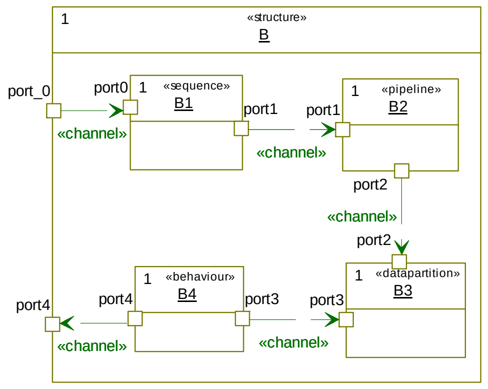


Figure 6: Hierarchic behavior

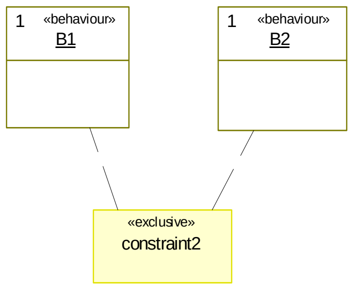


Figure 7: Mutually exclusive behaviors

Communication modeling

Two different communication models are supported, signal passing and shared memory.

1. Signal passing

In this case, behaviours communicate via abstract channels. Each channel is connected to two ports. We define a stereotype called ***"channel".*** This stereotype is applied on SysML flows. it has two tagged values : the *SampleMax* specifying the maximum size of the channel in term of data tokens number and the communication style which is can be blocking read-blocking write (BRBW), blocking read- non blocking write (BRNW), or non blocking read- non blocking write (NRNW).

2. Shared memory

In this case, tasks communicate via shared data. Here, two styles are also possible, synchronous and asynchronous. In synchronous mode, a lock is associated with each shared memory block and only one behaviour can access the memory at one specific time. Meanwhile, the asynchronous mode does not have a lock associated with the memory, and therefore concurrent accesses can happen. We define a stereotype called ***"shared data"*** with two tagged values: S*izeData* specifying the size of shared data in term of token numbers and *Communication Mode* which is can be synchronous or asynchronous.

7 ARCHITECTURE MODELING

In contrast to software, abstracting hardware is not a trivial task. For any abstraction, two requirements must be realized:

1. Keeping the model as realistic as possible, so good performance estimations can be done at even higher level of abstraction

2. The abstraction level of hardware model must match the abstraction level of the application model.

Hardware is generally parameterized by performance values like horologe frequency, instruction width, etc. One main question is how to represent the same performance parameters at a higher level of abstraction? Table 2 shows a possible correspondence between some known low level performance parameters and the abstract ones.

	Low level	**High level**
Horologe frequency	MHz	Speed Factor (SF)
Memory size	Kb	tokens number
Execution time	nano seconds	Cycles number
Rate transfer	bit/s	tokens/cycle

Table 2: Correspondance between low level and high level parameters

1. The CPU model

This model concerns both General Purpose Processors (GPP) and Application Specific Instruction Processors ASIP (e.g. DSP). It is a composite unit offering the services of a 'Player', which is capable of executing a behaviour (task). The CPU can execute one or many behaviours. In the latter case, it needs a scheduler. Each CPU is parameterized by five parameters that are cost, speed factor (SF), local data memory size, scheduling policy and context switching overhead. We denote a cycle to be the amount of time needed to execute an elementary instruction. The speed factor is a number showing the relative speed of the CPU. For a GPP, SF = 1. For a Computing Resource faster than GPP (e.g. FPGA, IP), SF < 1.

2. The IP model

This model concerns pre-characterized blocks. Each IP is parameterized by its cost, cycles number, and power units number.

3. The FPGA model

It is parameterized by cost, speed factor, memory size, and reconfiguration time in term of cycles number.

4. The BUS model

At this level of abstraction, we make assumption that bus communicate directly with other components without using interfaces. Each bus is parameterized by three parameters that are: cost, transfer rate (TRB) in term of number of tokens transferred per cycle, and bus type (shared or dedicated). In the case of a shared bus, we must specify the arbitration mechanism to solve the problem of concurrent transfers. If two hardware components need a fast link between them without arbitration, the designer may configure the bus as dedicated. We denote a token to be a single data value sent or received over the bus.

5. The Memory model

It is characterized by its transfer rate (TRR) in term of number of read tokens per cycle, and transfer rate (TRW) in term of number of written tokens per cycle.

Figure 8 shows an example of a hardware platform modelled as an UML structure diagram.

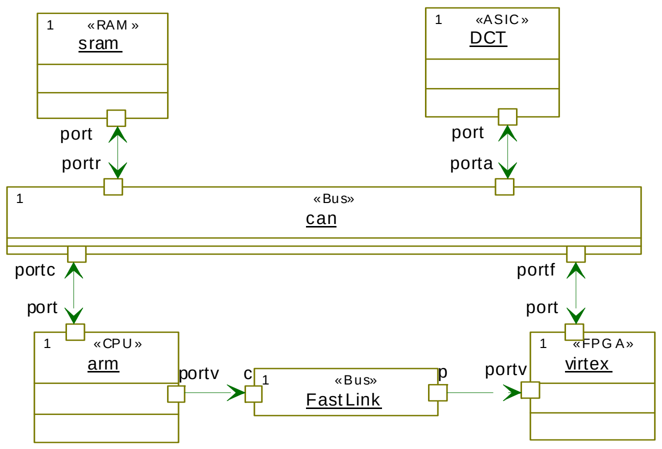


Figure 8: Abstract Platform model

8 MAPPING MODELING

Mapping consists in allocation and scheduling of application components to architecture components, so behaviours are mapped to computing resources (CPU, IP, FPGA), communication channels are mapped to buses, and data to memories. We define a new stereotype called ***"AllocatedTo"***. This stereotype is applied on the UML constraint and it has one stereotype specifying the hardware resource to which logical component will be allocated. We must note that the allocation concerns only leaf behaviours and not hierarchical ones (structure, pipelines, sequence, datapartition). Figure 9 shows an example of mapping using UML constraints. In this example Behaviour B is allocated to a physical resource called FPGA existing in the HWPlatform package.

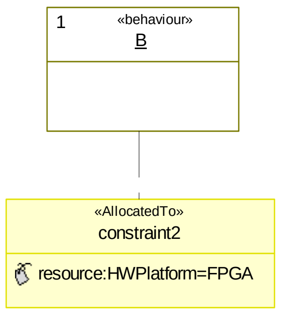


Figure 9: Mapping modeling

9 GUIDELINES FOR PARALLELISM EXTRACTION FROM SEQUENCE DIAGRAMS

In this section, we describe our technique to extract concurrent, pipelined, hierarchic behaviours, and mutually exclusive behaviours from an UML2.0 sequence diagram.

Our proposed flow starts by establishing a pure sequential model of the application. For this purpose we use the sequence diagram (SD). The latter is a good choice to model sequential (eventually hierarchic) interactions between objects. Furthermore, it exposes control and data dependencies, loops, and conditions explicitly. We enrich the SD with temporal constraints (WCETs). Figure 10 shows an example of a sequence diagram that supports hierarchy (due to interaction occurrence). There are five objects named MAIN,O1, O2,O3,and O4. These objects interact via message sending. Since we are dealing with a pure sequential application (classical object paradigm), all objects are considered passive and all messages are supposed synchronous.

Step 1: Identification of concurrent and pipelined behaviors

End-to-end scenarios may be concurrent behaviours. An end-to-end scenario is a sequence of dependant methods triggered by a method call or external event. In the example of figure 10 we can identify three concurrent behaviours named B1, B2, and B4. B1=(M1,M2,Get_attrib,M3,M4, Set_attrib), B2=(M21,M4,Set_attrib,,M31), B4=(SD1). Communication between these behaviours is achieved through shared memory. In the example 10, B1 and B2 access to attribute "attrib" of object O3 via methods Get_attrib (read), and Set_attrib (write). So we create a new data object called "attrib" stereotyped by "shared data" stereotype. Another form of parallelism that we can extract from SD is pipeline. In figure 10, methods M22,M23, and M24 of behavior B3 can be executed in a pipeline fashion. In the same example, we remark that the returned value of B2 (e) serves as input for B3 then B2 and B3 are in sequence. Since messages are synchronous, the caller must wait (blocked) for returned values. Not all methods receive or return data. In this case we will introduce two zero delay control events: the ***Request*** event and the ***Ret*** event. At this stage, we can also extract mutually exclusive behaviours. For instance B1 and B2 belong to different branches of the operator "alt" so they are mutually exclusive. Finally B4 is a hierarchic behaviour. Figure 11 shows the result of guidelines application where "main" is the controller and it executes concurrently with other behaviours. Using the sequence diagram, we can for each behaviour extract , a set of tagged values (see section 6.1). For example B1 WCET = 40 + 10 + (5 + 20) * 40 = 1050 cycles. Here loop<1,40> specifies the min and the max iterations number. B1 iterations number = 1, Max Read access number to shared data = 1 (Get_attrib), Max write access number to shared data = 40. The dominated behaviour information is introduced by the designer.

Step 2: Identification of DataPartition behaviors

This kind of parallelism can not be identified from sequence diagram. It requires knowledge on method internal data structures and the fashion the method manipulates these structures. Generally, methods with a large execution time are good candidates for splitting into less intensive concurrent behaviors.

10 MAPPING GUIDELINES

As mentioned above, these guidelines do not guarantee optimal mapping, rather than they enable designer to find a good start solution. These guidelines deal with two contradictory goals : communication overhead decreasing and behaviours execution time minimization.

The information constituting by WCET and the dominated behavior give us a good indication on the target source where behavior should be allocated.

- In general, computing-intensive behaviours are implemented in hardware (as IP or FPGA). But due to limited hardware resources, some resources should be shared. To overcome this problem, we map sequential and mutually exclusive behaviours to the same CR.
- If a behavior is control-dominated, it will not be mapped to FPGA, rather than it is preferable to map it to a GPP (General Purpose Processor).
- If a behavior is data-dominated with a very high computational load, then it will not be mapped to GPP, unless there is no available hardware resource.
- If the first objectif is the communication overhead decreasing then map behaviours with a high communication workload to the same computing resource, even they are concurrent.
- To minimize bus congestion, map channels with high traffic to fast links.

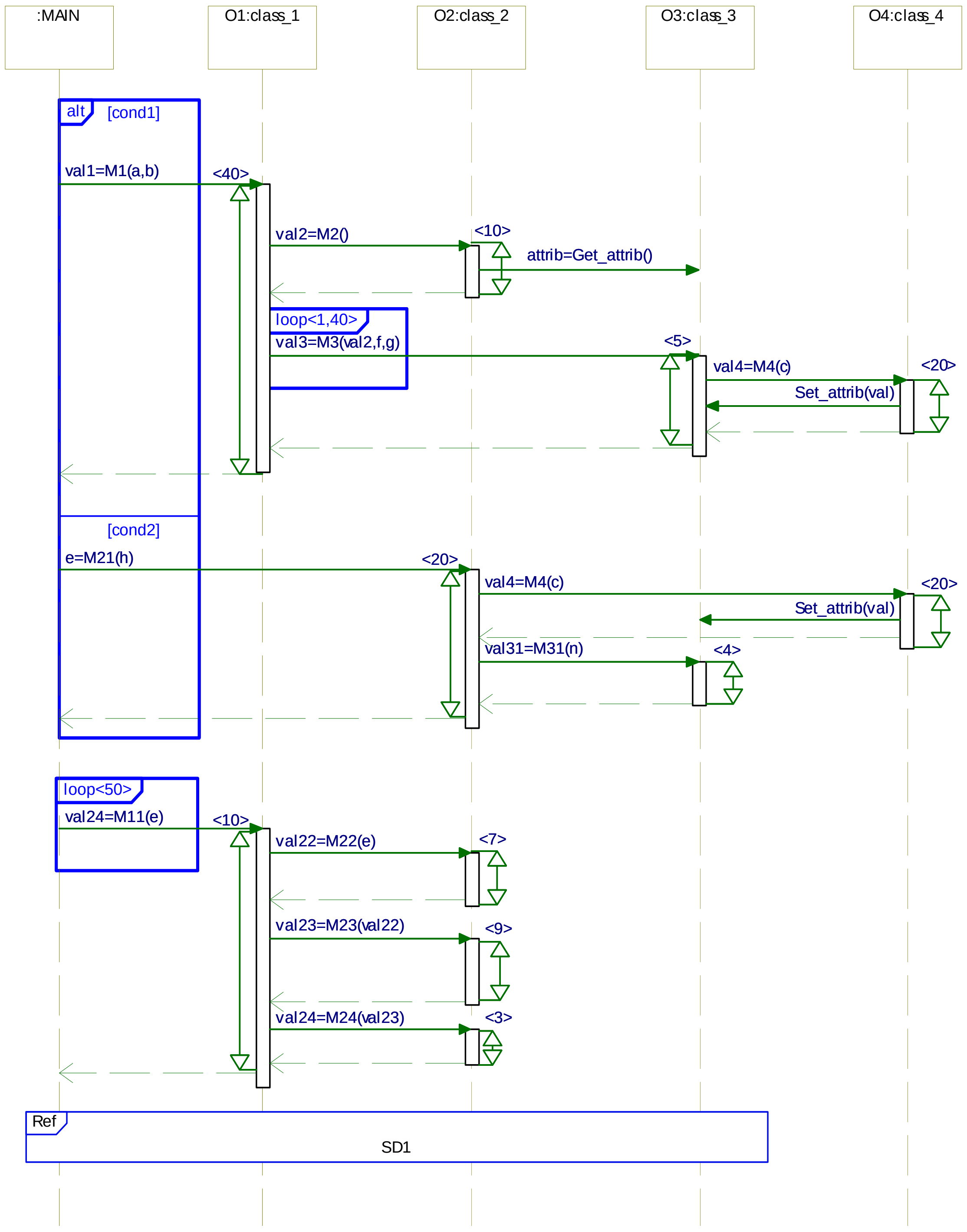


Figure 10: Hierarchic Sequence Diagram

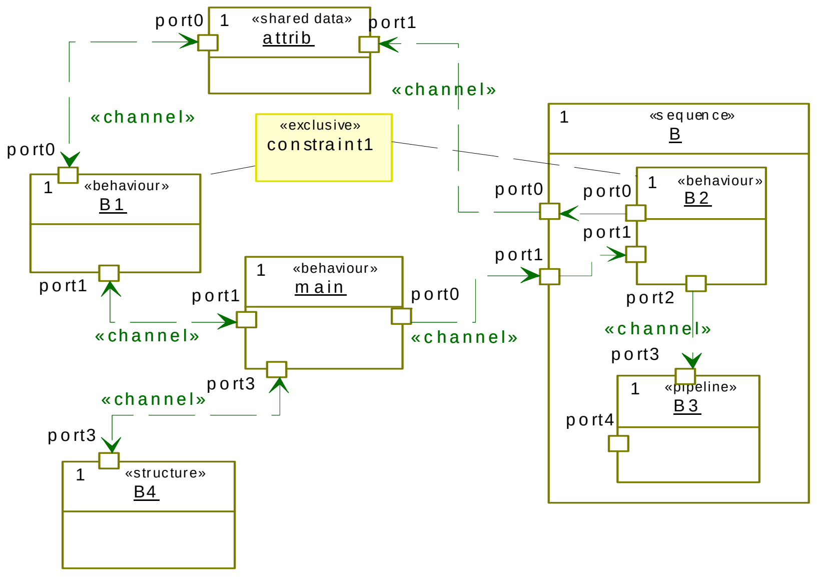


Figure 11: Hierarchic tasks model

11 OPTIMIZATION GUIDELINES

In our proposed approach, the optimization concerns application but not hardware architecture. At this level of abstraction, we think that application structure has a big influence on the quality of design. The impact of architecture model appears in subsequent refinement steps. For this reason, we decide to neglect architecture optimization. Indeed, there are many factors that affect the overall performances. The most important ones are:

1. Granularity of behaviours
2. Granularity at which data is communicated.
3. The amount of data processed by each task in the case of data partition
4. Management of shared data and communication scheme.
5. The data scattering and gathering mechanisms in the case of data partition.
6. The depth of tasks pipeline

These parameters have to be chosen carefully. The overhead caused by synchronization may counteract the benefits of parallelism, coarse granularity behaviours (that means small number of behaviours) are better than fine grained behaviours (big number) in term of communication overhead. However, behaviours may have to wait longer. Finally, depending on the application, data partitioning will give communication overhead for data dependencies between partitions. According to estimation results, designer may:

1. Merge behaviors with high communication workload into one behavior.

2. In figure 10, we remark that method M4 is duplicated into B1 and B2. Since M4 has a large execution time, it will be preferable if we put it separately.
3. For each computational bottleneck behavior, refine it following guidelines of section 9.
4. In order to minimize the effect of synchronization overhead due to data partitioning, it is preferable that each data partition processes larger data blocks [1].
5. If the size of a CPU local memory is adequate to stock read shared data, then copy this data to the local memory of this CPU, so minimization of bus congestion is acheived. The same thing with FPGA.
6. The FPGA reconfiguration overhead can be minimized if we put sequence behaviours in the same bitstream.

12 TIME AND COST ESTIMATIONS

In this section, we present our technique for time and cost estimations.

With regard to performance estimation, most of works target application profiling where time is computed on the basis of executed code. Since this technique requires the complete code, so it can not be applied on early stage of development. However, good estimation can be obtained prior to coding, even at the early stages of design, based on previous experience and similar existing designs. So our estimation model is based on analytic formula. Since we are dealing with higher level of abstraction the analytic analysis seems more appropriate. Of course our formula is inexact, but it serves as a good first attempt to model aspects related to time at higher level of abstraction. In our case we will interested in WCET (Worst Case Execution Time). In addition to time, we will also estimate the overall cost of the SOC platform.

Time estimation

We will be interested in WCET estimation which is expressed in term of cycles number. But before presenting our estimation technique, we make two assumptions :

1. All concurrent executions, transfers, readings/writtings in a shared resource (CPU, BUS, shared data) are processed using the round robin policy.
2. The communication mode on channels is supposed NRNW (FIFO with infinte size).

The first assumption is justified by the fact that at higher level of abstraction, the tasks blocking time can not be determined, so it is not possible to adopt an analytic method for penalty estimation caused by this blockage. To handle this issue, we will use Round Robin policy because we think that is more suitable for a WCET based estimation . In our case, all behaviors, transfers, readings/writtings to shared data have the same priority. The time quantum is equal to the minimum WCET time. On the other hand, the NRNW mode will facilitate the estimation task and the actual FIFO size will be known at the second level of abstraction. Let t is the WCET for behaviour B and SF is the Speed Factor of the Computing Resource CR.

If B is allocated to a CPU, then the estimated time Et for B is Et = t * SF.

If B is mapped to an IP (Pre-characterized), then Et = IP cycles number.

If B is mapped to an FPGA, we include the overhead due to reconfiguration :

Et = Et + Treconfig.

If there are other concurrent behaviours which are allocated to the same CPU (exept the mutually exclusive behaviours), then Et = Et + Tcpu where Tcpu is the overhead due to CPU round robin scheduling.

If B access shared data , then Et = Et + Tdata. Where Tdata is the overhead due to shared data access. Tdata = Tread + Twrite.

If B is the only behaviour that access to the shared data then

Tread = NBread * (Ttrans + Trmem) where Ttrans is the time of bus transfer, Trmem is the actually reading time and NBread is the number of read access. (we suppose that shared data access is done via a shared Bus).Ttrans = DataSize/TRB, and

Trmem = DataSize/TRR.

Twrite = NBwrite * (Ttrans + Twmem) where Twmem is the actually writing time, and NBwrite is the number of write access. Twmem = DataSize/TRW.

If there are many behaviours that access to the same shared data concurrently, then

Ttrans = Ttrans + Tbus, where Tbus is the overhead due to transfers round robin scheduling. Trmem = Trmem + Tr. Where Tr is the overhead due to memory readings round robin scheduling. Twmem = Twmem + Tw. Where Tw is the overhead due to memory writtings round robin scheduling.

Furthermore if B is executed iteratively, then TEt = Et*Iter. Where Iter is the maximum number of iterations and TEt is the execution total time of B.

Let t1, t2 the WCETs of behaviours B1, B2 respectively.

1. B1 and B2 are in sequence

- If B1 and B2 are mapped to the same CR, then T = TEt1 + TEt2 (we neglect the communication time between B1 and B2). If CR = FPGA, Treconfig is added one time.
- IF B1 and B2 are mapped to two distinct CRs and the two CRs are linked by a fastlink, then T = TEt1 + TEt2 + Tcom with Tcom = DataSize/TRB : is the communication time between B1 and B2 where DataSize is the size of the transferred data between B1 and B2. If the link between CRs is a shared bus then Tcom = Tcom + Tbus.

2. B1 and B2 are mutually exclusive

T = MAX (TEt1,TEt2)

3. B1 and B2 are concurrent

- If B1 and B2 are mapped to two distinct CRs then T= MAX (TEt1,TEt2).
- If B1 and B2 are mapped to the same CR, then we apply round robin scheduling and recompute T.

4. B1 and B2 are pipelined

Let n the number of iterations of the pipeline. B1 is mapped to CR1 with SF1 and B2 is mapped to CR2. Only behaviour B1 will be executed in the first iteration. In the second iteration, B1 and B2 will be executed concurrently. In the third and all following iterations, both behaviours are executed in parallel. After the n iteration, only B2 will be executed (in the n +1 iteration).

T = TEt1 + (n-1) * MAX(TEt1,TEt2) + TEt2.

5. B is a datapartition

Let n the number of data partitions and let S1,S2,…, Sn are their data partitions size respectively. A datapartition behaviour B will play the role of a controller. It will create n behaviours B1,B2,…Bn executing concurrently. It also splits and collects data. Assuming that the execution time is proportional to the amount of processed data, for each behaviour Bi, we can estimate its WCET by the formula ti = t*Si/S. where t is the WCET of B, and Si is the data partition size for behaviour Bi, and S is the sum of all data partitions sizes. When Bi is allocated to CRi then Eti = SFi*t*Si/S. Before slaves execution starts, the master should transmit data partitions to his masters. We can estimate transmission time by formula Tpart = MAX(S1/Ttrans, S2/Ttrans,….,Sn/Trans).

The controller itself executes concurrently with its slaves and takes time for splitting and data collection. T = Tpart + MAX (TEt1+Tcol1, TEt2+Tcol2…,TEtn+Tcoln). Where Tcoli is the overhead due to collection data. Tcoli = DataSizei/Ttrans. Where DataSizei is the size of Bi data outputs.

Cost estimation

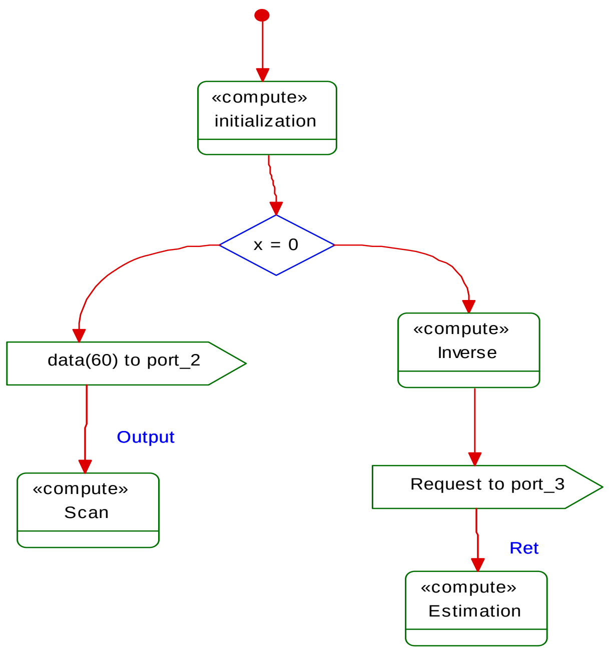
The overall cost of the hardware platform is given by the formula: C = CostCR + CostBus + CostMem where CostCR, CostBus and CostMem are the costs of resource computing (CPUs, IPs, FPGA), buses, and memories respectively.

13 SECOND LEVEL OF ABSTRACTION

The initial application model is a black box representation of a set of stereotyped communicating tasks with well defined interfaces and temporal constraints. Similarly, the architectural model is an abstraction of hardware components with a minimal set of parameters. These abstract models are well suitable for a quick design exploration. In order to trace a seamless way towards implementations, we must refine the initial models. So the application model is enriched by defining tasks internal behavior and the architecture model is enriched by introduction of power consumption units for each elementary instruction type. We can also add more detailed parameters such as execution cycles associated with basic operations, Instruction width (e.g. 8, 16 or 32 bits), cache-miss penalty for both instruction and data, CPU scalar factor (< 1 for sub-scalar, > 1 for super-scalar, = 1 for scalar CPU) , pipeline depth and branch miss-prediction penalty. The introduction of these parameters will lead to more accurate estimations.

Tasks behavior modeling

To model tasks internal behavior, we will use two types of UML2.0 diagrams: activity diagrams to model tasks which are data-dominated, statecharts to model tasks with control intensive computations, and activity/statechart diagrams to model tasks with mixed control/data computations. Since, we are dealing with higher level of abstraction, the data-dominated behavior is expressed in term of coarse grained actions (CGAs). Each CGA belongs to one of the three generic types: Computation Actions (CAs), Read Actions (RAs), or Write Actions (WAs). In addition, we have to model branches and loops. For this purpose, we will define a new stereotype called "*Compute"* . We use this stereotype to model CGAs computation. It contains one tagged value that specifies the number of elementary instructions inside a computation. For RAs and WAs, we will not define new stereotypes, rather than, we will use UML2.0 send and receive actions. Each RA or WA has two arguments: the source or the target port, and the read or the written data expressed in term of token numbers. Another novelty of our approach is the net separation between control and data activities. In the case of a mixture behavior, we will use an FSMD (Finite State Machine with Datapath) like model, where control part is modeled by a StateChart, and the data part is modeled by an activity diagram. Figures 12 shows an example of an activity diagram with coarse grained actions. In this example *"compute"* specifies a coarse grained action. *Data (60) to port_2* means write 60 tokens to a channel via port_2. Figure 13 shows an FSMD like model. It is composed of two objects: data to which, we attach an activity diagram, and control to which we attach a StateChart. The two objects communicate via an abstract channel.



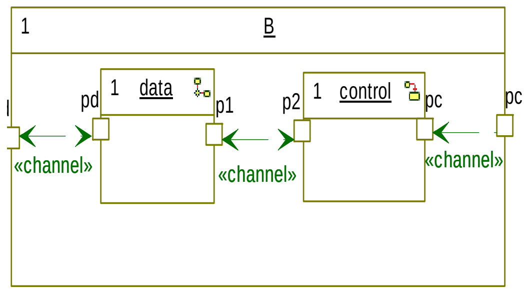


Figure 13 : A mixture contro/data behavior [6]

Figure 12 : Activity diagram with coarse grained actions

14 CONCLUSION

In this paper, we present a novel UML-based methodology, that given a sequential object oriented model, and a set of guidelines permits to generate a hierarchical task graph. It also enables designer to model his/her abstract architecture model on which application should be executed. After mapping, an abstract performance and cost estimation is performed. Indeed our approach deals with two abstraction levels: the specification level and the behavioral level. Although the proposed approach is based on a synchronous model (which is more suitable for streaming application), we can apply it on an asynchronous model (reactive systems). In the latter case messages calls are replaced by trigger events and the "main" object can be eliminated since first occurring events come form external environment (the main object is the external environment). However a lot of work stays in front. As a perspective we plan to:

1. Automation of the passage from the sequence diagram to the task model
2. Refinement of architectural model specifying the interfaces between bus and other hardware components.
3. Propose an estimation technique matching the second level of abstraction, and
4. Formal verification at the second level using the MAUDE system. Among properties, we should verify are the CPU deadlock and bus congestion.

REFERENCES

[1] I.Ahmed, Y. He, and M. L. Liou. Video compression with parallel processing. In Parallel Computing Journal 28, pp. 1039-1078, 2002.

[2] L. Appvrille, M. Waseem,R. Ameur Boulifa, S. Coudert, and R. Pacalet. Abstract application modeling for system design space exploration. Euromicro Conference on Digital System Design (DSD'06), Dubrovnik, Croatia, August 2006.

[3] L. Appvrille, M. Waseem,R. Ameur Boulifa, S. Coudert, and R. Pacalet. A UML-based Environment for System Design Space Exploration. 13th IEEE International Conference on Electronics, Circuits and Systems (ICECS'2006), Nice, France, December 2006.

[4] M. K. Bhatti, and L. Apvrille. Modeling and simulation of SoC hardware Architecture for Design Space Exploration. In SAME 2007 Forum. Session : Academic Posters. LaboSOC GET/ENST Paris, Sophia Antipolis, France, October 3 & 4, 2007.

[5] R. Ben Atitallah, P. Boulet, A. Cuccuru, J.L. Dekeyser, A. Honré, O. Labbani, S. Le Bleu, P. Marquet, E. Piel, J. Taillard, and H. Yu. INRIA. Rapport technique, Gaspard2 UML profile documentation.. N° 0342. September 2007.

[6] F.Boutekkouk, and M. Benmohammed. A Novel UML2.0-based approach for System On a Chip modeling and Co-design. In VLSI-SOC PhD Forum, Nice, France, 2006.

[7] F. Boutekkouk, and M. Benmohammed. Analyse dirigée par les scénarios de l'ordonnaçabilité d'une application embarquée temps réel. Dans CGE'05. Ecole militaire polytechnique, Bordj El Bahri, Alger, 16-17 Avril 2007.

[8] F. Boutekkouk, and M. Benmohammed. Modeling and simulation of embedded systems using UML2.0 and the Y-chart approach. In 4 th workshop on Object-oriented Modeling of Embedded Real Time Systems (OMER4), Paderborn, Germany, 30-31 october 2007.

[9] F. Boutekkouk et M. Benmohammed. Un outil d'aide à la conception des systèmes embarqués en suivant la méthode en Y. Dans MOAD'07, Méthodes et Outils d'Aide à la Décision. Béjaia, Algérie, les 18,19,20 Novembre 2007.

[10] R. Domer. System-level modeling and design with the SpecC language. Dissertation zur Erlangung des Grades eines. Doktors der Naturwissenschaften der Universitat Dortmund am Fachbereich Informatik. Dortmund, 2000.

[11] S.J. Mellor, J.R. Wolf, C. McCausland. Why Systems-on-Chip Needs More UML like a Hole in the Head. In Proceedings of the Design, Automation and Test in Europe (DATE'05) Volume 2.

[12] K. Keutzer, S. Malik, R. Newton, j. Rabaey, and A. Sangiovanni-Vincentelli. Sytem level design: orthgonalization of concerns and Platform-Based Design. In IEEE transactions on computer-aided design of circuits and systems, Vol. 19. , No. 12 ,December 2000.

[13] B. Kienhuis, Ed F. Deprettere, P.V. Wolf, and K. Vissers. A Methodology to design Programmable Embedded Systems. The Y-chart approach. In LNCS series vol. 2268, page 18-37 by Springer Verlag © 2001.

[14] P. Kukkala, J. Riihimaki, M. Hannikainen, T.D. Hamalainen, and K.Kronlof UML2.0 Profile for Embedded System Design. In Proceedings of the Design, Automation and Test in Europe Conference end Exhibition (DATE’05).

[15] T. Kangas, P. Kukkala, H. Orsila, E. Saminen, M. Hannikainen, and T.D. Hamalainen. UML-Based Multiprocessor SOC Design, in ACM transactions on Embedded computing Systems, vol. 5, No. 2, pp. 281-320, May 2006.

[16] E.Riccobene, P. Scandura, A. Rosti, and S. Bocchino. A SOC Design Methodology Involving a UML2.0 Profile for SystemC. In Proceedings of the Design, Automation and Test in Europe Conference end Exhibition (DATE’05).

[17] OMG. UML Profile for MARTE, Beta 1. *OMG Adopted Specification, ptc/07-08-04,* August 2007.

[18] OMG. UML Profile for System on a Chip (SOC). *OMG Available Specification, version 1.0.1 formal /06-08-01,* August 2006.

[19] OMG. Systems Modeling Language (SysML) Specification. *OMG document: ad/2006-03-08-01,* version 1. Draft, April 2006.

[20] T. Schattkowsky. UML2.0 Overview and Perspectives in SOC Design. In Proceedings of the Design, Automation and Test in Europe (DATE'05), Vol. 2.

[21] A. Viehl, O. Bringmann, and W. Rosentiel. Performance Analysis of Sequence Diagrams for SOC design. In proceeding, 2nd UML for SoC Design Workshop at 42nd Design Automation Conference (DAC), Anaheim, California, 2005.

[22] www.ilogix.com

About the authors

BOUTEKKOK Fateh (Fateh_Boutekkouk@yahoo.fr) received his BS degree in Computer science from the University of Constantine and his MS degree from the University of Jijel-Algeria- Now he is preparing a PhD degree at the University of Constantine. He is a lecturer at the University of Oum el Bouaghi since 2003. His research interests include Embedded systems, SOCs, and software engineering.

BENMOHAMMED Mohamed was born in Constantine, Algeria. He received his B.Sc. degree from the High School of Computer Science (C.E.R.I) Algiers, Algeria, in 1983, and the Ph.D degree in Computer Science from the University of Sidi Belabbes, Algeria, in 1997.

He is currently an assistant Professor at Constantine University. His current research interests are Parallel architectures and high level synthesis.

Sebastien Bilavarn received the B.S. and M.S. degrees from the University of Rennes in 1998, and the Ph.D. degree in electrical engineering from the University of South Brittany in 2002 (at formerly Lester, now Lab-STICC). Then he joined the Signal Processing Laboratories at the Swiss Federal Institute of Technology (EPFL) for a three year post-doc fellowship to conduct research with the System Technology Labs at Intel Corp., Santa Clara. Since september 2006 he is an assistant professor at Polytech'Nice-Sophia school of engineering, and LEAT Laboratory, University of Nice-Sophia Antipolis - CNRS. His research interests are in system modeling, design, exploration and optimisation from high level specifications with investigations applied to heterogeneous and reconfigurable architectures, multiprocessor systems, ESL design, UML, power management, more especially in the field of mobile applications.

Michel Auguin has currently a position of Research Director at CNRS (Centre National de la Recherche Scientifique) in the group " System level modelization and design of communicating objects" of the LEAT laboratory from University of Nice Sophia Antipolis in France and CNRS. In this group he is working on SoC system level design methodologies. Previously, he has been involved since 1980 and for nearly 15 years in the area of parallel processing and architecture. Since 1995 he has been a staff member of several national research programs focusing on parallel architecture and SoC. In the field of SoC design methodologies he currently participates to regional, national and European collaborative projects.