A High Level Synthesis Flow Using Model Driven Engineering

Sébastien Le Beux, Laurent Moss, Philippe Marquet, Jean-Luc Dekeyser

A High Level Synthesis Flow Using Model Driven Engineering

Sébastien Le Beux[1]*, Laurent Moss[1], Philippe Marquet[2] and Jean-Luc Dekeyser[2]

[1]Ecole Polytechnique de Montrál
2900 boulevard Edouard-Montpetit
Campus de l'Université de Montrál
2500, chemin de Polytechnique
Montrál, Quebec H3T 1J4, Canada
{Sebastien.Le-Beux, Laurent.Moss}@polymtl.ca

[2]LIFL and INRIA Lille Nord-Europe
Parc Scientifique de la Haute Borne
Park Plaza - Bât A - 40 avenue Halley
59650 Villeneuve d'Ascq, France
{Philippe.Marquet, Jean-Luc.Dekeyser}@lifl.fr

*contact author: Sebastien.Le-Beux@polymtl.ca

Key words: High Level Synthesis, hardware accelerators, Model Driven Engineering, intensive signal processing.

1 Introduction

Intensive Signal Processing (ISP) applications handle large amounts of data and are characterized by hierarchical and data parallel tasks, which manipulate multidimensional data arrays according to complex data dependencies. Performance requirements often preclude ISP applications from being implemented purely in software and instead call for using custom and efficient hardware accelerators. A hardware accelerator is an electronic design dedicated to the execution of a specific application. Its hardware architecture can be designed for a maximal parallelization of the algorithm needed to execute its application and for optimal execution support for regular and repetitive tasks. However, the complexity of hardware accelerators makes them difficult to manipulate at low abstraction levels (in a Hardware Description Language (HDL) for instance). The description of complex ISP applications is also error prone and tedious when using tools that constrain the number of dimensions of data arrays.

High Level Synthesis (HLS) seeks to simplify the design of hardware accelerators by describing applications at a high abstraction level and by generating the corresponding low level implementation. Application specification is easier at a high abstraction level since hardware designers do not need to handle all low level implementation details. HLS thus aims to achieve algorithm-architecture matching by construction, through the automated synthesis of a hardware architecture for an application specified at a high level. The automatic generation of low level implementations drastically reduces non-recurring engineering costs and the time to market compared to hand-tuned implementations in HDL. For these reasons, HLS tools have been increasingly successful among the hardware designer community. This trend is followed by the continual integration of new capabilities and functionality in the tools. Therefore, successful HLS has to support rapidly evolving technologies and be maintainable in order to capitalize on efforts. We present some design challenges faced by HLS and how model-driven engineering can meet them.

1.1 Design Challenges

This section presents some critical design challenges faced by both HLS tool *users* (i.e. hardware designers) and HLS tool *designers*.

1.1.1 HLS Tool User

From the tool user's point of view, the specification's abstraction level is sometimes not high enough to be really independent of low level implementation considerations: each particular implementation of a same application requires a particular specification. Such specifications are generally done in C or C-like syntax (e.g. Handel-C or SystemC) [15, 16, 30]. Unfortunately, such textual low level descriptions do not provide the opportunity to immediately extract specific information such as data dependencies, data parallelism and hierarchy. Conversely, a graphical representation associated to a factorized expression of the potential data parallelism and a powerful expression of data dependencies can solve the difficulties faced by HLS tool users. Moreover, a standard representation will considerably enhance discussions between the different field experts who take part in the specification of an application.

1.1.2 HLS Tool Designer

The gap between high abstraction levels and low abstraction levels is often bridged with one or several Internal Representations (IR) [15, 16, 19] in HLS tools. The set of concepts associated to an IR is generally difficult to handle

due to the lack of formal definition of these concepts and of the relations between them. Therefore, IR extension and maintenance (necessary for the development and evolution of the tool) rely on new specifications of the IR itself. Conversely, with a formal definition, extensions and maintenance are supported by the addition of new concepts and new relations. This ensures a high extensibility and maintainability of the IR, and consequently of the tool itself. Furthermore, the clear identification of concepts and relations in an IR allows a compilation process based on *concept to concept* translations to take care of the relations between these concepts. The consequences of the introduction of new concepts or relations in the source or target IR are then localized in the compilation (*i.e.* translation) process.

At the level of the design flow, a clear separation of the compilation phases implies a clear identification of the concepts, which helps to capitalize on the tool designer's efforts. Such tool development requires a strong methodology well-suited to designer habits and a stable and advanced technology to ensure the reuse, extension and maintainability of designer developments.

1.2 Proposed HLS Flow

This chapter presents a HLS flow dedicated to massively parallel ISP applications. The input is a graphical UML model of such an application. This model is at a high abstraction level: it is independent from any implementation technology. The output is an hardware accelerator able to execute the corresponding application. The generation of an hardware accelerator from the input model is handled by three successive transformations. The first transformation generates an internal representation of ISP applications and keeps only useful concepts necessary to represent them. The second transformation refines ISP models into RTL models; a RTL model represents an hardware accelerator able to execute the corresponding ISP application. The last transformation ensures the generation of the VHDL code corresponding to the hardware accelerator. Usual Electronic Design Automation (EDA) tools are then used to synthesize the resulting VHDL code to either a FPGA or an ASIC.

Our flow is entirely built with the Model Driven Engineering (MDE) methodology [32]: abstraction levels are defined in metamodels and refinements are done by model transformations. By allowing a clear specification of concepts and relations between concepts, MDE helps to reduce designer difficulties in developing and maintaining HLS tools. MDE also eases extensions of the proposed HLS flow:

- A *fine grain* extension extends the purpose of the HLS flow, for instance to manage control flow applications. This is successfully accomplished with the addition of new concepts in the metamodels and new rules in model transformations.

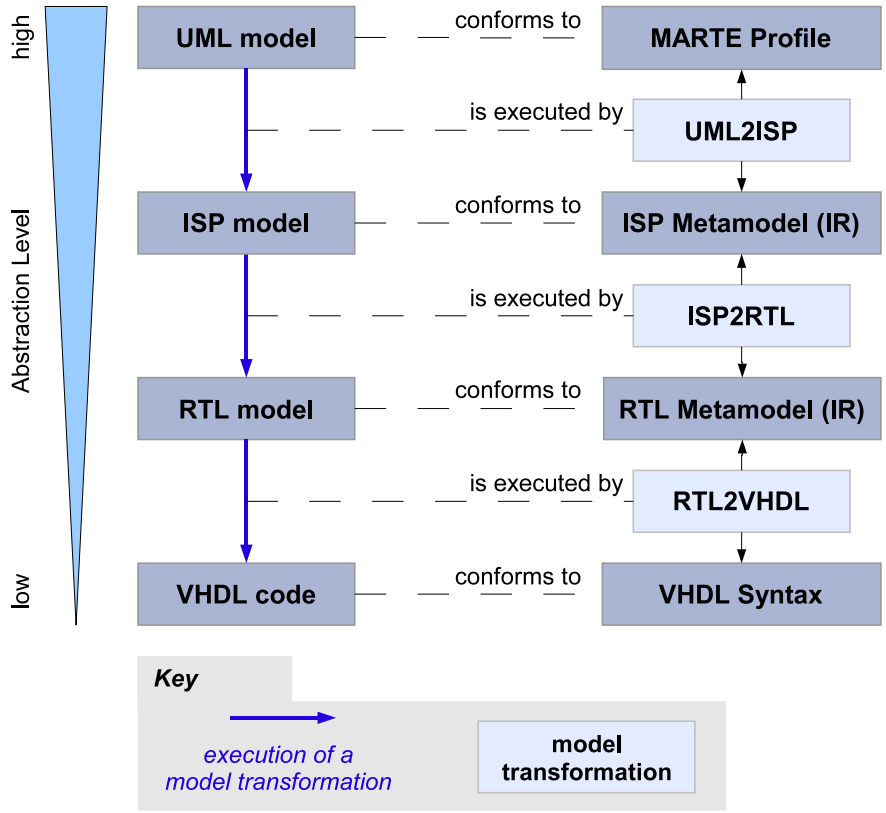


Fig. 1 The proposed HLS flow for ISP applications.

- A *coarse grain* extension consists of a modification of the design flow itself for new purposes. For instance, a model transformation could be added to generate Verilog [34] code from the RTL metamodel or to create a RTL model from another metamodel (a metamodel used in another tool for instance).

By successfully meeting the design challenges enumerated above, our HLS flow suitably solves the major difficulties encountered by both tool users and designers.

This chapter is organized as follow. Related works are presented in Section 2. Section 3 introduces MDE. We present our flow in Sections 4 and 5. Experimental results are presented in Section 6. The last section concludes this work.

2 Related works

During the last few years, the trend in HLS research [12, 15, 30, 16] and in commercial state-of-the-art HLS tools [8, 7] has been to generate HDL code from C/C++ or C-like languages such as Handel-C or SystemC. Using C code to generate hardware designs allows working with a well known language and at a higher abstraction level than RTL. However, since C is a sequential imperative language and neither a HDL nor a parallel programming language, C by itself is not well-suited to describing hierarchical applications that manage both task and data parallelism. Users of such a tool must strictly adhere to its coding guidelines both to make sure that their C-based input falls within the tool's synthesizable subset and to allow the tool to infer hierarchy and parallelism from the sequential code. System-level design languages such as SystemC help in specifying basic connections between the RTL blocks generated by HLS, but face the same challenge than C when it comes to actually performing HLS for each of these blocks. These works are compared to our flow in the following paragraph.

HLS tools usually infer data parallelism from loops with bounded indexes. However, the extraction of data dependencies across loop indexes from user-provided code is both tedious and error-prone: its complexity dramatically increases with the number of dimensions and the shape of the pattern. In our flow, the expression of data dependencies relies on *tilers* [5]. Tilers do not share these drawbacks since data dependencies are expressed explicitly and independently of each other through a matrix-vector expression. Each tiler has Origin, Paving and Fitting attributes which express how a data pattern is built from an array. The origin vector specifies the origin of the reference pattern in the array. The paving and fitting matrices respectively specify how an array is covered by patterns and how the patterns are constructed with array elements. A formal description of tilers is given in [5].

Some other approaches aim to specify application with code that follows the polyhedral model. This implies stricter user code restrictions than conventional HLS tools while offering more opportunities for optimizations in the IR using existing libraries [14, 3, 11]. Such approaches share the drawbacks previously identified for conventional HLS tools.

Another high level text-based approach would be for the user to specify the application as a mathematical formula. In [21], linear signal transforms are formally specified in the SPIRAL language [26] as products of structured sparse matrices and several RTL implementations, with different degrees of parallel execution, can be generated for a given user-provided transform. This formalism is well-suited to transforms which can be defined recursively, such as a Discrete Fourier Transform (DFT). However, this formalism is not well-suited to specifying a complete ISP application which involves several different transforms with complex data dependencies. A graphical formalism such as UML is better suited to representing the hierarchy, data dependencies and task parallelism of such an application. Furthermore, the use of MDE in

our flow allows greater opportunities for extensions by tool designers while the input language of [21] is narrowly domain-specific. Tools based on the polyhedral model or mathematical formulas could also be used to generate elementary components to be integrated in our flow, so these works are complementary to ours.

MDE has been increasingly adopted in the design of embedded systems in general [31]. The basic modeling formalism is the general purpose language UML, which offers attractive graphical representations. Because of its generality, UML is refined by the notion of *profile* to address domain-specific problems. There are currently several profiles for the design of embedded systems such as SysML [25], UML SPT [24], UML-RT [33], TUT Profile [17], ACCOR/UML [18] and Embedded UML [20]. Because all these profiles may potentially overlap, significant standardization efforts have been recently undertaken by the OMG, resulting in the single unified and effective MARTE standard profile [23], on which our HLS flow relies. MARTE stands for Modeling and Analysis of Real-Time Embedded systems. Among other things, MARTE provides mechanisms to express in a factorized way the potential parallelism available in applications. MARTE is thus well suited to the design of intensive signal processing applications and is used to model input applications in our HLS flow.

While these profiles allow one to specify a system with high level models, refinements from such models towards low level models have to be achieved. Some proposals use specific notations, defining a fully *executable* model semantics [22, 1, 28]. Such expressive notations allow one to define models with sufficient information so that the specified system can be completely generated. However, code is directly generated from the specifications, without any intermediary representation. The same is observed in previous works on VHDL code generation from UML [9, 4, 29, 35], where code is obtained directly by mapping UML concepts on VHDL syntax. These tools focus on finite state machines, so they do not address ISP applications. Furthermore, the absence of successive refinements leads to a lack of flexibility when targeting new abstraction levels or new languages. While these approaches rely on an abstraction of the system by using high level models, they only exploit a little of its benefits by being directly dependent on target languages or abstraction levels. Compared to these works, the high level synthesis flow we propose considers intermediate abstraction levels. This allows a smooth refinement from high abstraction level descriptions to low level implementations. This eases extensions of the flow (e.g. to target dynamically reconfigurable architectures [27] or to generate Verilog).

3 Model Driven Engineering

Complex systems can be easily understood via abstract and simplified representations: *models*. A model highlights the intent of a system without describing the implementation details. Several methodologies sought to manipulate models in the past decades, starting with Chen [6], and MDE [32] is one of these methodologies. It has been oriented towards the modeling of software engineering systems. Since the resulting models must be comprehensive and machine-readable, MDE also covers code generation. In this way, MDE stands apart from other model-based methodologies. This section details the major aspects of MDE that are *models*, *metamodels* and *model transformations*. General mechanisms are introduced and their relevance to ISP applications is highlighted and discussed.

3.1 Model and Metamodel

A model is an abstraction of reality. Models can be graphically observed from different points of view in order to highlight specific aspects of a given reality. Models focusing on aspects such as data parallelism and task parallelism can represent ISP applications well. A metamodel gathers the set of concepts and relations between the concepts used to describe a model according to a particular purpose (e.g. according to a given abstraction level). A model is then said to *conform to* a metamodel. Generally speaking, a metamodel defines the syntax of its models, like a grammar defines its language. A metamodel dedicated to the modeling of ISP applications at a given abstraction level thus gathers the corresponding set of concepts and relations. Such metamodel is assimilated to an IR in the HLS tool.

3.2 Model Transformations

A model transformation [10] is a compilation process which transforms a *source* model into a *target* model, as illustrated in Figure 2. The source and target models respectively conform to the source and target metamodels. A model transformation relies on a set of small *rules*. According to such a decomposition, particular and specific attention can be provided to the concepts or set of concepts handled by a given rule. For instance, data parallelism and task parallelism can be transformed with the specific attention they require.

Figure 3 illustrates a graphical representation of a simple rule used to transform components at a high abstraction level (c:Component) into components at a lower abstraction level (tc:Component. Each rule is divided into three parts: the *rule input pattern*, the *signature* and the *rule output pattern*.

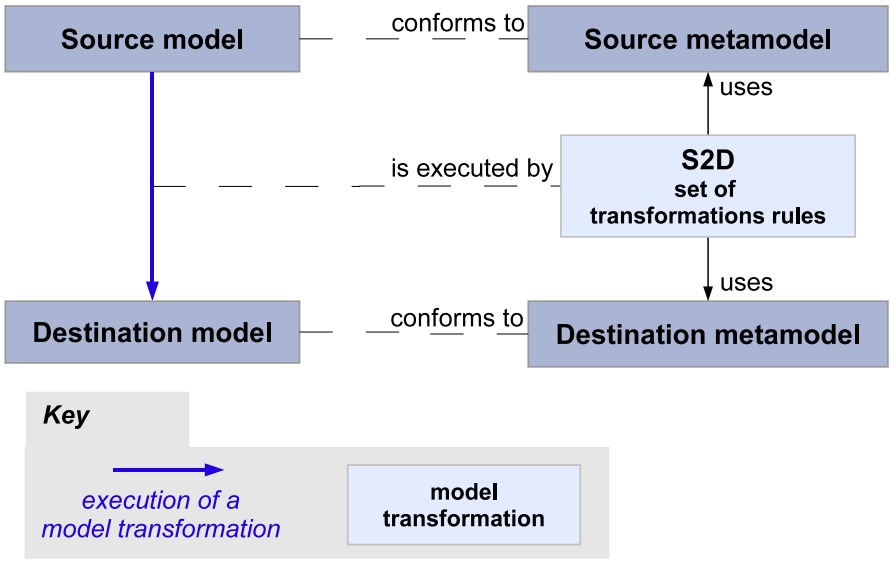


Fig. 2 A model transformation.

- The rule compares the input pattern to the source model in order to detect a concept or a set of concepts which trigger an execution. Such a condition is illustrated on the top part of the figure. In this example, the rule input pattern is very simple and contains the single concept c:Component.
- The signature of a rule is represented in the center of the graphical representation, it corresponds to the Component2Component concept. The signature allows the identification of the rule input and output patterns by the source and destination relations. During the transformation's execution, the signature identifies the set of concepts matching the rule input pattern, stores the information associated to these concepts and potentially calls other transformation rules (so called sub-rules).
- The rule output pattern, illustrated on the bottom part of the figure, corresponds to a set of concepts in the target model that are created during a rule execution. In the example, it includes four concepts. tc:Component is the main one since it is directly linked to the signature. Such a rule allows adding information during the transformation. For instance, clock and reset ports are attached to the transformed components (concepts clock:InputPort and reset:InputPort.

Model transformations are well-suited to performing refinements from high abstraction level specifications to code generation. For this purpose, model transformations add implementation details all along the compilation process. The code generation is a *model to text* transformation. Unlike model to model

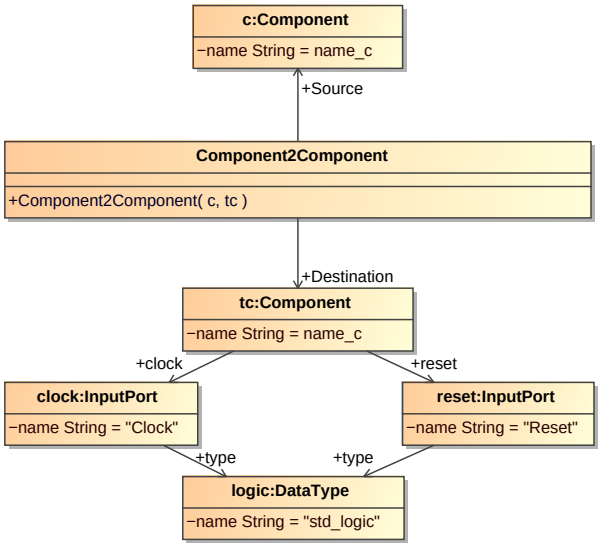


Fig. 3 Graphical representation of a transformation rule.

transformations, they are made of templates. However, the transformation principle remains the same.

4 High Level Specification Models

This section presents the high level models used in our design flow.

4.1 UML Model

Applications are modeled in UML, which is an OMG standard commonly used by the MDE community. We use the MARTE *profile* (i.e. extension) and its mechanisms to represent data parallelism, task parallelism and data dependencies. The concepts present in such UML models include high-level components with their inputs and outputs and how these components are instantiated, assembled and connected together to model the application. These concepts are illustrated here through the modeling of a matrix multiplication example.

Figure 4 represents the UML model of a matrix multiplication example which multiplies matrix MA and MB in order to produce a matrix MC. Ma-

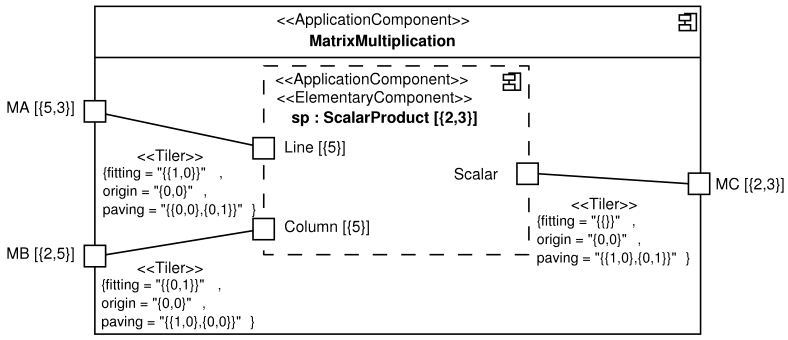


Fig. 4 UML model of the matrix multiplication example.

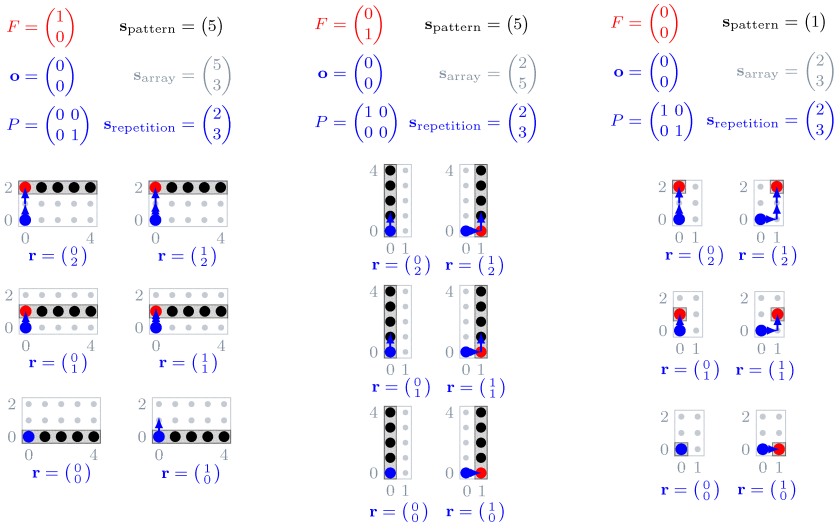


Fig. 5 The data dependencies expressed by the tilers of the MatrixMultiplication task.

trixMultiplication is a hierarchical task. The data consumed and produced by this task are respectively represented by the input ports MA and MB and the output port MC. In this example, each port corresponds to a matrix and the dimensions of each port are those of the corresponding matrix: 5×3 for MA, 2×5 for MB and 2×3 for MC. In such UML models, input and output data are represented as multidimensional arrays. There are no restrictions on the number of dimensions of data arrays. This allows the modeling of multidimensional data manipulations typical of ISP applications. For instance, video processing applications handle data over two spatial and one temporal

dimensions, whereas sonar chains process data over spatial, temporal and frequency dimensions.

The multiplicity {2,3} of the component instance sp of task ScalarProduct indicates that it is a data parallel task. In this example, the ScalarProduct task is repeated 2×3 times. Each iteration in the repetition space consumes input data patterns and produces output data patterns. *Tiler* connectors model the data dependencies used to generate these patterns. Each tiler models data dependencies linking a M-dimension data array to a N-dimension pattern. These data dependencies are not limited to compact and axis-aligned patterns.

Figure 5 represents the data dependencies expressed by the tilers used in the matrix multiplication example. The left-hand side of Figure 5 represents the tiler that links MA with Line, the center corresponds to the second input tiler and the right-hand side illustrates the output tiler. This figure represents the data consumed and produced in the data arrays (*i.e.* MA, MB and MC). For instance, in the first iteration on the repetition space $r = \binom{0}{0}$, the first line of data array MA and the first column of MB are read[1]. This line and column are used in the first iteration of task sp to produce the first data (*i.e.* the data at $(0, 0)$) in output data array MC). In iteration $r = \binom{1}{0}$, the first line of MA is read again while the second column of MB is used[2]. The data at position $(1, 0)$ in MC is computed and so on and so forth: by iterating over the whole repetition space, the whole output data array MC is produced.

The task sp consumes two input patterns (Line and Column) and produces the output pattern Scalar which corresponds to the result of a scalar product of a line and a column. ScalarProduct is an elementary task, i.e. a leaf in the application model. Its behavior is provided by the deployment part of the UML model [2] which links each elementary component to a given IP available in a library. Hence, the application model remains independent from the implementation target.

4.2 ISP Model and UML2ISP

Generally speaking, the ISP metamodel includes the interesting subset of MARTE dedicated to the description of ISP applications. Additional features allow to modify the *hierarchy* and to set the *data parallelism execution partitioning*. The hierarchy in ISP models can be modified through the loop transformations proposed in [13]. These loop transformations can modify, create or delete the hierarchy and move the data parallelism inside this hierarchy. Since the result of a loop transformation is an ISP model, successive loop transformations can be applied. The ISP metamodel allows specifying

[1] The line and the column are constructed through the fitting field.

[2] The shift of the line and the column are constructed by the paving.

a data parallel execution for each hierarchical task. Thus, part of the data parallelism can be executed sequentially while the other part is executed in parallel. To satisfy constraints of the RTL metamodel (which will be further detailed in Section 5), the following rules must be respected: the top level tasks are sequentially executed, and the lowest ones are executed in parallel. The set of specified executions defines the data-parallelism partitioning.

Specifying parallel or sequential execution for a set of hierarchical tasks is similar to the operations of allocation, scheduling and binding performed by conventional HLS tools [7]. Thus, specifying a given data parallel execution means allocating a given set of computing units, scheduling tasks to given clock cycles and binding tasks to the allocated computing units. However, conventional HLS tools typically allocate only simple fine-grained components, such as adders and multiplexers, contained in a fixed RTL library. On the other hand, the computing units allocated by our HLS flow can be pre-defined library components, user-defined components, or a hierarchical composition thereof, and our flow can thus allocate computing units covering a large spectrum of complexity and granularity.

UML2ISP ensures the transformation of an UML model into an initial ISP model. The hierarchy of the resulting ISP model matches the hierarchy defined in the UML model.

5 Implementation at a Low Level

5.1 RTL Model

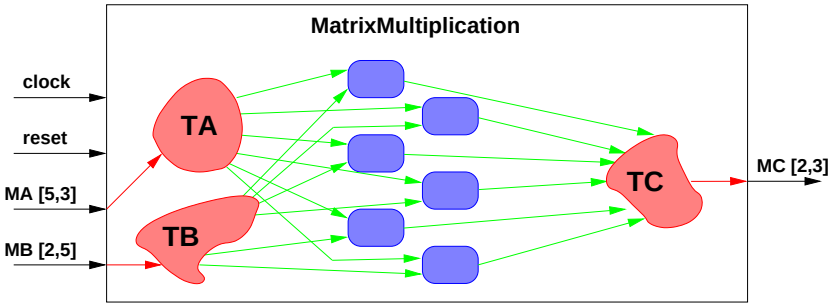
The RTL metamodel gathers the set of concepts used to describe hardware accelerators at the RTL level. Such hardware accelerators can execute the targeted ISP applications according to a specific execution model[3]. This execution model is data flow oriented and handles, among others, hierarchy, multidimensional data dependencies, data parallelism and task parallelism. The following provides an overview of the RTL metamodel.

In order to model hierarchical and well structured hardware accelerators, the RTL metamodel relies on a component based approach. Communications between components go through interfaces, which are composed of ports. There are input and output ports: a component can receive or send data. The shape and type of each port can also be specified. The RTL metamodel gathers the set of concepts used to implement parallel and sequential execution. These concepts are illustrated with the matrix multiplication example.

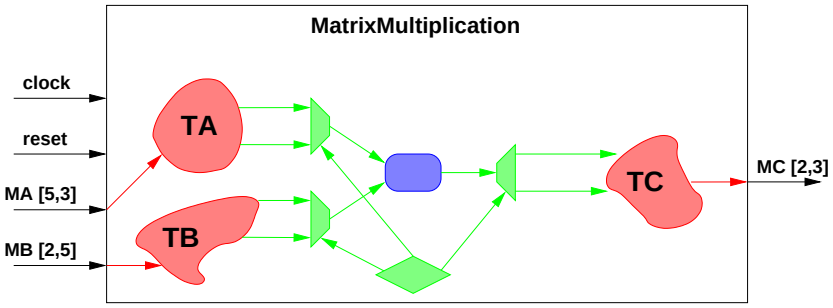
[3] The word *model* is different from the term *model* used in MDE. In order to avoid any confusion, the term *execution model* is used when dealing with the way an application is executed.

The repetition space around the ScalarProduct task is {2,3}. This task can be executed in parallel or sequentially :

- With a parallel execution, this task is instantiated 2×3 times, as illustrated in Figure 6(a). The six filled boxes represent the instances. Each instance computes a separate given line-column scalar product through connections to the customized data paths.
- With a sequential execution, ScalarProduct is instantiated only once, as illustrated in Figure 6(b). The overall computation is coordinated by a controller (represented with a lozenge) using multiplexers and demultiplexers (latches are also used in order to store data, they are not drawn on the figure in order to keep it readable). The controller iterates over the repetition space and, by controlling the multiplexers, sends the right data (*i.e.* the right line and the right column for this example) to the single computing unit. Conversely, the demultiplexers send the right data to the output tiler. A constraint we have is that the most top level task has to be sequentially executed.



(a) Parallel execution.



(b) Sequential execution.

Fig. 6 Hardware execution of the data parallelism in RTL models.

The mixed parallel/sequential execution relies on a combined use of these concepts. Thus, the hierarchy of the accelerator can be modified and the data

parallelism moved through this hierarchy. The data parallelism included in each hierarchical component is then executed independently from each other.

Data dependencies are implemented through data paths that are composed of connectors, buffers, latches, multiplexers and demultiplexers. These data paths can implement simple data array dependencies used in task parallelism as well as complex multidimensional data array dependencies used in data parallelism.

5.2 ISP2RTL Transformation

ISP models are independent from any implementation technology. Their automated implementation in either FPGA or ASIC technologies thus requires very specific refinements assumed by the ISP2RTL transformation. ISP2RTL is composed of rules. While some rules are very simple (such as the so-called one-to-one rules), some others are more tedious. For instance, the creation of a customized data path starting from a pure expression of data dependencies relies on a quite complex set of rules. In this set of rules, Tiler2InputTiler generates the data path and Tiler2InputTilerInstance interconnects the resulting data path into the component. The execution of Tiler2InputTilerInstance is triggered each time a part of ISP model matches the rule input pattern, as illustrated in Figure 7(a). This rule input pattern identifies the source and target of the tiler's connectors, the shape of the source port, the repetition space of the repeated task, etc. When Tiler2InputTilerInstance is triggered, the Tiler2InputTiler blackbox rule is automatically triggered; it analyzes the tiler's attributes (i.e. Origin, Paving and Fitting) in order to generate the right data path. For this purpose, Tiler2InputTiler computes, for each data in the pattern and for each iteration in the repetition space, the data read in the input array. Basically, the computation is done in two successive steps:

- based on the iteration in the repetition space, the origin coordinates in the input array are computed;
- based on the data in the input pattern and the origin coordinates, the coordinates of the data read in the input array are computed.

As a result, we obtain a set of one-to-one links between the input pattern and the input array. From these links, wires and buffers are allocated.

Figure 7(b) represents the execution of these rules onto an ISP model. Two inputs tilers, highlighted through the dashed shapes, match the Tiler2InputTilerInstance rule input pattern[4]. This rule is thus triggered twice and Tiler2InputTiler is subsequently triggered, resulting in TA and TB. Due to the simplicity of the example, the generated data paths (not detailed in the figure) only include wires.

[4] the figure represents the UML model, which is very close to the ISP one.

(a) Graphical representation of the rule. (b) Execution of the rule.

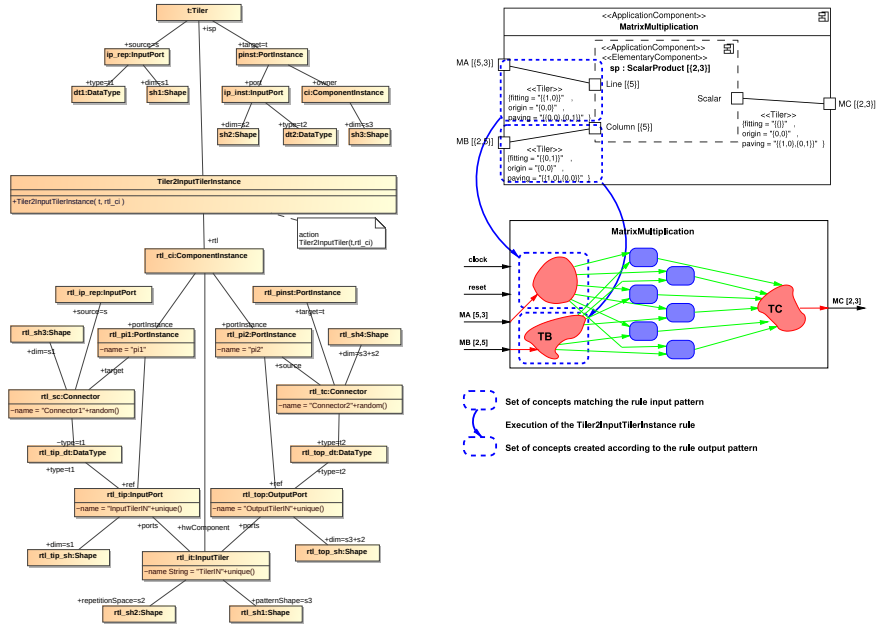


Fig. 7 The Tiler2InputTilerInstance rule transforms the tilers into customized data paths.

5.3 RTL2VHDL Transformation

The RTL metamodel is independent from any HDL syntax, but is low level enough to allow code generation through the RTL2VHDL transformation. VHDL code generation from the RTL metamodel is performed through templates that navigate the RTL model to find their associated concepts and print them in a VHDL syntax. Figure 8 a) presents the template associated to the Component concept in the RTL metamodel. Figure 8 b) shows excerpts of the generated code for the MatrixMultiplication component. Special attention was given to keep multidimensional data arrays and data parallelism factorized. The generated code can be directly synthesized (e.g. on FPGA) with standard logic synthesis tools.

6 Case Study

This section illustrates the correctness and efficiency of our flow dedicated to intensive signal processing applications. For this purpose, a correlation

(a) Textual representation of a rule.

```
ENTITY <%=element.getName()%> IS
PORT (
<%=ts.generate(element.getClock())%>;
<%=ts.generate(element.getReset())%>
<%for (Port p : (List<Port>) element.getPorts())
{%>;
  <%=ts.generate(p)%><%
}%>);
END <%=element.getName()%>;
```

(b) Resulting VHDL code for the example.

```
ENTITY MatrixMultiplication IS
PORT (
 clock : IN Std_Logic;
 reset : IN Std_Logic;
MA : IN Type_5_3_Integer;
MB : IN Type_2_5_Integer;
MC : OUT Type_2_3_Integer);
END MatrixMultiplication;
```

Fig. 8 A rule transforming RTL components into VHDL entities.

algorithm is studied. This algorithm is well known and frequently used in intensive signal processing. Eq. 1 gives its mathematical formulation, which is composed of a set of multiplications and additions that can be executed in parallel.

$$C_{cy}(j) = \sum_{i=0}^{1023} c(i) \cdot y(i+j) \tag{1}$$

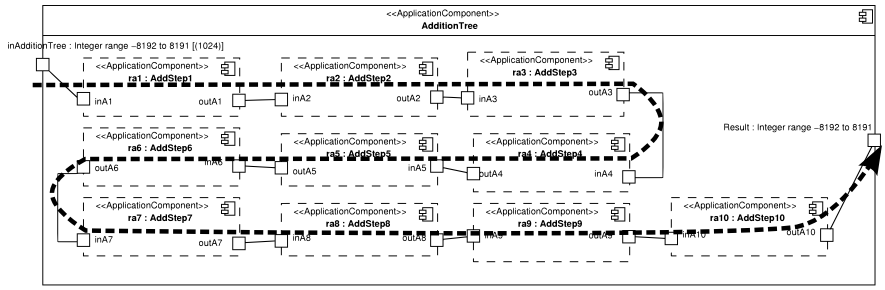
6.1 UML Model

The correlation algorithm application has been modeled in UML and independently from any implementation. All the data parallelism and task parallelism are extracted so that they can be used to generate an efficient implementation. In the following, representative parts of the UML are presented.

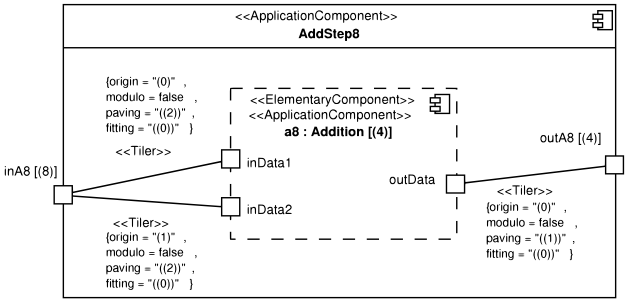
Figure 9(a) illustrates component *AdditionTree* that realizes the sum in the correlation algorithm. The input port *inAdditionTree* is composed of 1024 data (i.e. the data to sum) and the output port is a scalar value (i.e. the result of the sum). The 10 component instances represent the 10 pipeline stages of the tree topology used to realize the sum and the data flow is represented by the dashed arrow. Figure 9(b) represents the data parallel task `AddStep8`, which is the 8^{th} component instantiated in the pipeline stage. Its input port *inA8* and its output port *outA8* are respectively composed of 8 data and 4 data. The elementary task *a8* is repeated 4 times to realize the necessary computation.

6.2 Generated Hardware Accelerator

From the UML model presented above, our flow automatically generates a hardware accelerator able to execute the correlation algorithm. For this pur-



(a) additions at task parallelism level (data flow represented by dashed arrow).



(b) multiplications at the data parallelism level.

Fig. 9 UML model of the correlation algorithm.

pose, an ISP model, a RTL model and a VHDL code are successively generated, as illustrated in Figure 10.

The resulting VHDL code was synthesized for an Altera Stratix 2S60 FPGA using the Quartus tool from Altera. Figure 11(a) illustrates the 6 last stages of the tree topology and the corresponding reduction of data arrays from a pipeline stage to another. Figure 11(b) represents the synthesis results for the 8^{th} pipeline stage. In the figure, marks **1** and **5** correspond to the input and the output ports, marks **2x** and **4** point out the generated components that resolve data dependencies initially expressed with tilers. Finally, marks **3x** represent the four elementary tasks that realize parallel execution of additions. By expressing the data dependencies through tilers in UML, our flow finds that data dependencies are efficiently implemented in hardware with shift register, as illustrated in Figure 11(c).

The relevance of the HLS flow is evaluated through a comparison between a manually implemented hardware accelerator and an automatically generated one. Synthesis results are summarized in Table 1. As a first result, the latency of both accelerators remains strictly the same. The maximum frequency of the automatically generated hardware accelerator is 1.9% higher compared to the manually implemented one. The number of resources required to implement the accelerators are also close to each other since only

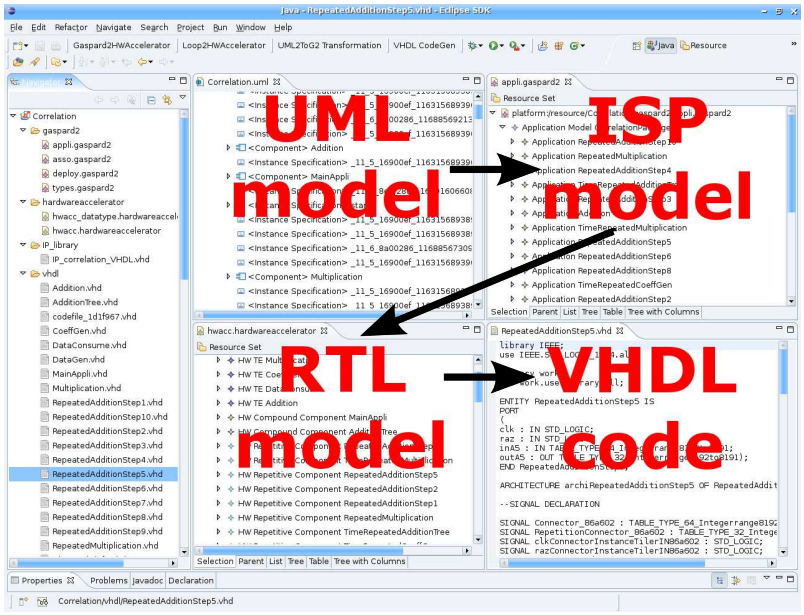


Fig. 10 Generating VHDL code from UML model.

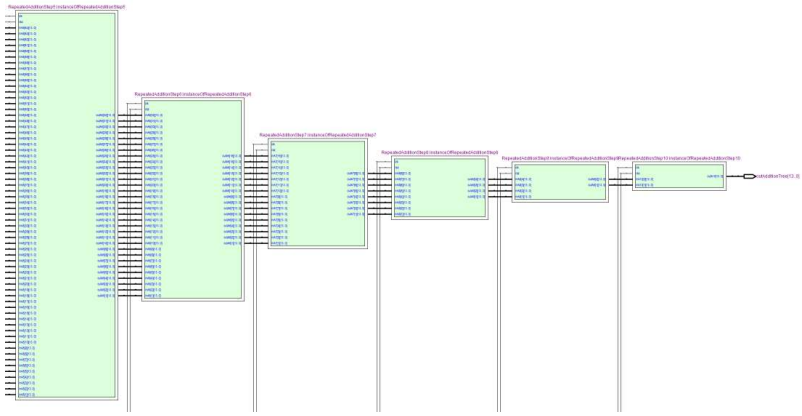
10.7% additionnal resources are necessary for the automatically generated hardware accelerator. Whereas manually implementing the hardware accelerator takes weeks, modeling the UML model of the application only takes a few hours. Hence, development time can be greatly shortened with the HLS flow at the price of using additional hardware resources.

Table 1 Synthesis results of the manually and automatically generated hardware accelerators.

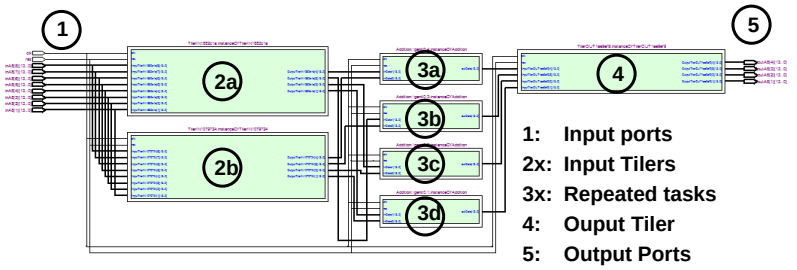
Version	Max frequency (in MHz)	Latency (cycles)	Used resources (ALUTs)	Development time
Manual	213	11	17006	weeks
Automated	217	11	18834	hours

7 Conclusion

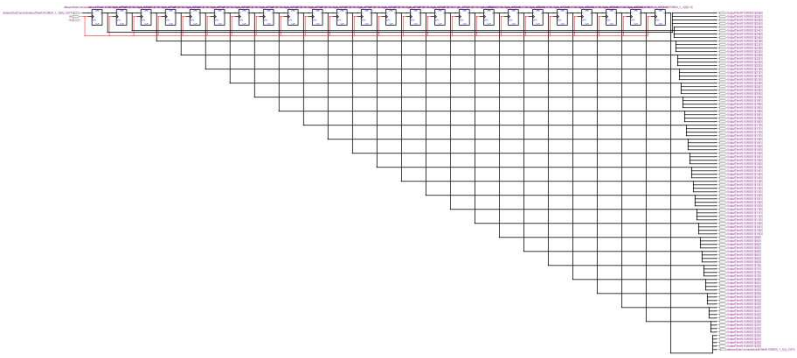
This chapter advocates the use of the MDE methodology for high level synthesis. In order to demonstrate the benefits of MDE, we developed a model-



(a) synthesized addition tree.



(b) synthesized data parallel task.



(c) data dependencies synthesized as shift register.

Fig. 11 Synthesis results of the generated hardware accelerator.

based HLS flow. This flow relies on a precise definition of features, such as data parallelism and data dependencies, that are important for intensive signal processing applications. We have also shown that MDE provides key benefits to both users and designers of our HLS flow: users work in a standardized unified graphical environment and designers can easily extend and maintain the flow.

From applications modeled at a high abstraction level in UML, the flow automatically performs successive refinements and generates the corresponding VHDL code. Such refinements rely on a clear identification of concepts in the different abstraction levels and on a suitable decomposition of the model transformations into rules. We have validated the relevance of our HLS flow for correlation algorithms. The quality of results achieved by our HLS flow is almost as good (same latency with 10.7% more hardware resources) than that achieved with hand-coded VHDL. The flexibility and productivity advantages of high-level specifications and automated refinements more than outweigh the small degradation in the quality of results. Also, the performance of such an application-specific hardware accelerator is generally much higher than that achieved by software running on a (non-application-specific) processor.

MDE could also enable extensions to the flow to target other types of applications, other implementation languages or other abstraction levels.

References

1. Marcus Alanen, Johan Lilius, Ivan Porres, Dragos Truscan, Ian Oliver, and Kim Sandstrom. Design method support for domain specific soc design. In *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES'06)*, pages 25–32, 2006.
2. Rabie Ben Atitallah, Eric Piel, Smail Niar, Philippe Marquet, and Jean-Luc Dekeyser. Multilevel MPSoC simulation using an MDE approach. In *IEEE International SoC Conference (SoCC 2007)*, Hsinchu, Taiwan, September 2007.
3. Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.
4. Dag Björklund and Johan Lilius. From UML Behavioral Descriptions to Efficient Synthesizable VHDL. In *Proceedings of the 20th IEEE Norchip Conference*, November 2002.
5. Pierre Boulet. Array-OL revisited, multidimensional intensive signal processing specification. Research Report RR-6113, INRIA, February 2007.
6. Peter Pin-Shan Chen. The entity-relationship model–toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.
7. P. Coussy, D.D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.
8. Philippe Coussy and Adam Morawiec, editors. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, New York, NY, 2008.

9. Frank P. Coyle and Mitchell A. Thornton. From UML to HDL: a model driven architectural approach to hardware-software co-design. *Information Systems: New Generations Conference (ISNG)*, pages 88–93, April 2005.
10. Krzysztof Czarnecki and Simon Helsen. Classification of model tranformation approaches. In *Proceeding of OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture*, 2003.
11. Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout, and Dirk Stroobandt. From loop transformation to hardware generation. In *Proceedings of the 17th ProRISC Workshop*, pages 249–255, Veldhoven, November 2006.
12. Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the streams-C C-to-FPGA compiler: an applications perspective. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field Programmable Gate Arrays (FPGA)*, pages 134–140, 2001.
13. Calin Glitia and Pierre Boulet. High level loop transformations for multidimensional signal processing embedded applications. In *International Symposium on Systems, Architectures, MOdeling, and Simulation (SAMOS VIII)*, Samos, Greece, July 2008.
14. Anne-Claire Guillou, Patrice Quinton, and Tanguy Risset. Hardware synthesis for multi-dimensional time. In *IEEE 14th International Conference on Application-specific Systems, Architectures and Processors (ASAP 03)*, pages 40–51, The Hague, The Netherlands, June 2003.
15. Zhi Guo, Betul Buyukkurt, Walid Najjar, and Kees Vissers. Optimized Generation of Data-Path from C Codes for FPGAs. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 112–117, Washington, DC, USA, 2005. IEEE Computer Society.
16. Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *Intl. Conf. on VLSI Design*, pages 461–466, 2003.
17. Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen, Jouni Riihimäki, and Kimmo Kuusilinna. UML-based multiprocessor SoC design framework. *Transactions in Embedded Computing Systems*, 5(2):281–320, 2006.
18. P. Lanusse, S.Gérard, and F.Terrier. Real-time modeling with UML : The ACCORD approach. In *UML 98 : Beyond the notation*, Mulhouse, France, 1998.
19. Jack Lo, Susan Eggers, Henry Levy, and Dean Tullsen. Compilation issues for a simultaneous multithreading processor. In *Proceedings of the First SUIF Compiler Workshop*, pages 146–147, January 1996.
20. G. Martin, L. Lavagno, and J. Louis-Guerin. Embedded UML: a merger of real-time UML and co-design. In *Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES)*, pages 23–28, 2001.
21. P.A. Milder, F. Franchetti, J.C. Hoe, and M. Puschel. Formal datapath representation and manipulation for implementing DSP transforms. In *2008 45th ACM/IEEE Design Automation Conference*, pages 385–90, Piscataway, NJ, USA, 2008.
22. Kathy Dang Nguyen, Zhenxin Sun, P. S. Thiagarajan, and Weng-Fai Wong. Model-driven SoC design via executable UML to SystemC. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 459–468, Washington, DC, USA, 2004. IEEE Computer Society.
23. Object Management Group. A UML profile for MARTE, 2007. `http://www.omgmarte.org`.
24. Object Management Group, Inc., editor. *(UML) Profile for Schedulability, Performance, and Time Version 1.1.* `http://www.omg.org/technology/documents/formal/schedulability.htm`, January 2005.
25. Object Management Group, Inc., editor. *Final Adopted OMG SysML Specification.* `http://www.omg.org/cgi-bin/doc?ptc/06-0504`, May 2006.
26. Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang

Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.
27. Imran Rafiq Quadri, Samy Meftali, and Jean-Luc Dekeyser. From MARTE to dynamically reconfigurable FPGAs : Introduction of a control extension in a model based design flow. Technical report, DART - INRIA Lille - Nord Europe - INRIA - CNRS : UMR8022 - Université des Sciences et Technologies de Lille - Lille I, 2009.
28. E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A model-driven design environment for embedded systems. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 915–918, New York, NY, USA, 2006. ACM.
29. Medard Rieder, Rico Steiner, Cathy Berthouzoz, Francois Corthay, and Thomas Sterren. *Rapid Integration of Software Engineering Techniques*, chapter Synthesized UML, a Practical Approach to Map UML to VHDL. Springer Berlin / Heidelberg, 2007.
30. Robert Rinker, Margaret Carter, Amitkumar Patel, Monica Chawathe, Charlie Ross, Jeffrey Hammes, Walid A. Najjar, and Wim Böhm. An automated process for compiling dataflow graphs into reconfigurable hardware. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1):130–139, 2001.
31. Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):41–47, February 2006.
32. Ed Seidewitz. What models mean. *IEEE Softw.*, 20(5):26–32, 2003.
33. Bran Selic. Using UML for Modeling Complex Real-Time Systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260, London, UK, 1998. Springer-Verlag.
34. Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, fourth edition, May 1998.
35. Jorgiano Vidal, Florent de Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. A co-design approach for embedded system modeling and code generation with UML and MARTE. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2009.