A co-design approach for embedded system modeling and code generation with UML and MARTE

Jorgiano Vidal*, Florent de Lamotte*, Guy Gogniat*, Philippe Soulard†, Jean-Philippe Diguet*
* European University of Brittany - UBS – CNRS, UMR 3192, Lab-STICC
Centre de Recherche - BP 92116 – F-56321 Lorient Cedex - FRANCE
† SODIUS – 6 rue de Cornouaille – F-44300 NANTES - FRANCE

***Abstract*—In this paper we propose a UML/MDA approach, called MoPCoM methodology, to design high quality real-time embedded systems. We have defined a set of rules to build UML models for embedded systems, from which VHDL code is automatically generated by means of MDA techniques. We use the MARTE profile as an UML extension to describe real-time properties and perform platform modeling.**

The MoPCoM methodology defines three abstraction levels: abstract, execution and detailed modeling levels (AML, EML and DML, respectively). We detail the lowest MoPCoM level, DML, design rules in order to perform automatically VHDL code generation. A viterbi coder has been used as a first case study.

I. Introduction

UML [1] has been used for application modeling since its first definition. The wide range of supporting tools, the extensions mechanism and improvements from its later versions – notably 2.0 – has stimulated its use in hardware and hybrid system modeling.

A set of UML properties was identified concerning embedded systems modeling [2]. These properties have encouraged UML adoption in embedded systems design, but there were some lacks, such as a platform modeling. To address these issues, MARTE profile [3] was defined and is in adoption by OMG.

The MoPCoM co-design methodology [4] defines three levels of abstractions in real-time embedded systems models: Abstract Modeling Level (AML), Execution Modeling Level (EML) and Detailed Modeling Level (DML). A complete system (application and platform) is defined within each MoPCoM level. In this paper, we show how to design embedded systems with MARTE and UML. Our approach is focused on performing automatically code generation from the model.

The rest of this paper is organized as follows: section II recalls related works. Section III introduces MoPCoM methodology and section IV presents the most detailed MoPCoM level, DML. In section V, we show the global rules concerning code generation. Section VI shows a modeling example . Finally, in section VII, the conclusions until now are proposed.

II. Related Work

The use of model based approaches for co-design has been discussed in [5], which points out some advantages: cost decrease, silicon complexity handling, productivity increase, etc. UML/MDA has been adopted in co-design methods [6], [7], [8], [9] in the last years with success. The extensions mechanisms introduced in UML since its version 1.3 has stimulated its use in embedded systems modeling, as such kind of systems need specific models.

In [6], the authors define an UML profile to model SystemC elements. SystemC skeleton code is generated from an UML model. Also, in [10], a SystemC profile is defined and behaviors can be specified by means of UML state machines, where a TLM (*Transaction-Level Modeling*) SystemC [11] code is generated.

In [7] an extension is done defining a new profile, the TUT profile, to embedded systems designs. It defines a set of stereotypes to model application tasks and platform. The platform uses a library to allow performance analysis. The design flow only allows software code generation in C. It has an architecture space exploration tool that back annotates the UML model. A complete example is done in [12].

In [8], the authors define a UP-based (Unified Process) process that uses a SystemC profile to model embedded systems and generate SystemC code.

In [9], the authors use UML to VHDL code generation. They use the same model to generate HW and SW parts. Partition is done manually, separately from the model.

All design methodologies shown in literature prove UML to be well suited to embedded systems design. As UML metamodel lacks platform design and real-time characteristics, extensions had to be made in order to capture these properties. Each methodology has made its own extensions to adapt UML to their needs. Extensions used by these methodologies usually limit code generation to SystemC.

Compared to existing efforts, our approach uses standardized UML and extensions, which allows the use of generic UML tools and model portability. Moreover, our model defines behavior and platform separately, which allows evolving partition with the model. Moreover, as it is not directly connected to any implementation language, we are able to target any language (SystemC, VHDL, etc).

978-3-9810801-5-5/DATE09 © 2009 EDAA

III. MoPCoM methodology

Defined in [4], the MoPCoM approach is a co-design methodology based on OMG standards. MDA [13] techniques are used to perform code generation. The highest system model level is done with an Harmony process [14] from Telelogic. It is based on the SysML [15] profile and is enhanced with some MARTE elements. MoPCoM methodology defines three design levels:

- AML *Abstract Modeling Level* is the first design level, where the goal is to model system behavior.
- EML *Execution Modeling Level* is the intermediate level, where performance analysis can be done, due to a final topology model of the system.
- DML *Detailed Modeling Level* is the last modeling level, which allow code generation to be done. Other levels allows code generation to simulate the system, whereas DML allows implementation code to be automaticaly generated.

MoPCoM methodlogy defines three models to be specified at each level: application, platform and allocated models, where:

- Application — Contains the functional specification of the system where connected objects communicate through messages and signals. The MARTE NFP sub profile is used to express real-time constraints.
- Platform — Is composed of a set of components, without behavior, connected together. It models the system topology.
- Allocation — Connects behavior (application) with platform. It consists of a set of UML dependencies with MARTE ≪**allocate**≫ stereotype.

IV. Embedded system modeling

In this section, we detail DML, the lowest MoPCoM level. First, in section IV-A, a DML overview is proposed, then in section IV-B, we detail the application model at this level, and, in section IV-C, we show how to model a DML platform. Last, in section IV-D, the allocation model is presented.

A. DML overview

The DML defines the platform at a clock cycle tick accurate definition, where the final target RTL model can be generated. At that level, hardware specification is finished. Thus, hardware components can be generated or existing ones can be used (IP blocks). Figure 1 shows the main elements at that level and the code generation possibilities.

All elements are constrained by UML 2.1 metamodel and we have three defined models: Functional, Platform and Allocation.

First, the functional architecture model is an UML model consisting of a set of interfaces, classes and objects. Each class owns a behavior, which is defined by means of a state machine and/or an action language.

The platform model consists of a set of components connected through ports. Each one is stereotyped with MARTE HRM profile elements in order to detail its characteristics. They can be connected directly or using an hardware bus, where a bus is also a stereotyped component. UML ports are stereotyped as ≪*HwEndPoint*≫.

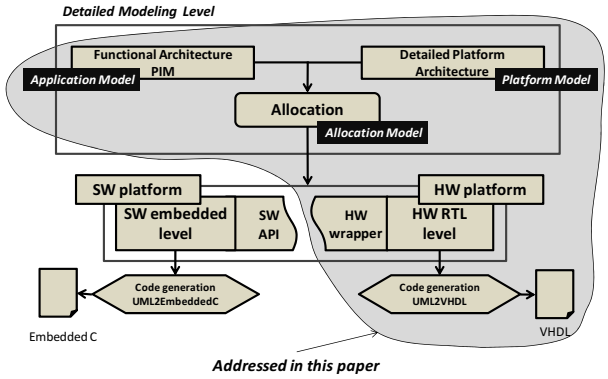

Fig. 1. Detailed Modeling Level

The allocation involves defining where functional objects (behavior) exist in platform ones. In our methodology only objects (classes intances) are possible to be allocated.

It also important to remark that the functional model contains the system behavior, which will be realized by a hardware or software part. The concrete implementation element of some behavior is done by the allocation, where functional objects allocated to hardware elements – PLD or ASIC – are the hardware parts of the system and the functional objects allocated to processor are the software elements. The SW/HW partition is defined in allocation in DML.

B. Application model

Application defines behavior and functional architecture. The model used to define the application is built from the following modeling elements: interfaces, classes, ports and instances for structural modeling; state machine and action language for behavioral modeling. Figure 2 shows the diagrams used. Application modeling concerns defining system services and their behavior.

1) Class diagram: This diagram is used to defined interfaces, classes and associations. Interfaces represent a set of public operations –services–. Classes may **realize** and **use** interfaces. The first one defines how such operations are implemented and the second calls the operations. Moreover classes can define private operations and attributes – variables – to implement the service. Within classes we can define the behavior unit of the application specified with state machines and action language. Public attributes are forbidden and all public operations must be defined in the interfaces.

2) Composite structure diagram: Composite structure diagrams are used to define classes internal structures and communication ports, where a port offers/requires a service. This diagram allows us to improve our design element - class - with ports, used as a communication point with other elements, and design its internal structure. The service offered/required

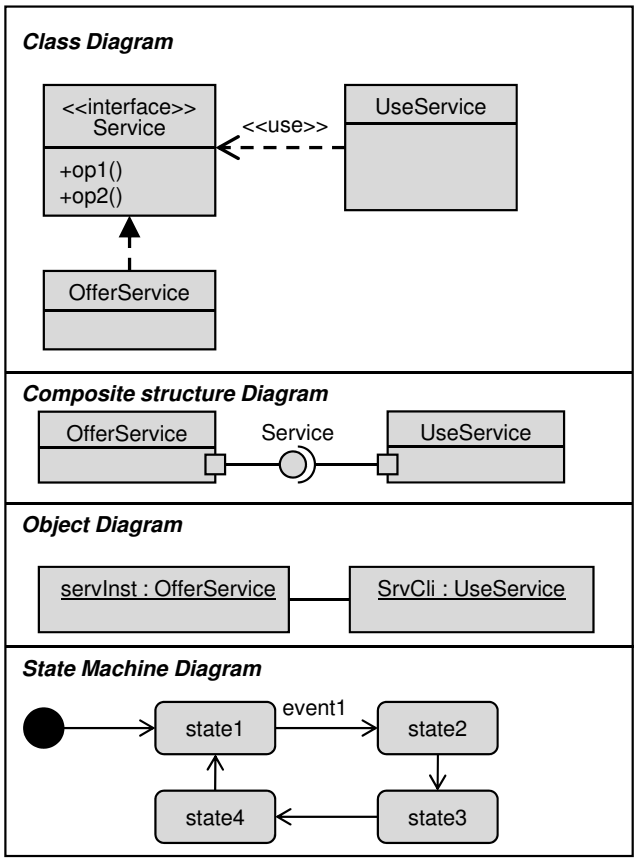

Fig. 2. Application modeling diagrams

by a port is specified attaching an interface to the port. All communication must be done by ports, which means that only operations defined in interfaces are visible externally. An explicit communication point must be done in order to allow clear code generation for hardware components, as there is a wide range of protocol types.

3) Object diagram: object diagrams define application instances and their connections. The behavior defined within classes are performed by instances, which are used to model the execution scenario. We also define here the object connections, which is a class association instance.

4) State machine: A state machine can be used to express a class behavior (every state machine is attached to a class), and each state behavior is defined using the action language. State transitions may be triggered by events and/or guarded by Boolean expressions. State machines are used to model high level behavior.

5) Action language: The used action language is a subset of C++ which allows synthesizable VHDL generation and can be extended to support syntax constructs offered by HLS – *High Level Synthesis* – tools, like GAUT [16] or CatapulC, from Mentor Graphics [17].

The action language subset is defined with usual restrictions according RTL ode generation. Only combinational code is allowed: assigments, if and switch statements, bounded loop structures. A variable cannot be assigned twice neither can be used after an assigment. Pointers are not allowed, so attributes must be accessed directly and all access to other objects must be done by ports.

A parser is used to generate VHDL, where a low-level C++-to-VHDL translation is done. The action language is used to define operations in classes and state behaviors in state machines.

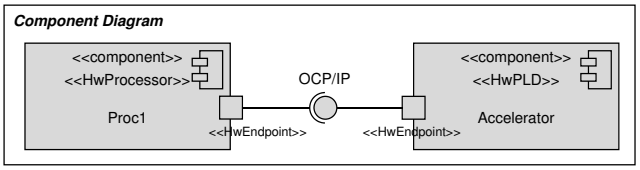

Fig. 3. Platform model

C. Platform model

The platform defines the structural hardware components. A component diagram is used to model it. MARTE HRM – *Hardware Resource Modeling* – sub-profile is used to define which kind of elements each object represents, such as ASIC, PLD, Clock, etc. MARTE SRM -*Software Resource Modeling*– sub-profile is used to model operating system properties, like task and virtual memory. MARTE SRM elements are not addressed here.

A platform is defined as a set of components connected through ports. For each port a stereotype, which defines a communication protocol, is attached. A library is associated to each protocol stereotype, which is used in code generation. Figure 3 shows the elements in a platform model.

A component with a ≪HwClock≫ must be present in the platform. A clock is used to allow performance analysis and synchronous component code generation.

1) Component diagram: The component diagram contains the platform resources. At least two stereotypes must be present for each component: ≪HwLogical≫ and ≪HwPhysical≫[1]. Both must be present to characterize DML, although ≪HwPhysical≫ is not used for code generation. Components are used to model the platform as they are reusable unit that offer services, which abstrat their behavior. Each component can also be identified by an IP number, which allows IP reuse.

Components are connected together by UML ports, where the ports contain the stereotype ≪HwEndPoint≫. An endpoint is an interaction point to communicate with the component.

2) Protocol definition: Inter component communication is done by some communication protocols. To facilitate specification, a protocol stereotype, ≪protocol≫, is defined. The concepts used to model a protocol are the same as in OCP/IP [18]. There must exist a protocol definition for each port, and two communicating ports must use the same protocol definition.

[1]Actually, HwLogical and HwPhysical are abstract types and we must use some of their subtypes. See MARTE specification [3]

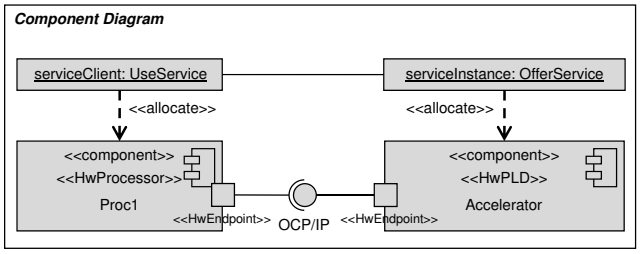

Fig. 4. Allocation model

OCP-IP protocol specification was chosen because it is parameterizable, allowing wrappers to be made for most existing protocols. Each paramesizable attribute defined by OCP/IP is a tag within the protocol stereotype.

D. Allocation model

The allocation model is built from the application and platform models, linking each functionality with a platform component. To allocate functionality to platform components, we use the MARTE alloc sub-profile. The allocation is done using UML dependencies with MARTE ≪allocate≫ stereotype, where classes instances are allocated to platform components, as shown in Figure 4. Application objects are virtual instances with a behavior. Such behavior will be executed in a platform component. ≪allocate≫ stereotype maps each instance to a component. MARTE allocation is simple and flexible, thus it is well adapted for co-design modeling issues.

It is important to remark that an object whose behavior is defined with a state machine must be allocated to a component connected to a clock.

E. Modeling issues

Our methodology proposes a complete model from which code can be generated automaticaly. Such requirements limit UML elements, diagrams, MARTE and action language as described in this section. Next section explains our approach to automaticaly generate VHDL code.

V. Code generation

Code generation is made from the allocated model, which holds the application, the platform and the allocation. In the paper, we address VHDL code generation for hardware parts of the system, and we do not consider IPs integration neither IP reuse. The goal is to produce synthesizable VHDL code for each new IP. For now, the real-time properties – MARTE Non-Functional Properties sub-profile – are not taken into account for the code generation algorithm. Moreover, such properties can ben used as constraints by an HLS tool. To achieve our goal, we decompose the code generation in 3 steps: structure, behavior and communication code generation.

A. Structure code generation

Structures are derived from platform components and represent the system blocks. Each platform component is translated into a VHDL entity. The entities own a set of VHDL ports. A port definition in a platform defines the set of VHDL ports to be implemented. For instance, a protocol definition associated to a platform port is translated into a set of data and control signals connected to the entity, and the VHDL ports are derived from a protocol library. A reset is added to any entity connected to a clock. Algorithm 1 shows how VHDL entities are generated.

Algorithm 1: GenerateStructure

```
Input: System model s
Output: VHDL model v
platformModel ← s.getPlatformModel();
foreach Component c ∈ platformModel do
    entity = CreateEntity(c.name());
    foreach Port p ∈ c do
        entity.createVHDLPorts(p);
    end
    v.addEntity(entity);
end
```

B. Component behavior generation

Behavior is defined in the application model and is translated as VHDL processes or VHDL function. Three main elements are used to build VHDL behavior code: state machines, methods and attributes. A component behavior does not use entities ports. It just handles internal variables and signals, which are defined by object attributes and method parameters. Algorithm 2 describes behavioral code generation. The overall rules for the elements are:

1) State machine: State machines are translated into VHDL processes. The generated entity must be connected to a clock, and the state transition is done by clock ticks. For each state, the VHDL code is translated from action language code specified in the state machine.

2) Methods: Methods are translated into VHDL functions in a package, available for all instances of the class owning the method. If a method needs to access attributes, each one is coded like *inout* function parameter. Local variable is coded as VHDL process variable.

3) Attributes: As we do not accept public attributes in the object model, each attribute will be a shared signal (register) into a VHDL module. Such signals can be used as function parameters and/or accessed from a state machine derived process.

C. Communication

Communication is addressed separately from structure and behavior, as ports are part of static domain (components), but may also derive from the behavioral domain (function parameters for instance) depending on the communication channels. Communication structure is a key point addressed in DML. In order to perform RTL code generation, we use a set of protocol concepts and generate the signals and protocol state machine from the platform model. The protocol behavior is translated onto a VHDL process that links the

Algorithm 2: GenerateBehavior

Input: System model s
Input: VHDL model v
Output: VHDL model v
objs = s.getObjects();
foreach *Object* $o \in objs$ **do**
  comp = o.allocatedTo();
  e = v.getEntity(comp.name);
  arch = createArchitecture(e);
  foreach *Attribute* $a \in o.getAttributes$ **do**
    arch.createVariable(a);
  end
  foreach *operation* $op \in o.getOperations()$ **do**
    v.createFunction(op);
  end
  if *o.hasStateMachine()* **then**
    stm = CreateStateMachine(o.getStateMachine());
    arch.addProcess(stm);
  end
  v.addArchitecture(arch);
end

entity ports with the behavior processes. The protocol proces controls the behavior process by means of shared variabl and signals. Point-to-point communications signals are derive from methods/events parameters. Algorithm 3 performs VHDl code generation from a given model.

Algorithm 3: GenerateCommunication

Input: System model s
Input: VHDL model v
Output: VHDL model v
foreach *Entity* $e \in v$ **do**
  objects = $s.getAllocatedObjectsIn(e.name)$
  foreach *Object* $o \in objects$ **do**
    ports = o.allocatedTo().getPorts();
    comm = CreateCommProcess(o,e,ports);
    e.getArch().addProcess(comm);
  end
end

D. Tooling

The selected UML modeller is Rhapsody [19], which is one of the most convenient software tools for code generation dedicated to embedded and electronic applications, although affected by some limitations regarding compliance with UML 2.1 standard. The choice was defined by MoPCoM project, due to its high integration with MDworkbench [20], the code generation tool used by MoPCoM project. MDworkbench is a transformation tool widely used in the industry, developed by Sodius, a MoPCoM partner.

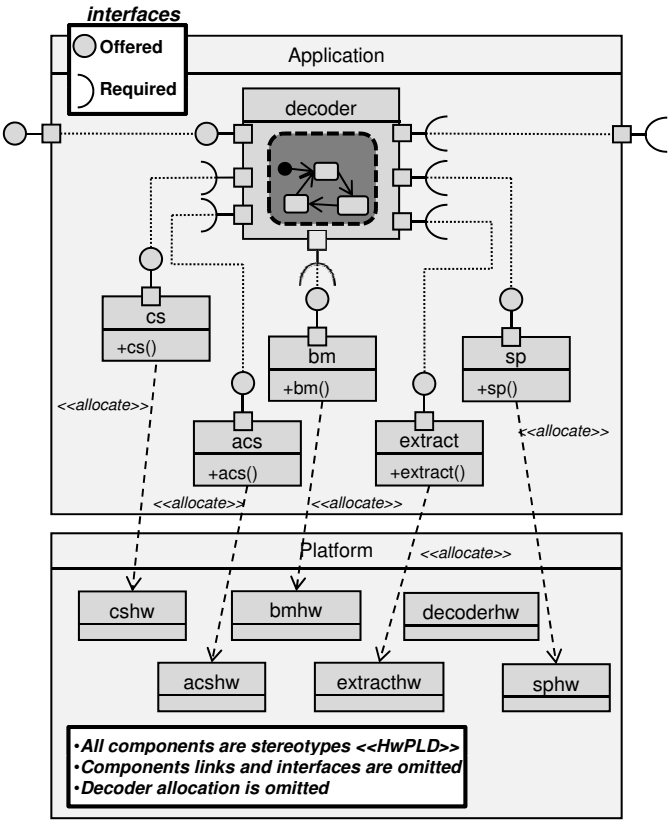

Fig. 5. Viterbi application model

VI. A CASE STUDY EXAMPLE

To test our approach, we designed a Viterbi decoder to validate the rules. An external component sends events with an integer parameter(the value to be decoded). After decoding, a new event, also with an integer parameter is sent from the decoder to another external component.

As Rhapsody is not fully UML 2.1 compliant, some adaptations had to be done. The Rhapsody **Object Model Diagram** substitute four UML diagrams: class, object, composite structure and component diagrams, incorporating their characteristics. Object modeling capability is increased with component modeling properties, which allows Rhapsody objects to be modeled as UML2.1 components.

Figure 5 shows the Viterbi model [2]. The decoder interface defines one operation, decode, and uses a set of helper operations (acs, extract, bm, etc...), each one defined in one interface and realized by a class with its name. A state machine defines the decoder behavior, and all other operations are combinational ones and the action language is suficient to define them.

The platform follows the same functional decomposition, where it is decomposed also in six components. The application objects are allocated to the components, an one-to-one allocation in the example. The decoder component has a port

[2]Application specification is not shown due to space limitations

connected to a **HwClock**. All platform objects communication is point-to-point and the system external connections are specified with a protocol definition, which is translated to OCP/IP code (not in the figure).

Code generation facility builds a VHDL entity for each platform component with a ≪HwLogical≫ stereotype. Action language elements are translated into VHDL ones. As defined in the algorithm, for each platform component a VHDL entity is defined. The component with the state machine contains an architecture with two processes: one for the state machine and another to perform communication. All other components contains just one process to perform communications and the operations are implemented as VHDL functions. The communication process calls the functions and sends the result by the communiation channel.

`viterbihw` is the top level viterbi entity, with 185 lines of VHDL code. It is composed of 6 entities: `decoderhw` (789 lines), `acshw` (65 lines), `bmhw` (113 lines), `extracthw` (44 lines), `cshw` (71 lines) and `sphw` (28 lines). Generated code was synthesized with Synplify Pro [21] targeting a Xilinx Virtex 2 Pro FGPA [22]. Total LUTs was $1,106$ (3% of a XC2VP30, package FF896).

VII. Conclusions

A co-design methodology, as specified in [4], has shown UML suitable for HW/SW modeling. A well-define design methodology helps MDA adoption in co-design, allowing code generation facility. Our experiment has shown a VHDL code generation possible from RTL level UML system models.

Our approach defines three models: functional, platform and allocation. In the functional model the designer specifies the behavior of the systems by means of an object oriented model. The platform is a set of hardware components where behavior will resides. The allocation maps the behavior onto the platform components, where the HW/SW partition is done.

The code generation tool extract the new hardware components to be generated and writes VHDL code for each one. In order to generate code we define three different parts to be generated: structure, behavior and communication. The behavior is quite simple and takes the state machine and action language from the functional model. The structure is built from the platform definition, where UML components are translated onto VHDL entities. VHDL ports are defined from protocol definition and methods parameters derivation. The communication is a key point in code generation and depends both from functional and platform models. The main difficulties in this approach concerns communication issues between components.

MARTE profile was used in order to allow platform elements to be present in our model. As UML and MARTE are OMG standards, our methodology can be used in any UML2 compatible tool. Moreover, we build a complete embedded system model – application and platform. As we use MARTE alloc sub profile, SW/HW partition is done entirely within the model by means of allocation. Our approach considers the entire system to be modeled. Design rules are well defined and we are able to generate behavioral VHDL code.

Our approach aims to generate input for usual HLS tools, that can performs architecture optimization by means of behavioral synthesis. In such case our code generation tool create wrappers that connect such output IP blocks with the system.

Acknowledgment

The UML/MDA approach presented above is experimented in the RNTL06 research program MOPCOM SoC/SoPC supported by the French National Research Agency (ANR – contract 2006 TLOG 022 01), the "cluster of clusters", "Media and Networks" and the Brittany and Pays de la Loire regions.

References

[1] OMG, "The Unified Modeling Language (UML)," http://www.omg.org/uml. [Online]. Available: http://www.omg.org/uml

[2] G. Martin, "Uml for embedded systems specification and design: motivation and overview," *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pp. 773–775, 2002.

[3] OMG, "Uml profile for marte, beta 1," Object Management Group, Tech. Rep. ptc/07-08-04, 2007.

[4] A. Koudri, D. Vojtsiek, P. Soulard, C. Moy, J. Champeau, J. Vidal, and J.-c. Le lann, "Using marte in the mopcom soc/sopc methodology," in *workshop MARTE*, 03 2008.

[5] J. Dekeyser, P. Boulet, P. Marquet, and S. Meftali, "Model driven engineering for soc co-design," *IEEE-NEWCAS Conference, 2005. The 3rd International*, pp. 21–25, 2005.

[6] T. Wang, X.-G. Zhou, B. Zhou, L. Liang, and C.-L. Peng, "A MDA based SoC Modeling Apporach using UML and SystemC," in *Proceedings of the sixth IEEE International Conference on Computer and Information Technology (CIT'06)*. IEEE Computer Society, september 2006, pp. 245–245.

[7] P. Kukkala, J. Riihimaki, M. Hannikainen, T. D. Hamalainen, and K. Kronlof, "Uml 2.0 profile for embedded system design," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 710–715.

[8] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio, "Designing a Unified Process for Embedded Systems," in *Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*. IEEE Computer Science, mars 2007, pp. 77–90.

[9] M. Rieder, R. Steiner, C. Berthouzoz, F. Corthay, and T. Sterren, "Synthesized uml, a practical approach to map uml to vhdl," in *RISE*, 2005, pp. 203–217.

[10] E. Riccobene, A. Rosti, and P. Scandura, "Improving SoC Design Flow by means of MDA and UML Profiles," in *3rd Workshop in Software Model Engineering (WiSME 2004)*, october 2004.

[11] Open SystemC Initiative, "OSCI TLM2 User Manual," 12 2007.

[12] P. Kukkala, M. Setälä, T. Arpinen, E. Salminen, M. Hännikäinen, and T. D. Hämäläinen, "Implementing a wlan video terminal using uml and fully automated design flow," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 20–20, 2007.

[13] OMG, "Model Driven Architecture (MDA)," http://www.omg.org/mda. [Online]. Available: http://www.omg.org/mda

[14] B. P. Douglass, "The harmony process-the development spiral," 2005.

[15] OMG, "Sysml profile," Object Management Group, Tech. Rep. ptc/06-04-03, 2006.

[16] "GAUT – High-Level Synthesis tool from C to RTL," http://web.univ-ubs.fr/lester/www-gaut/.

[17] "Mentor graphics," http://www.mentor.com.

[18] Open Core Protocol International Partnership, "OCP/IP Specification," http://www.ocpip.org.

[19] "Rhapsody UML modeller," www.telelogic.com/products/rhapsody, from Telelogic, an IBM company.

[20] "MDWorkbench platform," www.mdworkbench.com, from SODIUS.

[21] "Synplify Pro," http://www.synplicity.com/products/synplifypro.

[22] *XUP Virtex-II Pro Development system*, http://www.digilentinc.com/xupv2p.