Model Driven Engineering Benefits for High Level Synthesis

Sébastien Le Beux, Philippe Marquet, Jean-Luc Dekeyser

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Model Driven Engineering Benefits for High Level Synthesis

Sebastien Le Beux — Philippe Marquet — Jean-Luc Dekeyser

N° 6615

August 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6615--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche LILLE - NORD EUROPE

Model Driven Engineering Benefits for High Level Synthesis

Sebastien Le Beux* , Philippe Marquet† , Jean-Luc Dekeyser‡

Thème COM — Systèmes communicants
Équipes-Projets DaRT

Rapport de recherche n° 6615 — August 2008 — 43 pages

Abstract: This report presents the benefits of using the Model Driven Engineering (MDE) methodology to solve major difficulties encountered by usual high level synthesis (HLS) flows. These advantages are highlighted in a design space exploration environment we propose. MDE is the skeleton of our HLS flow dedicated to intensive signal processing to demonstrate the expected benefits of these software technologies extended to hardware design. Both *users and designers* of the design flow benefit from the MDE methodology, participating to a concrete and effective advancement in the high level synthesis research domain. The flow is automatized from UML specifications to VHDL code generation and has been successfully evaluated for the conception of a video processing application.

Key-words: High Level Synthesis, Design Space Exploration, hardware accelerators, Model Driven Engineering, MARTE standard profile.

* *Sebastien.Le-Beux@lifl.fr*
† *Philippe.Marquet@lifl.fr*
‡ *Jean-Luc.Dekeyser@lifl.fr*

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley, 59650 Villeneuve d'Ascq
Téléphone : +33 3 59 57 78 00 — Télécopie : +33 3 59 57 78 50

Bénéfices de l'Ingénierie Dirigée par les Modèles pour la Synthèse de Haut Niveau

Résumé : Ce rapport présente comment l'Ingénierie Dirigée par les Modèles (IDM) permet de résoudre les difficultés majeures que rencontrent les outils dédiés à la synthèse de haut niveau. Les paradigmes de l'IDM nous servent de base à la définition d'un flot de conception dédié aux applications de traitement de signal intensif. Ce flot de conception génère automatiquement du code VHDL à partir de modélisations en UML. Il permet à la fois à ses utilisateurs et à ses concepteurs de tirer profit de l'utilisation de l'IDM. Son efficacité est évaluée au travers d'une application dédiée au traitement vidéo.

Mots-clés : Synthèse de haut niveau, exploration, accélérateurs matériels, ingénierie dirigée par les modèles, profil standard MARTE.

1 Introduction

High Level Synthesis (HLS) aims to simplify the hardware design of accelerators by describing applications at high abstraction levels and by generating the corresponding low level implementation. The specification of the applications becomes easier at high abstraction level. A HLS tool *user* is not concerned with the implementation details. The automatic generation of low level implementation drastically reduces the fabrication cost and the time to market compared to the hand-tuned implementations in Hardware Description Languages (HDL). For these reasons, the HLS tools encounter a large success by the hardware designers community. This trend is followed by the regular integration of new capabilities and functionality in the tools. Therefore, successful HLS has to face the rapidly evolving technologies and has to be maintainable in order to capitalize efforts provided by the tool *designers*.

1.1 Design Challenges

From the tool user point of view, the abstraction level of the specification 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) [21, 22, 41]. Unfortunately, such textual low level descriptions do not provide the opportunity to immediately extract specific information such as data dependencies, data parallelism and hierarchy. Furthermore, textual descriptions have no ability to highlight the different concepts, as opposed to graphical descriptions.

Intensive Signal Processing (ISP) applications handle large amount of data manipulated by sets of regular tasks. Such applications are characterized by hierarchical and data parallel tasks, which manipulate multidimensional data arrays according to complex data dependencies. ISP applications are efficiently executed by customized hardware accelerators. A hardware accelerator is an electronic design, dedicated to the execution of a specific application. It allows a maximal parallelization of the computation needed to execute this application. It provides an optimal execution support for regular and repetitive tasks. However, the complexity of the hardware accelerators makes them difficult to manipulate at low abstraction levels (in HDL language for instance). The description of complex ISP applications is error prone and becomes tedious when using tools that constrain the number of dimensions. Alternatively, a graphical representation associated to a factorized expression of the potential data parallelism and a powerful expression of data dependencies can solve the difficulties encountered by the HLS tool users. Moreover, a standard representation that respects the user habits will considerably enhance the exchanges between the different field experts acting in the description of an application.

ISP applications are becoming more and more sophisticated and resource demanding. Meeting the performance requirements relies on the semiconductor technology, which enables to integrate even more hardware resources on a single chip. This contributes to increase the possible implementation solutions (*i.e* the design space) and then makes difficult the design of an effective hardware accelerator. As a consequence, the productivity of flow users strongly gets penalized. It thus becomes necessary to deal with an efficient design space

exploration; *i.e.* how the analysis obtained from the implementation level are exploited for an efficient redesign by modifying the high abstraction level representation of the application. So, an important challenge here is to provide efficient design space exploration strategies that adequately address all these issues concerning complex and resources demanding ISP applications.

The gap between the high abstraction levels and the low abstraction levels is often assumed using one or several Internal Representations (IR) [21, 22, 30]. The set of concepts defined for an IR is generally difficult to handle due to the lack of formal definition of these concepts and of the relations between these concepts. Therefore, IR extensions and maintainability (necessary to ensure the tool development evolutions) rely on new specifications of the IR itself. Alternatively, with a formal definition, we just need to add new concepts and new relations. A graphical representation of the IR provides an additional documentation that considerably highlights the concepts and relations. This ensures a high extensibility and maintainability of the IR, and consequently of the tool itself.

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 the target IR are then localized in the compilation (*i.e.* translation) process.

At the level of the global design flow, a clear separation of the compilation and exploration phases implies a clear identification of the concepts, which helps to capitalize the efforts of the tool designer. Such a development of a tool requires a strong methodology suitable for the designers habits and a steady and advanced technology to ensure the reuse, extension and maintainability of the designer developments. Concerning ISP, the exploration process is based on heuristics that aim to apply usual loop transformations inside IR in order to find the most effective hardware design according to resource and time constraints. The SPARK tool [22] separates loop transformations and optimization heuristics, allowing independent modifications and extensions: this demonstrates the interest in the decomposition of relevant difficulties into smaller difficulties ones.

1.2 Our HLS Flow

This paper presents a HLS flow dedicated to massively parallel ISP applications, the flow is entirely build within the Model Driven Engineering [43] (MDE) methodology. Thanks to MDE, we successfully answer the major difficulties, for both users and designers of the HLS flow. Applications are graphically specified at a high abstraction level with Unified Modeling Language (UML). Such graphical representation facilitates exchanges and communications with various field experts, and makes the specification easy to reuse, extend and maintain. Factorized expressions of parallelism, multidimensional data arrays and powerful constructs of data dependencies are managed thanks to the use of the MARTE standard profile. MARTE thus provides a modeling environment well suited to massively parallel ISP applications, even to the more complex and critical ones (e.g. sonar chain, obstacle detection algorithm). From such a high level representation of a given ISP application, our flow automatically generates hardware accelerators able to execute the modeled application. The design space exploration is considerably facilitated since users only deal the modification of

high abstraction models, the corresponding implementations being automatically regenerated. Furthermore, an estimation process characterises the execution performance and the area cost of the generated hardware accelerator. It is thus no more necessary to use external tools to evaluate the effectiveness of generated designs, such facility accelerates the design space exploration.

The usage of MDE also reduces the designer difficulties to develop and maintain HLS tools. Each abstraction level (*i.e.* IR) is graphically expressed in UML by designers (UML is therefore used by both users and designers), highlighting the concepts and the relations between concepts. The refinements from high abstraction levels down to low abstraction levels are taken over by transformation steps, each step being composed of simple *rules*, which compile a well defined subset of concepts. For instance, a set of rules compiles data dependencies while another one manages data parallelism: a complex compilation process is divided into small and easily handled tasks. Each rule can be modified independently from other rules and new rules can be added to extend the compilation process. An exploration process modifies a high abstraction level specification according to the generated low abstraction level implementations. The overall exploration process thus benefits from the implementation details provided by successive refinements.

This paper is organized as follows: Section 2 introduces the Model Driven Engineering methodology and presents preliminary results about its relevance to high level synthesis of ISP applications. According to these considerations, we developed a HLS flow dedicated to massively parallel ISP applications. We present this flow in Section 3. Section 4 deals with strategies that allow one the rapidly explore the design space in our flow. Section 5 provides a brief overview of our flow and presents the corresponding tool-set. The successful utilization of our flow for a multimedia application is illustrated in Section 6. Some related works are presented in Section 7, highlighting the originality of our work. The last section concludes this work.

2 Model Driven Engineering

Complex systems can be easily understood thanks to abstract and simplified representations: *models*. Graphical representations of models considerably facilitate the comprehension of a given model. The UML language is often used for such graphical representations since its normalization in 1997. A model highlights the intention of a system without describing the implementation details. Several methodologies aimed to manipulate model in past decades, like Chen [8] in seventies. MDE [43] inherits from these methodologies. It is definitely oriented towards the modeling of software engineering systems. The resulting models must be comprehensive and interpretable by computer. MDE also covers the code generation, which puts a model in concrete form. In this way, MDE stands apart from the others methodologies based on models. This section details the major aspects of MDE that are *model*, *metamodel* and *model transformations*. Generals mechanisms are introduced and their relevant usage for ISP applications are highlighted and discussed.

2.1 Model

A model is an abstraction of the reality which is composed of concepts and relations. The concepts represent an abstraction of *things* in the reality and relations represent the links between the things. A model can be graphically observed from different points of view (*views* in MDE), which highlight specific aspects of the reality.

The ISP applications rely on clearly identified elements (things) such as data parallel tasks, data dependencies and multidimensional data arrays. The abstraction of each element corresponds to a concept in a model, the dependencies between these elements are represented by relations. Models can represent abstract descriptions of the ISP applications and facilitate their specifications and modifications because each concept and relation is clearly identified. Moreover, views can help to represent and document models of ISP applications by highlighting the relevant concepts and relations according to a particular purpose.

2.2 Metamodel

A metamodel gathers the set of concepts and relations between the concepts used to describe a model, *i.e.* the reality according to a particular purpose (a given abstraction level for instance). 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 thus can gather the set of concepts and relations necessary to represent the ISP applications at a given abstraction level. Such metamodel is assimilated to an IR, from which the HLS tools relies on to internally represent ISP applications.

2.3 Model Transformations

In MDE, a model transformation [11] is a compilation process which transforms a *source* model into a *target* model, as illustrated Figure 1. The source and the target models are respectively conformed to the source and the target metamodels. A model transformation relies on a set of *rules*. Each rule clearly identifies concepts in the source and the target metamodels. Such decomposition facilitates the extension and the maintainability of a compilation process: new rules extend the compilation process and each rule can be modified independently from the others.

The rules are specified with languages. The language may be imperative: it describes *how* a rule is executed; it can be declarative, it describes *what* is created by the rules. Declarative languages are often used in MDE because the rules objectives can be specified independently from the execution. A graphical representation is a good approach for representing the rules expressed in a declarative language [19, 32].

Such graphical representation of a basic rule is illustrated Figure 2. This representation is conformed to TrML, the Transformation Modeling Language [16], which aims to facilitate documentations and exchanges of the rules. The rule represented in this figure transforms components at a high abstraction level (c:Component) into components at a lower abstraction level (tc:Component. The

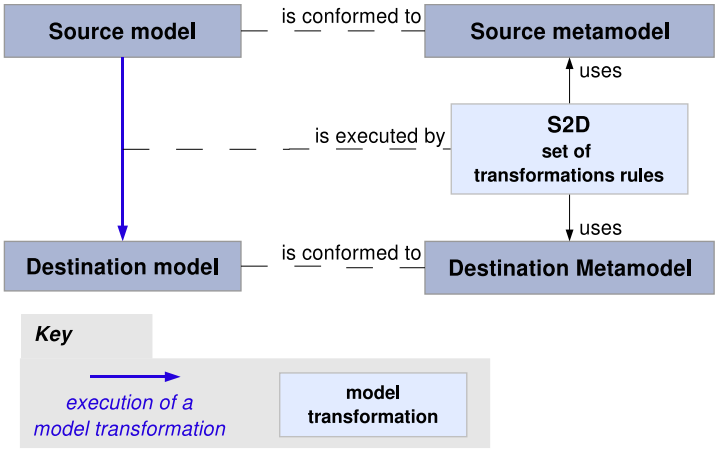


Figure 1: A model transformation.

following describes the semantic of the representation and details the represented rule.

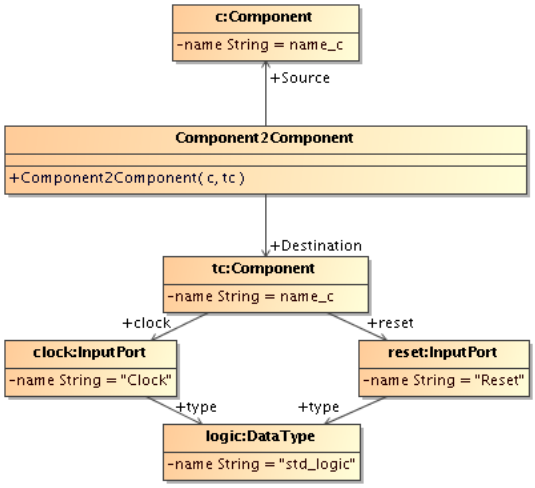


Figure 2: Graphical representation of a transformation rule.

A transformation rule is divided into three parts: the *rule input pattern*, the *signature* and the *rule output pattern*.

- 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 condition is illustrated on the top part of the graphical representation of a rule. In the example, the rule input pattern is trivial and contains the single concept c:Component which corresponds to the concept of applicative components in the source model. Each concerned component is characterized by a name.

- The signature of a rule is represented in the center of the graphical representation, it corresponds to the Component2Component concept in Figure 2. The signature allows the identification of the rule input and output patterns thanks to the source and destination links. During the transformation execution, the signature identifies the sets of concepts matching the rule input pattern, stores the information relative to these concepts and potentially calls other transformation rules (so called sub-rules).
- The rule output pattern corresponds to a set of concepts in the target model that are created during a rule execution. The rule output pattern is illustrated on the bottom part of the graphical representations. In the example, it includes four concepts. The main one is the tc:Component concept (the main concept is the one attached to the rule). The attributes of these concepts often depend on the elements that are stored by the signature of a rule. For instance, the name of the tc:Component concept corresponds to the name of the c:Component concept. clock:InputPort and reset:InputPort are attached to the main concept, they both have the same data type (logic:DataType concept).

For this purpose, there exist different kinds of rules which can be sequentially called, mutually called or automatically executed (there is no condition). Moreover, while certain rules are directly called from the transformation engine (the *top rules*), some others can only be called from others rules. These features allow to realize complex transformations.

As previously mentioned, a model transformation is composed of a set of rules. On the right-hand side of the Figure 3, the rule Component2Component is a part of the model transformation S2D. The rule input pattern and the rule output pattern respectively correspond to a part of the source and the target metamodels. In this example, we consider a source metamodel which is dedicated to the high abstraction level description of applications. At this level, the applications are composed of tasks (concept Component in the source metamodel). We then consider that the target metamodel is dedicated to the low implementation level description of hardware designs which are able to execute applications. In this metamodel, the component concept represents a hardware component which is able to execute a task. Each hardware component has a clock port and a reset input. The rule Component2Component is in charge of the transformation of the tasks in a source model (on the top left-hand side of Figure 3) into hardware components in the target model (on the bottom left-hand side). The source model is composed of three tasks named A, B and C. The Component2Component rule transforms these components into hardware components with input ports in the target model.

The result of a model to model transformation is a model. As models are not directly executable by computers, it is necessary to generate codes. In MDE, the code generation is considered as a *model to text* transformation, which is composed of a set of templates (instead of a set of rules for usual model to model transformations). In MDE based design flow, the last transformation is often a model to text transformation.

Model transformations are suitable in HLS in order to perform refinements going from the high abstraction level specifications to the code generations. For this purpose, model transformations add implementation details all along the compilation process. Several model transformations can be defined from the same

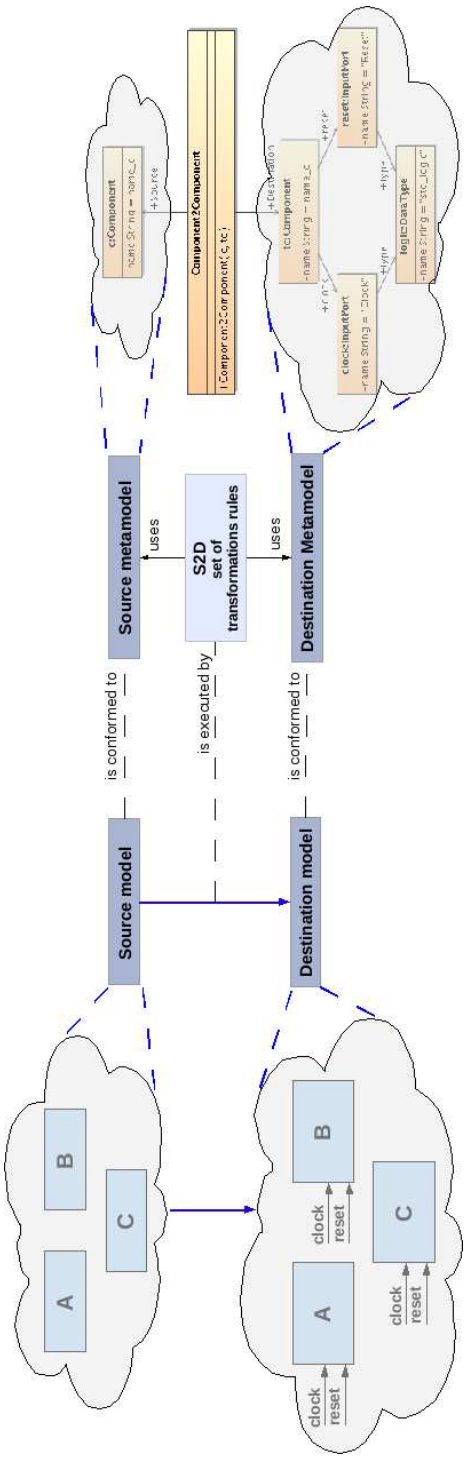


Figure 3: Models, metamodels, transformations and rules.

abstraction level, but towards different lower levels, offering opportunities to generate several implementations of a given specification. The decomposition into rules separates the compilation of the different concepts. Therefore, particular and specific attention can be provided in accordance to the concepts or set of concepts handled by a given rule. For instance, data parallelism and task parallelism can be compiled in two different manners.

2.3.1 Transformation Engine

Several tools allow to model and execute transformation rules: Kermeta [33, 46], ATL [23], ModelMorf [13], etc. QVT [38] (Query, View, Transformations) is the only one standardized at the Object Management Group[1] (OMG). QVT allows the graphical representation and the textual description of the rules. However, this standard is not commonly used since the tools which support this standard only manage the imperative part of the rules[2], the part which does not rely on the notion of rule. An alternative solution to QVT is EMF [15] (Eclipse Modeling Framework), a commonly used Java library which allow to create and modify models. The rules used in our HLS flow are implemented with EMF[3].

2.4 Unified Modeling Language

UML (Unified Modeling Language) has been standardized by the OMG and is commonly used as a metamodel in MDE community. UML is understood by the overall software engineering community and is taught in all computer sciences university. However, UML is not directly exploitable by MDE to precisely model a software engineering system because of its lack of semantics. For a specific context, UML is used with a *profile*, an extension composed of *stereotypes* to specialize UML classes and of *tagged values* to add attributes to these classes. Several standard profiles aid to model designs in specific fields. The MARTE [39] profile which stands for Modeling and Analysis of Real-Time and Embedded systems. Among other things, MARTE provides mechanisms to efficiently represent massively parallel ISP applications. Its standardization process by the OMG is under finalization. In the context of high level synthesis, UML is suitable to graphically model applications and to specify the metamodels.

3 Methodology for a HLS Flow

According to the preliminary conclusions sketched in the previous section, we developed a fully automatized HLS flow which is entirely based on MDE. The flow is dedicated to massively parallel ISP applications and, thus, has to assume some specific requirements: hierarchy, factorized expression of regular tasks in data parallelism, multidimensional arrays and complex data dependencies.

This flow offers relevant advantages to users *and* designers of the flow thanks to the MDE methodology. Models, metamodels and model transformations are

[1] `http://www.omg.org/`

[2] `http://smartqvt.elibel.tm.fr/`

[3] The very recent QVTO tool partially supports the QVT standard. We are currently rewriting our rules with QVTO tool in order to test its effectiveness and to check that the supported part of the standard is large enough for our requirement. However, this implementation of the rule does not have any effect on the work presented in this paper.

omnipresent: models are manipulated by the users, they respects the metamodels defined by the designers of the flow and are transformed by model transformations, also defined by the designers. Figure 4 illustrates the HLS flow, left-hand side represents the flow from a user point of view while the right-hand side represents the flow as considered by a designer.

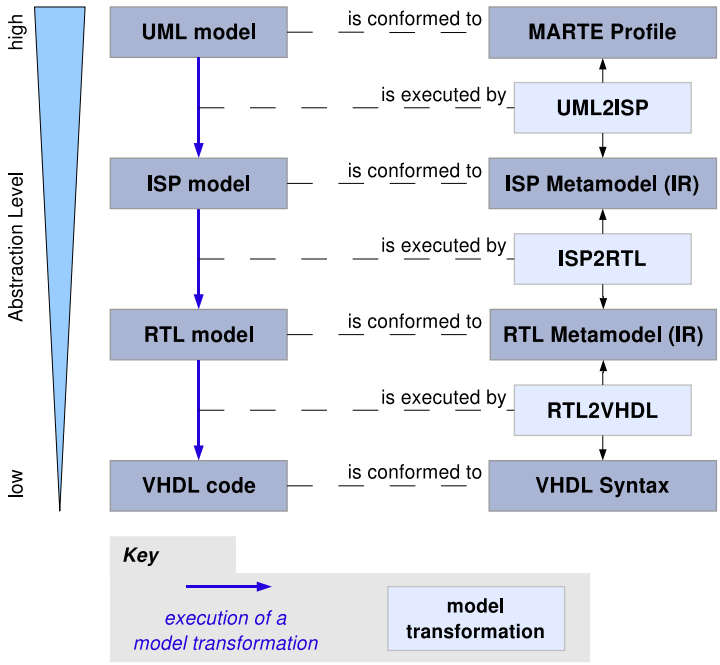


Figure 4: Our HLS flow for ISP applications.

A UML model corresponds to the modeling of an ISP application, this is the only specification provided by the user: everything else is automatized. The applications are entirely defined at a high abstraction level, independently from any implementation detail. A UML model is transformed into an ISP application model thanks to the UML2ISP transformation. The ISP metamodel corresponds to an IR in the HLS flow, it is also dedicated to the high level description of ISP applications. The ISP2RTL model transformation refines ISP application models into RTL models. A RTL model corresponds to a low abstraction level of an hardware accelerator able to execute the corresponding ISP application. A RTL model provides a precise estimation of the resources required for the resulting design implementation. An exploration process (not illustrated in the figure but detailed later on) is performed according to these estimations. The RTL2VHDL model transformation ensures the generation of the VHDL code corresponding to the hardware accelerator described in a RTL model. Usual Electronic Design Automation (EDA) tools are used to synthesize the resulting VHDL code onto FPGA or either ASIC. The subsequent parts of this section details this flow.

3.1 Specification at a High Abstraction Level

3.1.1 UML Model

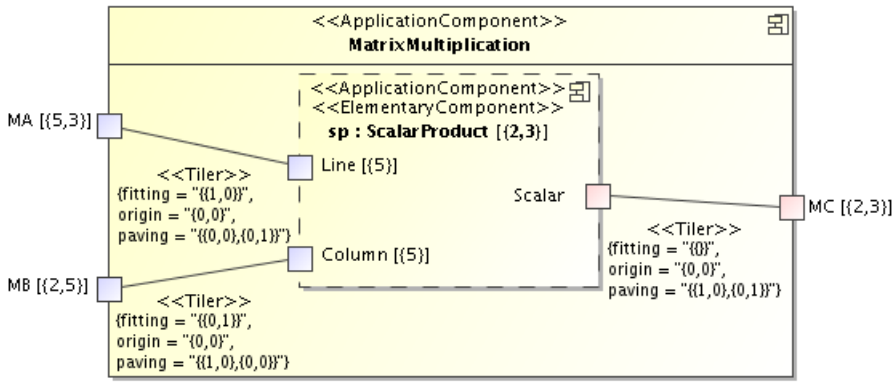


Figure 5: UML model of the matrix multiplication academic example.

Application modeling relies on the part of the MARTE profile dedicated to the factorized expression of the parallelism, which is useful to specify the ISP applications. Such applications are data flow oriented. Data are manipulated in the form of multidimensional arrays. The absence of restriction on the number of dimensions in data arrays allows to represent data as often manipulated in ISP applications. For instance, video processing applications handle two spacial and one temporal dimension. Sonar chain is another kind of application, which handles spacial, temporal and frequency dimensions. MARTE allows to models such applications.

The MARTE profile provides powerful mechanisms to specify the data dependencies. For task parallelism, data dependencies represent data array transactions from one task to another task thanks to *simple* connectors. Each task is represented as a UML component, as illustrated by Figure 5. This figure represents the modeling of a matrix multiplication academic example which multiply matrix MA and MB in order to produce a matrix MC. MatrixMultiplication 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, the ports correspond to the matrix and the dimension of each port correspond to the dimension of the corresponding matrix: 5×3 for MA, 2×5 for MB and 2×3 for MC. In the MARTE profile, a repetition space on a task expresses data parallelism. On Figure 5, the multiplicity {2,3} of the component instance sp (*i.e.* sp is an instance of the ScalarProduct task) represents such a data parallel task.

In this example, ScalarProduct is an elementary task (the elementary tasks are stereotyped ElementaryComponent). An elementary task does not include any hierarchical task (*i.e.* it corresponds to a leaf of an application model). The behavior of an elementary task is provided according to additional information included in the deployment, as described later on. In the matrix multiplication example, the task sp consumes two input *patterns* (Line and Column) and produces

Figure 6: The data dependencies expressed by the tilers of the MatrixMultiplication task.

an output pattern (Scalar) which corresponds to the result of a scalar product of a line and a column.

Each iteration in the repetition space consumes and produces patterns. The pattern construction relies on the data dependencies expressed thanks to the *tiler* connectors. A tiler allows to model data dependencies that links a M-dimension data array to a N-dimension pattern. These data dependencies are not limited to compact and parallel to axis patterns. In usual HLS tools, data dependencies are expressed within indexes. Indexes are tedious to manipulate and error prone when directly provided by users: their complexity dramatically increases with the number of dimensions and the shape of the pattern. Tilers do not cause this inconvenience. Three tilers are represented on the figure, they are identified thanks to the Tiler stereotype. Origin, Paving and Fitting are attributes of a tiler, they express data dependencies. Boulet [6] provides a formal description of tilers, details their construction and illustrates them according to some relevant examples. He also formally demonstrates that a computation described with such data dependencies is deterministic.

Figure 6 represents the data dependencies expressed with the tilers used in the matrix multiplication example (Figure 5). The left-hand side of Figure 6 represents the tiler that link 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, while considering the first iteration on the repetition space $r = \binom{0}{0}$, the first line and the first column of the data array MA and MB are read[4]. This line and this column are used by the first iteration of the task sp in order to produce the first data (*i.e.* the data on position $(0, 0)$) in the output data array MC). Regarding to the iteration $r = \binom{1}{0}$, the first line of MA is read again while the second column of MB is used[5]. The data on position $(1, 0)$ in the MC is computed by iterating around the overall repetition space, the overall output data array MC is produced.

This example demonstrates the effectiveness of the MARTE profile for representing data parallel applications because the high abstraction level modeling is completely independent from its low level execution. In order to transform such high abstraction level models into low implementation level models, very detailed deployment information are provided. In particular, each elementary component is linked to an existing code. For this purpose, a *deployment* profile has been in introduced in [2]. The aim of this methodology is to facilitate Intellectual Property (IP) reuse. Indeed, Piel *et.al* [2] introduce the concepts of AbstractImplementation and Implementation in their UML profile, they respectively expresses a functionality and a given implementation. Each Implementation is

[4]The line and the column are constructed thanks to the fitting field.
[5]The shift of the line and the column are constructed thanks to the paving.

linked to a file containing the code of the targeted IP. Using the ImplementedBy dependency, designers select the adequate IP for each elementary task.

Figure 7 illustrates the deployment of the ScalarProduct elementary task onto the ScalarProduct_VHDL IP (this is realized with the ImplementedBy dependency). The IP code itself is furnished by the CodeFile artifact. The ports of the elementary task are also deployed, ensuring the right deployment of the task ports onto the IP ports. ScalarProduct_C is another IP written in C language which can be used for an execution onto a processor-based architectures for instance [2]. Its functionality is equivalent of the VHDL one, this explains the gather of both IP in the AbstractScalarProduct abstract component.

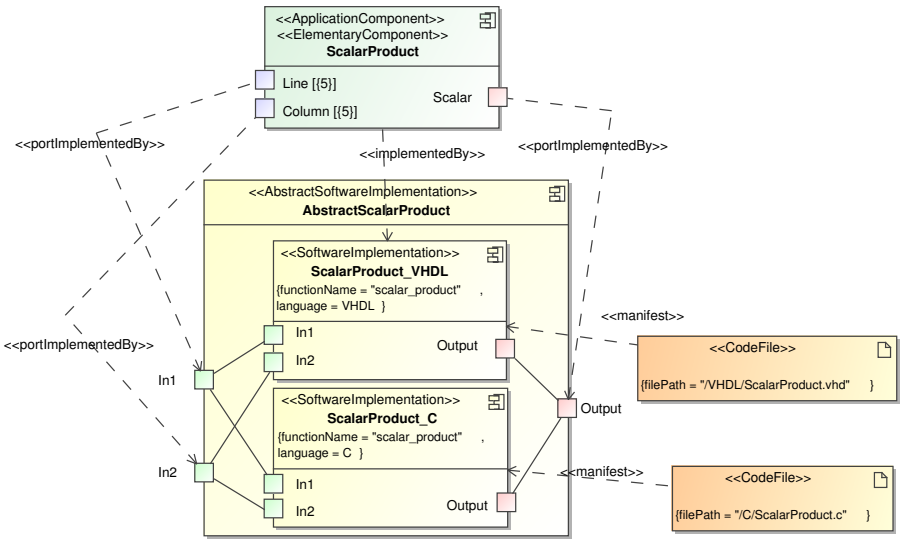


Figure 7: Deployment of the ScalarProduct elementary task onto the ScalarProduct_VHDL IP.

In this methodology, great care was taken to allow usage of IP libraries (models which contain a set of existing IPs) and to keep the application model independent from the implementation target. Using this methodology, we are able to provide additional information that will ensure the execution of the (automatically generated) low level implementation, thanks to existing IPs.

3.1.2 ISP Model and UML2ISP

The first IR is the ISP metamodel. Generally speaking, it corresponds to the interesting subset of MARTE dedicated to the description of ISP applications. UML2ISP ensures the generation of an ISP model from a UML model. The gap between the MARTE profile and the ISP metamodel is little enough to rely on relatively simple transformation rules. These rules identify the UML concepts and MARTE stereotypes used in a UML model in order to generate the corresponding concepts in an ISP model.

Figure 8 represents the ISP model corresponding to the internal representation of the matrix multiplication UML model. This ISP model is automatically

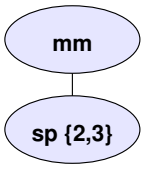


Figure 8: ISP model generated from the UML model of the matrix multiplication application.

generated by the UML2ISP model transformation. Such representation does not provide all details of the model but focuses on the hierarchy and the available data parallelism. The mm ellipse (the task MatrixMultiplication) corresponds to the top hierarchical task. mm instantiates the task sp according to the {2,3} repetition space.

Such high level models are independent from any technological implementation. ISP2RTL ensures the generation of low level implementations from such high level models.

3.2 Implementation at a Low Level

3.2.1 RTL Model

The RTL metamodel is an IR, which gathers the set of concepts used to describe hardware accelerators. Such hardware accelerators can execute the targeted ISP applications according to a specific execution model[6]. This execution model is data flow oriented and handle, among others, hierarchy, multidimensional data dependencies, data parallelism and task parallelism. The following provides a brief overview of the RTL metamodel, additional and more detailed information are provided in [27].

Components In hardware design, a component represents a hierarchy level which is often linked to a specific functionality. An instance of component allows to use such functionality in another component, which can itself be instantiated, etc. This component based approach and this instantiation mechanism are identify in the RTL metamodel in order to model hierarchical and well structured hardware accelerators. The communications between components and component instances are assumed thanks to interfaces, which are composed of ports. A port can be input or output: it can receive or send data. The overall mechanism is described in the RTL metamodel [27], a subset of this mechanism is introduced in the following.

The right-hand side of Figure 9 illustrates a set of concepts in the RTL metamodel used to model the components in RTL models. The concept Component contains one InputPort clock (relation clock), one InputPort reset and several ports. The concept Port is specialized into InputPort or OutputPort. Moreover, the dim relation allows to specify the Shape of a port and the type relation allows to

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

reference a DataType. This part of the RTL metamodel is used to described the components in RTL models.

The left-hand side of Figure 9 illustrates a component in a RTL model, which is issued from the previously illustrated UML model (ISP2RTL ensures this model transformation, it is described later on). The MatrixMultiplication component owns the input ports clock, reset, MA and MB, and the output port MC. This representation is very closed to the one manipulated in UML model because the both level use component based approach. This figure represents the hardware description of the MatrixMultiplication component included in an hardware accelerator.

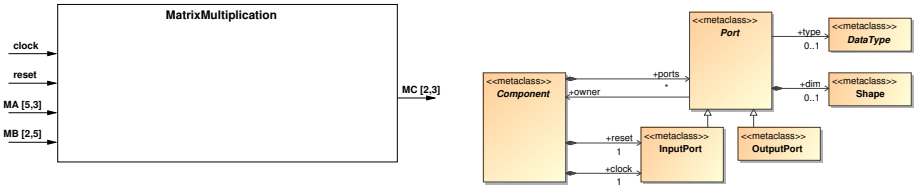


Figure 9: Excerpt of the RTL metamodel (on the right-hand side) and a component in a RTL model (on the left-hand side).

Data Dependencies A set of concepts in the RTL metamodel is dedicated to the description of hardware data paths. A data path corresponds to the hardware implementation of a data dependency in a given hardware accelerator. In task parallelism, data array dependencies allow to shift data arrays between tasks. Such simple data dependencies are implemented in hardware with connectors or buffers for instance. In data parallelism, data dependencies potentially express complex relations between a data array and data in patterns, or between data in patterns and a data array. For this purpose, we introduce the concepts of InputTiler and OutputTiler in the RTL metamodel. These concepts are elementary components that enable to implement the data dependencies expressed in high level models.

In the RTL metamodel, the data dependencies (specified with tilers in high abstraction levels models) are described by the concepts of connector, shift register, latch, etc. Combining the use of these concepts, it is feasible to design customized data paths for multidimensional data array dependencies. Both spacial and temporal data dependencies are managed with these concepts.

The right-hand side of Figure 10 illustrates a subset of the RTL metamodel which allow to model data paths hardware implementation. The left-hand side of this figure illustrates a RTL model composed of three data paths, which are represented by the irregular shapes TA, TB and TC. More precisely, TA,and TB are input tilers because they respectively read the data arrays MA and MB. Symmetrically, TC is an output tiler because it produces the data array MC. The data paths are instantiated in the MatrixMultiplication component, which is modeled with the corresponding RTL metamodel subset (previously detailed). In RTL models, the data paths instantiation in component is feasible thanks to some composition and reference relations in the RTL metamodel.

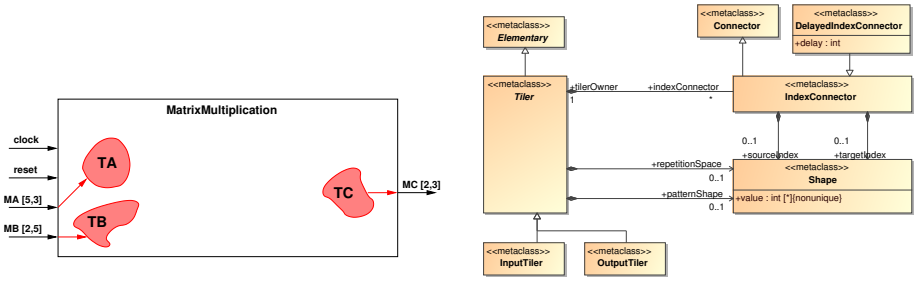


Figure 10: The right-hand side of this figure represent a subset of the RTL metamodel dedicated to the data path modeling. The left-hand side represents three customs data paths which are instantiated in the MatrixMultiplication component.

Execution of the Data Parallelism The aim of such hardware designs is to accelerate highly time and resource consuming applications by efficiently executing (*i.e.* in parallel) the data parallel tasks. Indeed, the execution of the data parallelism is a key point for the design of hardware accelerators, and therefore for the construction of the RTL metamodel. A major difficulty encountered for the development of the RTL metamodel subset dedicated to the data parallelism execution comes from its high level specification. Indeed, no implementation details are furnished in high level models, the execution of such high level model depends on the HLS flow. For instance, the multiplication matrix academic example described above can be executed in different manners (this is possible because there is no data dependencies between iterations in the repetition space of the task in this example):

- **case 1**: a single computing unit executes the overall iterations in the repetition space of the ScalarProduct task. The resulting execution, so called *sequential execution*, reduces the consumed resources (*i.e.* FPGA resources for instance) but is not efficient considering the execution time since the data parallelism is sequentially executed;
- **case 2**: as much computing units as iterations in the repetition space perform the ScalarProduct tasks. The resulting execution, so called *parallel execution*, is the most powerful but, compared to the sequential execution, consumes much more resources;
- **case 3**: alternative executions are feasible, they can be resumed as a *mix parallel/sequential execution*. Such combination provides flexibility and is suitable for finding good compromise between performance and resources usage.

The following discuss on our strategy for executing the data parallelism and the implementation issues. The sequential and the parallel execution correspond to the extremes implementation of a data parallel task: one consumes few resources but is not effective while the other one is very effective but consumes a large amount of resources. The choice between a solution or another thus depends on a single criteria: the execution performance or the area cost. In high level synthesis, the selected solution generally satisfy a set of criteria, like the

real time constraints and-or the area cost [25]. The mixed parallel/sequential execution thus seems to be an interesting alternative for finding a satisfactory implementation (*i.e.* which satisfy several constraints). Its major drawback is the necessity for providing many implementation choices during the creation of an RTL model (the choices specify how is executed the iterations in a repetition space). Such implementation choices concern the ISP2RTL transformation model and has a negative impact of the RTL metamodel itself. Indeed, this latter shall supports a generic execution of the data parallelism tasks, which is much more complicated than the single sequential and parallel executions.

In order to provide several executions of a given data parallel task, we use the loop transformations defined in [7]. In their works, loop transformations are developed for the interesting subset of the MARTE standard dedicated to the modeling of intensive signal processing applications. Technically, the loop transformations are realized onto ISP models. The functionality of these loop transformations are similar to those existing in literature, with the great advantage they can be directly used in our HLS flow. The aim of these loop transformations is to modify, create or delete hierarchy of the applications[7]. For instance, regarding to the matrix multiplication example, these loop transformations can create a hierarchical task "between" MatrixMultiplication and ScalarProduct and move a part of the repetition space around the instance of ScalarProduct onto the newly created task. As result, the application model contains two data parallel tasks (against a single one for the initial solution), each one can be executed independently from the other (*i.e.* both parallel, both sequential, parallel and sequential, sequential and parallel). According to another loop transformation, three data parallel tasks can be executed independently from each other: the number of potential combinations increases. The functionality provided by these loop transformations perfectly answer our flexibility objectives without any drawback on the RTL metamodel. Indeed, the fact that the RTL metamodel does not support any mixed sequential/parallel executions for a given component is fully compensated by the loop transformations.

The RTL metamodel subset dedicated to the modeling of the data parallelism execution is partially illustrated on the right-hand side of Figure 11. It gathers the set of concepts used to represent Repetitive components, which are components able to execute in parallel the data parallelism. The repeated task is instantiated as much as there are iterations in the repetition space. For instance, the repetition space around the ScalarProduct task is {2,3} in the corresponding UML model. For a parallel execution, the task is instantiated 2×3 times, as illustrated in the top left-hand side of Figure 11. The six filled boxes represent the six instances of the scalar product task. Depending on their iteration number, each instance computes a given line-column scalar product of the matrix multiplication. For this purpose, each instance is connected to the right data provided and furnished by/to the customized data paths.

The SequentialRepetitive concept of the RTL metamodel represents the components which sequentially execute a data parallel task. As opposed to a Repetitive component, a SequentialRepetitive component instantiates only once the repeated task, as illustrated on the bottom left-hand side of Figure 11. The overall computation is assumed thanks to a controller (represented with a lozenge),

[7] Additional details about these loop transformations and their usage in our HLS flow is provided later on.

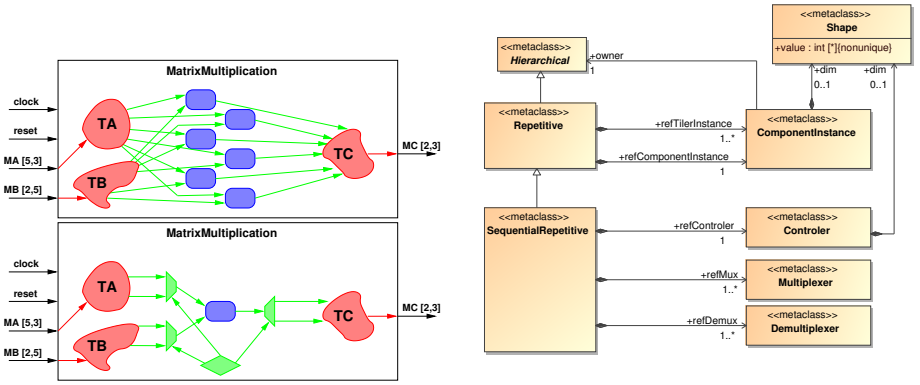


Figure 11: Subset of the RTL metamodel dedicated to the specification of the data parallelism execution (right). Two RTL models executing in parallel and sequential the data parallelism available in the matrix multiplication application (left).

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 onto 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. Symmetrically, the demultiplexers send the right data to the output tiler.

Hardware-Software Partitioning The RTL metamodel also assumes a *software execution*. Such execution is used in heterogeneous contexts including both processors and hardware accelerators. In this specific context, the hardware accelerators are slaves of the processors. The processors can decide to use an hardware accelerator by sending data, launching execution and retrieving processed data. Therefore, a data parallel task can be executed by an hardware accelerator which is controlled by a processor (the iterations on the repetition space is assumed by the processor). This execution is generally used when the grain of an application task level corresponds to a grain that embedded system designers use to handle with processors-based architectures (*i.e.* coarse grain). Fine grain computation are assumed by hardware accelerators, while coarse grain ones are managed by processors. The software execution is under development in the RTL metamodel but additional information on processor based architecture execution are provided in [18].

FPGA Implementation The execution time of an hardware accelerator is deduced from the repetition space of a data parallel tasks. However, the area cost (*i.e* the amount of hardware resources required for an implementation) is more difficult to evaluate since it depends on many factors. Indeed, the amount of resources consumed for a parallel execution depends on the repeated task, the generated data paths and the data type. Concerning the sequential execution, it becomes necessary to take into account the controller, the multiplexers, the

demultiplexers and the latches. Knowing such characteristic before an implementation accelerates the design space exploration and allows to automate it. For this purpose, we introduce some concepts in the RTL metamodel that help us for characterizing the hardware accelerators.

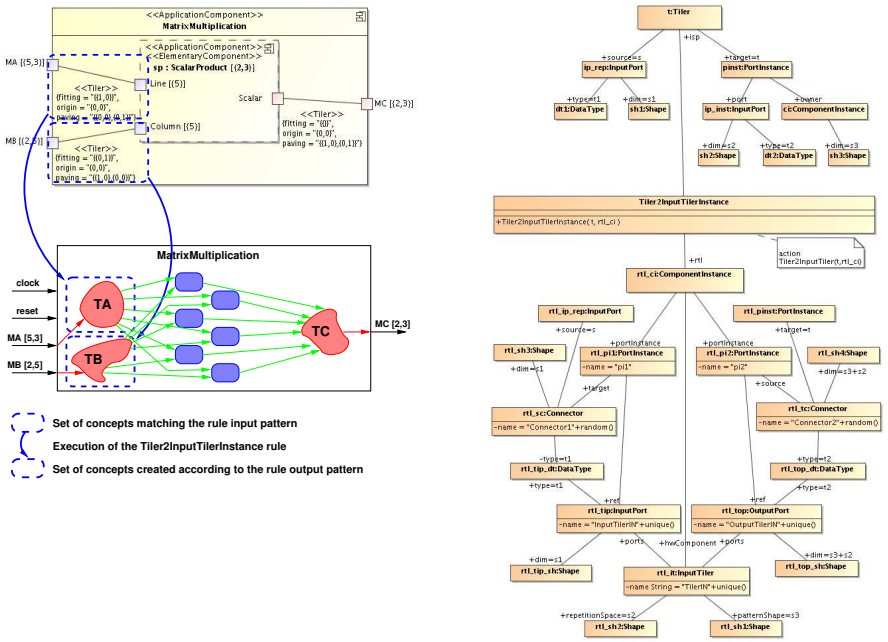


Figure 12: Graphical representation of the rule (right) which transforms the tilers into customized data paths (left).

In order to estimate the FPGA resources used for the FPGA implementations of hardware accelerators, the RTL metamodel is enriched by a set of concepts. These concepts helps for characterizing hardware elements described in RTL models, such as the controllers, the multiplexers or the used IP. Each concept in the RTL metamodel is thus associated to a basic mathematical expression which, depending on criteria, provides the quantity of basic FPGA resources used for FPGA synthesis [28]. For instance, an interesting criteria is the bit-length of the data handle by a multiplexer or the repetition space iterated by a controller. The basic resources used for the characterization are the basic resources contained in FPGA. The basic resources of the Stratix FPGA family are ALUT (Adaptative Look Up Table), DSP multiplier and memory bank. Furthermore, the exact details of FPGA are not taken into account since our objective is to guide FPGA synthesis tools (e.g. Quartus, ISE), but not to replace them [29].

Our estimation process is fully automated, it acts like a query on RTL model. This query thus provides an overview to the HLS tool of the FPGA implementation results. Such estimation results is used by the exploration process described later on.

3.2.2 ISP2RTL Transformation

The transformation from an ISP model towards an RTL model is performed by the ISP2RTL transformation, which adds implementations details to the high abstraction model to generate the low level implementation model. ISP2RTL is decomposed into rules. For instance, a set of rules transforms the tasks (ISP model) into hardware components (RTL model), the basic rule sketched Figure 2 assumes the creation of the clock and reset input ports of the hardware components. The rule input pattern is composed of a single concept and the rule output pattern contains four concepts. This rule is quite simple because the concepts manipulated in the ISP and those manipulated in the RTL metamodels are very closed each others.

While some rules are very simple (such as the so called one-to-one rules), some others are much more complied. This occurs when there are large differences between the source and the target metamodels. For instance, the data dependencies expressed in ISP models are factorized (thanks to the tilers) whereas customized data paths are manipulated in order to achieve high throughput in RTL models. Therefore, the creation of a customized data paths starting from a pure data dependencies expression relies on a quite complex set of rules. In this set of rules, the Tiler2InputTilerInstance rule transforms tilers into instances of a customized data path (the generation of these data paths is assumed by others rules). This rule is illustrated on the right-hand side of Figure 12. The top right-hand side represents the rule input pattern, it identifies the source and the target of the tilers connectors and also identifies the shape of the source port, the repetition space of the repeated task, etc. In a given ISP model, the execution of the rule is triggered each time a part of the model perfectly matches the rule input pattern. For instance, the ISP model illustrated on the top left-hand side of Figure 12 owns two inputs tilers that match the rule input pattern [8]. The corresponding subsets of the model are identified with the dashed shapes. Two executions of the Tiler2InputTilerInstance rule are thus performed. When triggered, this rule creates, a set of concepts in the RTL model according to the rule output pattern illustrated on the bottom right-hand side. This rule output pattern corresponds to the instantiation of a data path component. Such data path has one input port and one output port that are similarly managed in the rule, this explains the vertical symmetry in the rule output pattern.

On the bottom left-hand side of the RTL model, the shapes TA and TB are the results of the transformation rule execution. They respectively correspond to the instantiation of the data paths that produce the line elements and the column elements in the matrix multiplication example. As previously mentioned, the data path themselves are created by another rule, this rule requires additional information such as the origin, the paving and the fitting attributes. This rule is called from the Tiler2InputTilerInstance rule, which is itself called from others rules, etc. This decomposition into rules facilitates the comprehension, the maintainability and the extension of the transformations. In the same manner, each rule clearly identifies its rule input and output patterns, this drastically facilitates the extensions, modifications and maintainability of each rule. Combining these advantages ensures the high level to low level transformation process to be entirely handled by the tools designers.

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

3.2.3 RTL2VHDL Transformation

The RTL metamodel is independent from any HDL syntax, but is low level enough to allow their code generation. In MDE, a code generation is a model to text transformation. Such a transformation defines the relations between a concept or a set of concepts with HDL syntax. The VHDL code generation from the RTL metamodel is performed within templates that navigate into a RTL model in order to find concepts they are associated with. The templates print a VHDL syntax in files associated with elements of a RTL model. The top of Figure 13 represents the template associated to the Component concept in the RTL metamodel. The bottom of the figure represents excerpts of the generate code for the MatrixMultiplication component previously modeled. Special attention was given for the development of the RTL2VHDL transformation in order to make it suitable for ISP applications. Indeed, multidimensional data arrays are supported and data parallelism is still factorized.

```
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()%>;
```

```
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;
```

Figure 13: Excerpts of the RTL2VHDL transformation (top) and of a generated VHDL code (bottom).

The generated code can be directly synthesized (for instance on FPGA) according to usual synthesis tools. Figure 14 illustrates the synthesis result for the matrix multiplication example. The two left-hand side box create the patterns (*i.e.* the lines and the columns) starting from the arrays (*i.e.* the matrix). Symmetrically, the right-hand side box creates the matrix according to the data computed by the scalar product tasks. In the center of the figure, the six boxes correspond to these tasks. Since the line and the column are generated in the same time, the tasks are executed in parallel.

4 Design Space Exploration

A major goal of our HLS flow is to rapidly design an hardware accelerator that meets its requirements, in particular those related to performances and area costs. This goal is achieved by considering some basic notions that allow us to modify high level models and to estimate low levels models. Based on these basic notions, we defined a strategy that allows a rapid exploration of the design space.

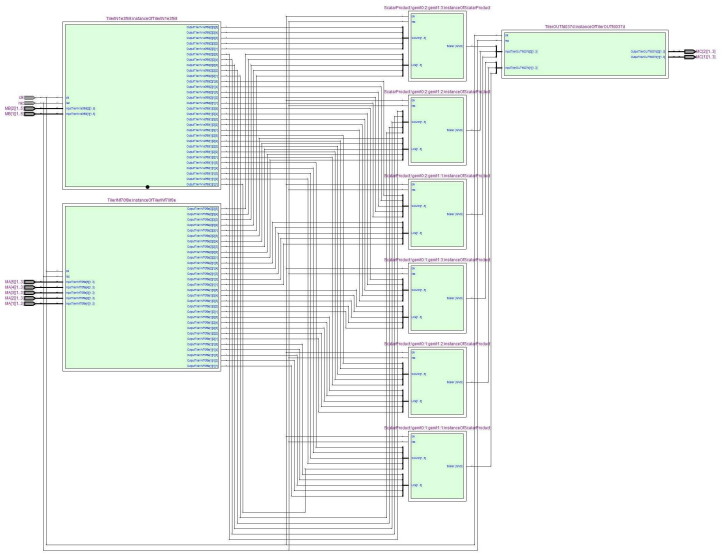


Figure 14: View of the synthesized hardware accelerator.

4.1 Basic Notions

The design space exploration relies on modifications of the high level specification according to characteristics available at the implementation level. The modification of high level models are performed through the loop transformations detailed below.

4.1.1 The Available Loop Transformations

Our HLS flow uses the loop transformation functions, or refactoring functions, proposed in [7]. Since they are fully compliant with MARTE, these functions are directly applied on ISP models in order to produce another ISP model. Successive loop transformations can therefore be applied on a given ISP model, the result is always conformed to the ISP metamodel.

The loop transformations can modify, create or delete the hierarchy according to the data parallelism. Moving the data parallelism into the tasks allows to structure an application according to specifics requirements: while a hierarchical decomposition is suitable for complexes applications, abyssal hierarchy becomes useless for trivial ones. Figure 15 illustrates the impacts of the loop transformations when applied on the ISP model generated from the UML model of the matrix multiplication (left-hand side on the figure). The following loop transformations allow one to modify as desired such ISP model:

- The *Tiling* loop transformation creates, for the matrix multiplication, the hierarchical task mm as illustrated on the top and the bottom of the figure. Depending on the parameters provided this function, mm assumes a part of the repetition space of the initial sp repetition space: {2}, {3} or {2,3}.
- *Change Paving* moves the data parallelism through the hierarchy without modification of the hierarchy.

- *Collapse* deletes the hierarchy. As result, the available data parallelism is concentrated on a single hierarchical level, as illustrated in Figure 15.
- *Fusion* extracts the potential data parallelism available in two successive tasks in a graph of tasks. This allows to factorize the data parallelism expression and to reduce the size of the data arrays transferred between tasks. This refactoring function is not illustrated here.

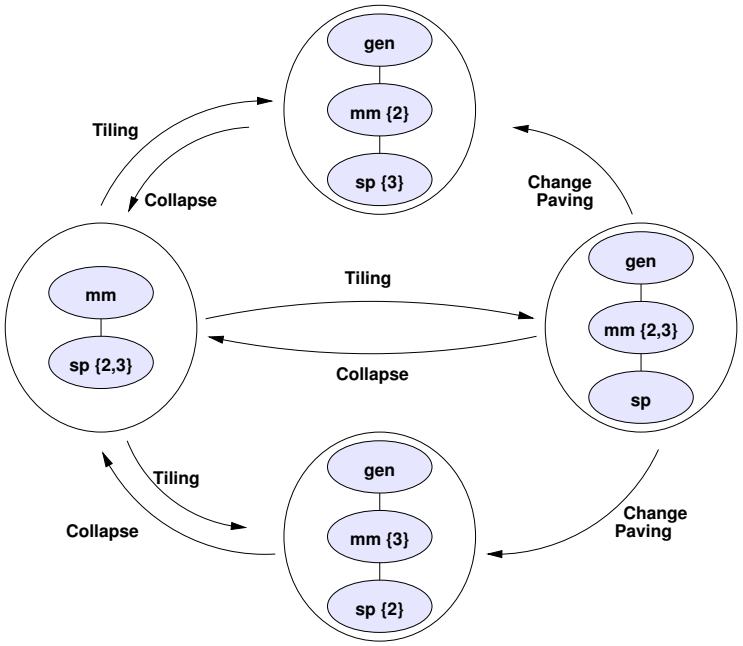


Figure 15: Impacts of refactoring functions on the matrix multiplication application.

4.1.2 The Data Parallelism Execution Partitioning

In a given ISP model, depending on the execution of the available data parallelism, different hardware accelerators may be generated. The data parallel tasks can be executed in three different ways: in software with processor, in sequential or in parallel inside accelerator. Each hierarchical task is thus associated with a given execution, with the following constraints: the top level tasks are executed in software, the lower ones are sequentially executed inside the accelerator, and the lowest ones are executed in parallel inside the accelerator.

Figure 16 represents some of the possible executions of data parallelism available in the matrix multiplication application. The left-most *configuration* represents a software execution on processor. This configuration does not benefit of the potential improvement offered by hardware accelerators, as opposed to the right-most configuration. Indeed, in this latter configuration, the overall data parallelism is executed in parallel inside the hardware accelerator. According to the configurations illustrated in the center of the figure, the data parallelism execution is partially accelerated thanks to the hardware accelerator. Compared

to the most-left and the most-right configurations, these intermediate configurations are expected to be respectively more effective and less effective. However, they are also supposed to respectively consume more hardware resources and less resources. Furthermore, other configurations not illustrated in this figure exist. They provide other alternatives to the implementation of this application

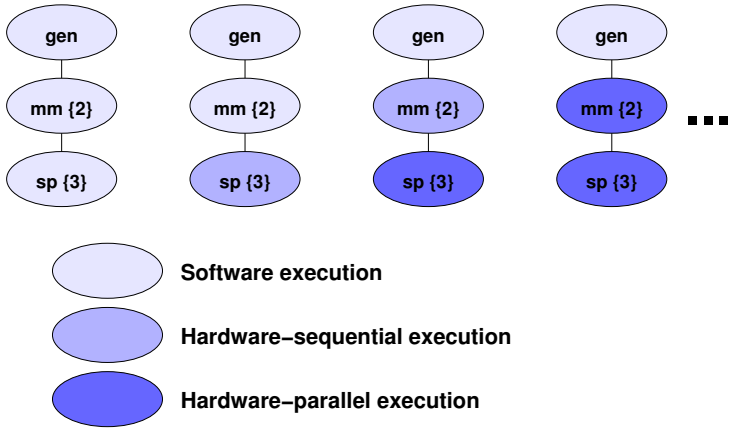


Figure 16: Possible executions of the data parallelism available in the matrix multiplication.

Since the loop transformations described above suitably modify the hierarchy in ISP models, they can be well combined to the data parallelism execution. The set of possible combinations defines the design space. This design space and covers solutions that vary from those which optimize the execution performances to those which minimize the area costs (*i.e.* the amount of consumed hardware resources) of the accelerator. In order to explore this design space, we introduce the PARALLELIZE and the SEQUENTIALIZE functions. They abstract the combination of the loop transformations and the data parallelism executions in order to focus on their impacts on the performance and area costs of the hardware accelerators:

- In order to increase the execution performances provided by a hardware accelerator, the PARALLELIZE function increases the number of iterations in the repetition spaces that are executed in parallel. For this purpose, *Change Paving* or *tiling* are applied onto tasks sequentially executed, moving this sequential execution into a parallel one.
- As opposed to the PARALLELIZE function, the SEQUENTIALIZE function increases the amount of iterations that are executed in sequential. For this purpose, COLLAPSE and TILING are successively applied on a ISP model. As consequence, the performances of the hardware accelerator are reduced bu it is less expensive in terms of hardware resources.

4.1.3 Performances Evaluation

In order accelerate the design space exploration, it is useful to quickly evaluate the effectiveness the hardware accelerators. Instance of interesting properties are the amount of consumed hardware resources (*i.e.* the area cost) and the execution time (in clock cycles). Our HLS flow estimates the performances of

hardware accelerators. The estimation process is performed on RTL models, which provide additional implementation details compared to ISP models since the RTL level is very close to the final implementation.

Table 1 summarizes the possible configurations and the characteristics of the generated hardware accelerators. The configurations have been produced thanks to the PARALLELIZE and the SEQUENTIALIZE functions described above. They are classified according to the selected hardware software partitioning, which determine the execution time of the software part, as described later.

In the first four configurations (those from the HW-SW 1 partitioning), the overall data parallelism is executed in hardware. In the first configuration, it is fully executed in parallel. According to the estimation, the corresponding hardware accelerator consumes 13548 hardware resources (an ALUT corresponds to an atomic configurable resource in FPGAs), while 13583 are necessary for the "real" implementation. To obtain this last result, the VHDL code corresponding to the hardware accelerator was generated with our HLS flow, and then was synthesized onto FPGA with a commercial synthesis tool. Our evaluation process approximately 0.26% under-estimates the required hardware resources for this configuration. This estimation is instantly provided while several minutes are necessary for a synthesis with commercial tools. The estimation process thus drastically increases the number of testable configurations in a given time. Moreover, since the estimation results are available in the HLS flow, they can be suitably used for automating the exploration.

In the first configuration, 1 cycle is necessary since the overall data parallelism is executed in parallel. In the second one, this data parallelism is partially executed in sequential, increasing the execution time to 2 cycles. 6 cycles are necessary for the fourth configuration, which thus corresponds the less powerful hardware accelerator for the corresponding hardware software partitioning. This table shows that the amount of used hardware resources diminishes with the execution performance. Therefore, the most powerful accelerator is also the most area consuming one in this example.

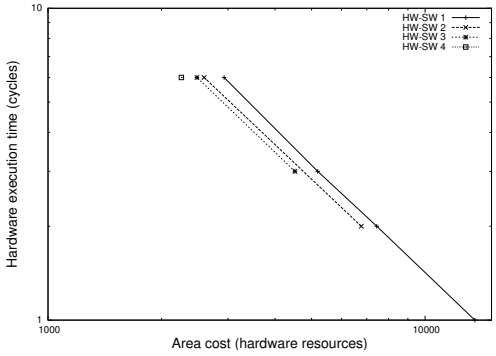
The software part manages the usage of the hardware accelerator for executing the matrix multiplication. It depends on the processor, the bus bandwidth, the memory latencies, etc. Currently, they can not be estimated in our HLS flow, but this is part of future work [18]. The software execution time is extracted from observations of manual implementations. Depending on the processor and the others hardware resources, they varies from 10 to 100 cycles for the four first configurations. This execution time increases the data parallelism to execute in software.

The execution performances and the estimated amount of hardware resources required for each configuration are summarized in Figure 17. In theses figures, each curve corresponds to a given hardware-software partitioning, and each point in a curve corresponds to a given sequential-parallel partitioning in the hardware execution. Each point thus corresponds to a given configuration. Figure 17(a) summarizes the hardware execution time of the generated hardware accelerators. The software execution time is taken into account in Figure 17(b) and Figure 17(c).

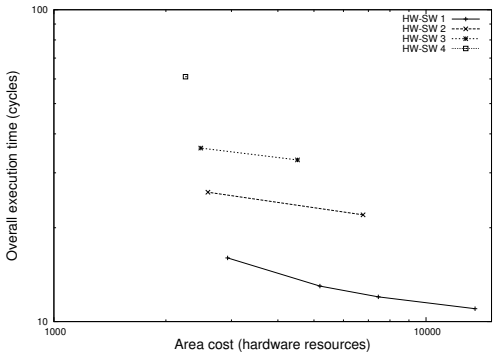
The most-right curve in Figure 17(a) corresponds to the configurations in which the overall data parallelism is executed in the hardware accelerator. The execution performances vary from 1 to 6 cycles and the area cost respectively vary from 14000 to 2900 hardware resources. In the others curves, the execution

Table 1: Characteristics of hardware accelerators generated for the matrix multiplication.

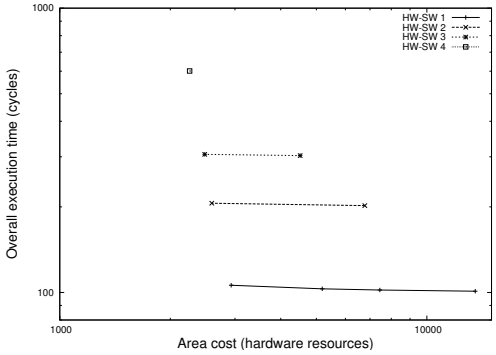
Hardware-software partitioning		HW-SW 1				HW-SW 2		HW-SW 3		HW-SW 4
Configuration										
Hardware resources (ALUT)	estimation	13548	7446	5188	2930	6774	2594	4516	2482	2258
	analysis	13583	7154	4898	2544	6768	2526	4512	2489	2258
Relative error of the estimation (%)		-0.26	4.09	5.92	15.17	0.09	2.69	0.09	-0.29	0
Execution time (cycles)	hardware	1	2	3	6	2	6	3	6	6
	software	$10 \sim 100$				$20 \sim 200$		$30 \sim 300$		$60 \sim 600$



(a) Execution time of the hardware part.



(b) Overall execution time with a high performance software execution.



(c) Overall execution time with a non efficient software execution.

Figure 17: Characteristics of the hardware accelerators executing the matrix multiplication application.

of the data parallelism is partially managed in software, reducing the area cost of the hardware accelerator. The overall execution time (*i.e.* including the hardware and the software part) is represented in Figure 17(b) and Figure 17(c), which respectively consider an efficient and a non-efficient software execution. These figures illustrate the impact of the software execution on the effectiveness

and the area cost of the hardware accelerator. They also highlight the diversity of solutions offered for the matrix multiplication application. While it was possible to explore the overall design space for such quite simple application, it is necessary to automate this exploration for "real-life" applications. The following deals with the automatic exploration process of the design space and its integration into our HLS flow.

4.2 Design Space Exploration Strategies

Depending on the application to be executed, the set of possible configurations provided by our HLS flow may be large. Manually testing each configuration of this design space is thus a time consuming task. This task becomes non-efficient when most of the explored solutions do not satisfy criteria, such as the execution performance and the area cost. In order to accelerate the design space exploration, we define the global and the local strategies. The global strategy allows one to widely explore the design space modifying the hardware software partitioning (*i.e* to move from a curve to another curve in Figure 17). The local strategy searches for a satisfying solution according to a given hardware software partitioning (*i.e* a solution in a given curve).

4.2.1 Global Strategy

The aim of the global strategy is to modify the hardware software partitioning in order to widely explore the design space. Since the interaction between hardware accelerators and processors is still work in progress in our HLS flow, the hardware software partitioning is assumed by the HLS flow users [18]. However, this exploration is facilitated by the estimation process performed on the generated RTL models.

The exploration with the global strategy starts with a specific configuration in which the overall data parallelism is executed in software. For this purpose, an initialization process applies the *Collapse* and the *Fusion* loop transformations onto the initial ISP model. *Fusion* extracts the data parallelism available in the tasks graphs, *Collapse* deletes the hierarchy and gathers the data parallelism into a single data parallel task. Finally, the initialization process specifies a software execution of this task. The resulting ISP model corresponds to the last configuration in Table 17. This configuration is supposed to be the less efficient one since the hardware accelerator only computes elementary tasks, without managing the data parallelism.

According to an execution performance criteria, the user modifies the hardware software partitioning, moving from a curve to another. The data parallelism is then moved from a software to a hardware execution in parallel, increasing the execution performance. The reachable solutions correspond to the most right points in the curves. Once a configuration satisfying the execution performance is detected, the local strategy is applied. Indeed, this configuration is supposed to correspond to be the most area consuming solution on a curve since the overall data parallelism managed by the hardware accelerator is executed in parallel.

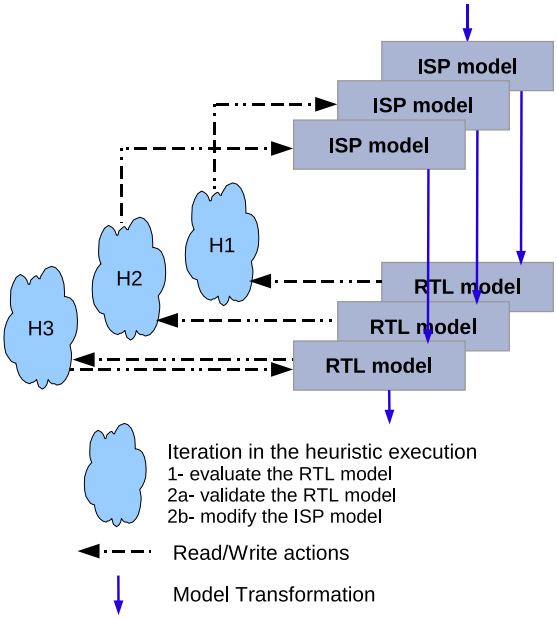


Figure 18: Applying the local strategy to explore the design space.

4.2.2 Local Strategy

The local strategy explores the subset of the design space defined by a given hardware software partitioning. Its aims is to find the less expensive configuration in a curve which still satisfies the execution performance. For this purpose, a heuristic detailed in [27] is executed according to the execution performance and the area cost criteria. The heuristic analyses the estimation results of the generated RTL model and eventually modifies the ISP model using the PARALLELIZE and the SEQUENTIALIZE functions. The ISP2RTL model transformation is launched again, and so on. The RTL2VHDL transformation is launched once a satisfying solution is find. The local design space exploration strategy is sketched in Figure 18. In this example, 3 iterations are necessary to find a satisfying solution. Since other satisfying configuration potentially exist, the selected solution is not necessary the optimal one. Moreover, the convergence time of this heuristic depends on the application itself: a large amount of iterations can be necessary to find an acceptable solution. For this purpose, we plan to use greedy algorithms [25] which allow one to accelerate the convergence time.

These strategies aim at efficiently explore the design space by successively taking into account the execution performance (in the global strategy) and the area cost (in the local strategy). While the global strategy is applied by the user, the local one is applied according to an existing heuristic. In future work, we plan to enhance the effectiveness and the complementarity of the global and the local strategies, taking into account additional information on the execution of the software part (providing by the extension of HLS flow to the processors based architecture). Additional metrics linked to the design of embedded systems based on hardware accelerator and processors will thus be taken into account [18].

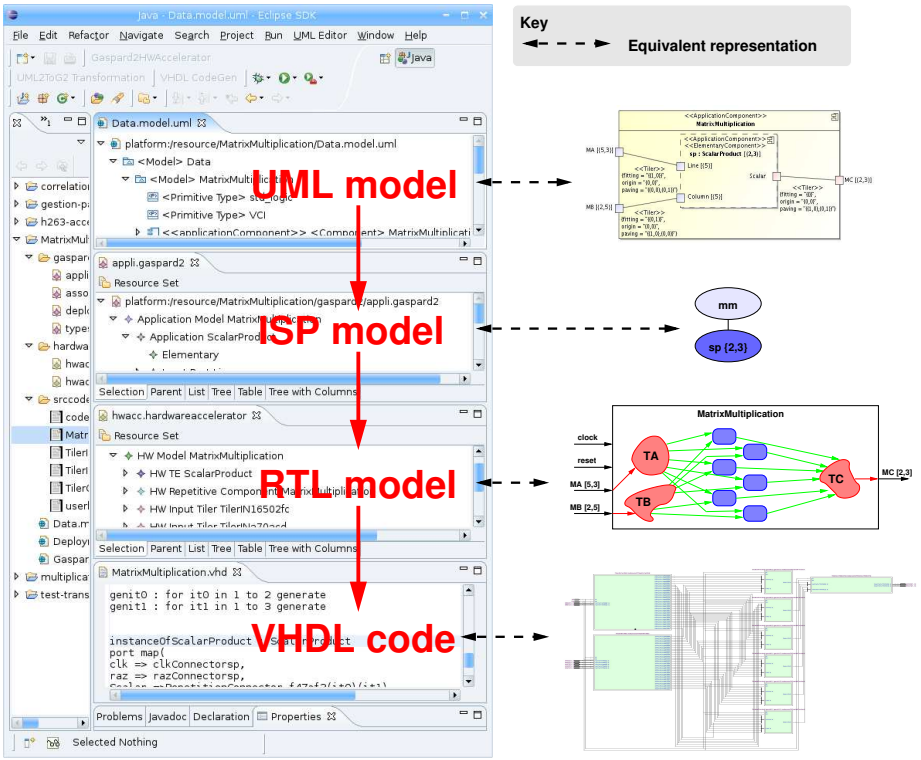


Figure 19: Using our HLS flow for the matrix multiplication application.

5 HLS Flow Overview

Our HLS flow provides to users an homogeneous environment for ISP applications. Applications are graphically modeled in UML, this drastically enhances the specification expressiveness by offering the opportunity to clearly identify the manipulated concepts (simple and complex data dependencies, potential parallelism, etc.): a UML model therefore matches the reality of the modeled application. The MARTE profile provides mechanism to factorize representation of data parallelism and complex data dependencies, it is therefore well suited for the modeling of real life massively parallel ISP applications. Moreover, the models relevancy is strongly enhanced by a specification which is independent of any implementation technology: an application designer just focuses on the specifications on the intention of the modeled application, not on its implementation. The refinement of such high level specification towards the RTL level is ensured by a model to model transformation. This demonstrates the abilities of the MDE based refinements between different abstraction levels. The exploration process extracts knowledge from RTL models in order to modify application models at high level, allowing a fast and a partially automatized exploration of the design space. The last key point for user is the VHDL code generation, it ensures the *productivity* of models specified at a high abstraction level.

The HLS flow is provided as an Eclipse plugin [12]. The screen-shot of Figure 19 corresponds to an usage of the HLS flow. The upper panel corresponds

to the UML model. The lower panel is the generated VHDL code. The other panels correspond to the intermediate models.

The design flow itself benefits from MDE for two categories of extension: *fine grain* and *coarse grain*. A fine grain extension aims to integrate new concepts in metamodels in order to extend them. The RTL metamodel is currently extended in order to manage the introduction of control flow into data flow applications. This is successfully realized with the creation of new concepts in metamodel and new rules in model transformations. A coarse grain extension consists of a modification of the design flow itself for new purposes. For instance, one can decide to create a model transformation in order to generate Verilog [45] code from the RTL metamodel or to create a RTL model from another metamodel (a metamodel used in another tool for instance). These flexibilities demonstrates the OMG point of view that advocates the development of tools using MDE: efforts done to develop a tool can be capitalized.

This flow is integrated in a co-design environment for high performance embedded system which is entirely build with MDE [18]. In this environment, the clear separation between the high level models and the technological models makes it easy to switch to different implementation technologies for a same application model: RTL and VHDL for synthesis, Transaction Level Modeling (TLM) and SystemC for simulation, procedural languages for multiprocessor execution and synchronous language for verification [12, 18].

6 Case Study

In order to evaluate the effectiveness of our HLS flow, we generate hardware accelerators for a video processing application. We consider the H.263 video codec standard [9]. It is suitable for low bit rate wireless video systems. In particular, it is useful for recent applications in cellular videophones, wireless surveillance systems, or mobile patrols. The encoder part of this application performs the most intensive computations. In this study, we focus on the DCT (Discrete Cosinus Transform) part of the encoder application. Indeed, this task needs up to 92% of the total computing power [4] of the encoder. The DCT task transforms the frame pixels into spatial frequency coefficients. The following tasks, not studied here, approximate the resulting coefficients by a small set of possible values in order to compress the data.

The application thus takes as input a stream of QCIF frames (176×144 pixels in the YCbCr format). Data are handled by *macroblocks*. A macroblock corresponds to a 16×16 pixel area of a video frame. It is represented in the YCbCr format, which contains a luminance component (Y), a blue chrominance component (Cb), and a red chrominance component (Cr). Luminance blocks describe the intensity, or brightness, of pixels, whereas chrominance blocks define the color of pixels. A macroblock contains six 8×8 blocks: four blocks for the luminance, one for the blue chrominance and one for the red chrominance. The DCT task is applied onto each macroblock of the original frames. Such bi-dimensional DCT task (so called DCT-2D) is often implemented by a mono-dimensional DCT task (DCT-1D) successively applied on the rows and the columns of macroblocks.

6.1 UML Modeling of the H.263 Encoder

The following deals with the UML modeling of the DCT part of the H.263 encoder application. We follow a bottom-up approach to present this UML model. We start with the selection of the elementary tasks, hierarchical tasks are afterward presented.

6.1.1 IP Selection and Deployment

We define the eight-point DCT-1D transformation as an elementary task. The chosen IP is the one developed in [4], it is based on an algorithm which minimizes the number of operations to perform. According to the direction (horizontal or vertical), the coefficients of the DCT are different. Two different IPs are thus distinguished, as illustrated in Figure 20: dctL manages the lines and dctC manages the columns. These IPs are contained in the same code file, as illustrated on the bottom part of the figure. The top of this figure represents the deployment of the DCTLine and DCTColumn elementary tasks (including the ports) onto these IPs.

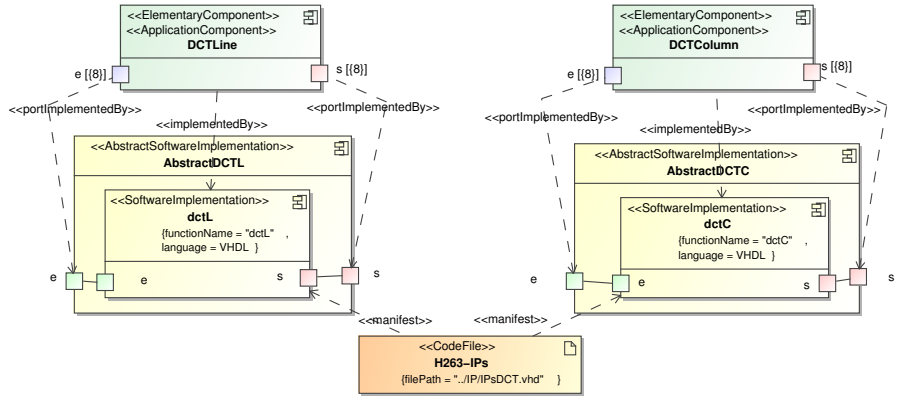


Figure 20: The DCTLine and DCTColumn elementary tasks are respectively deployed on the dctL and dctC IPs.

6.1.2 Block Modeling

These elementary tasks are used to build the DCTBlockLine and the DCTBlockColumn data parallel tasks, as illustrated on the top of Figure 21. The data arrays manipulated by these tasks are represented with the ports blockin and blockout. DCTBlockLine and DCTBlockColumn respectively apply the DCTLine and the DCTColumn tasks onto a 8×8 data array in order to produce a 8×8 data array. More precisely, DCTBlockLine applies the DCT on the lines of the input data array. The construction of the patterns relies on data dependencies expressed with tilers. These data dependencies are illustrated in the bottom of Figure 21. Each data in the data array is consumed once. Similarly to DCTBlockLine, DCTBlockColumn applies the DCTColumn task onto the columns of the input data array. The combining of these data parallel tasks allows one to construct the DCTBlock

hierarchical task illustrated in Figure 22. Indeed, by successively applying the DCTBlockLine and the DCTBlockColumn tasks onto a 8×8 data array, DCTBlock applies a DCT-2D task onto a block.

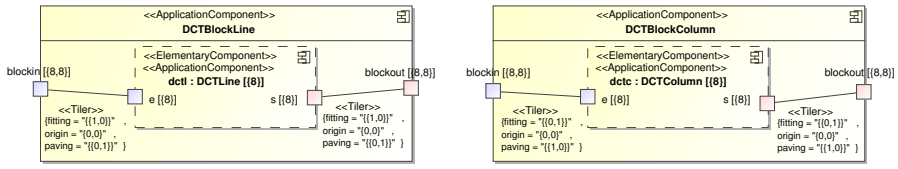


Figure 21: DCTBlockLine and DCTBlockColumn apply a mono-dimensional DCT onto blocks (top). Data dependencies expressed in DCTBlockLine (bottom).

6.1.3 Macroblock Modeling

A macroblock is composed of six blocks (four luminance blocks and two chrominance blocks). A macroblock thus corresponds to a of 8×8×6 data array. DCTMacroBlock applies the DCTBlock task onto each block contained in a macroblock, as shown in Figure 23. Each iteration in the repetition space can be performed independently from the others since each block is independent from the other blocks. In order to manage the overall video frame, another data parallel task is necessary. This data parallel task aims at applying the DCTMacroBlock task onto each macroblock contained in a QCIF video frame. This operation is performed by the QCIF task illustrated on the bottom of Figure 23. In a frame, the macroblocks are independent from the others. Each iteration of the DCTMacroBlock task thus can be executed independently from the others.

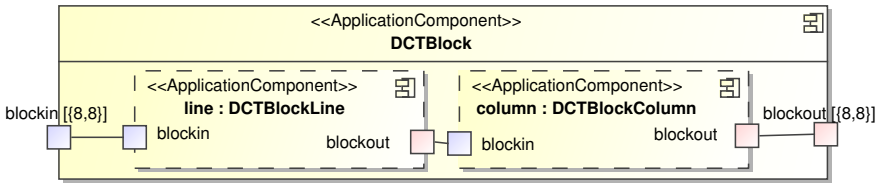


Figure 22: DCTBlock combines two data parallel tasks in order to apply a bi-dimensional DCT onto blocks.

6.2 Exploration of the Design Space

This UML model highlights the data parallelism available in the H.263 encoder application. According to the way this data parallelism is executed, different hardware accelerators can be generated by our HLS flow. In order to explore the overall design space, each configuration of the design space is evaluated. For this purpose, we use the PARALLELIZE and the SEQUENTIALIZE functions. They are applied on the ISP model generated with the UML2ISP model transformation.

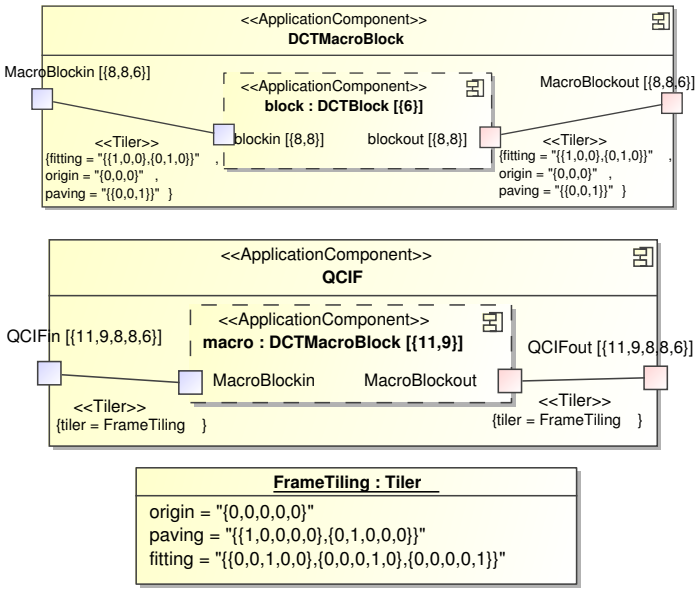


Figure 23: The DCTMacroBlock (resp. QCIF) data parallel task applies the DCTBlock (resp. DCTMacroBlock) task onto each block of the macroblock (up) (resp. each macroblock of the video frame (bottom)).

This ISP model is illustrated Figure 24. Here, the refactoring functions can impact the data parallelism available in the macro and the block data parallel tasks [9].

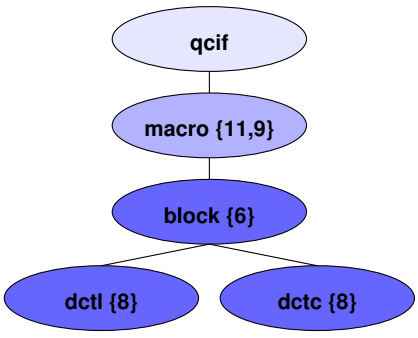


Figure 24: The ISP model generated from the UML model of the H.263 encoder application.

The multidimensional repetition spaces available in these data parallel tasks can be combined in many different manner, many configurations are thus gen-

[9]The loop transformation have no impact on the data parallelism available in dctl and dctc because of the *corner turn* like data dependencies expressed between these tasks (data are produced in line and are consumed in columns).

erated. According to the ISP2RTL model transformation and the estimation process, the corresponding hardware accelerators are instantaneously evaluated. Figure 25 summarises the characteristics of the generated hardware accelerators. As it was the case for the matrix multiplication example, each curve corresponds to hardware software partitioning and each point in a curve corresponds to a given parallel sequential partitioning of the hardware execution. Figure 25(a) represents the time necessary for the hardware accelerator to execute the DCT part of the H.263 encoder application onto a QCIF frame. As expected, the performance of the hardware accelerator depends on its area cost. Figure 25(b) and Figure 25(c) illustrate the overall execution time of the application, they respectively rely on an efficient and a less-efficient execution of the software part. These figures allow one to compare the hardware accelerator and to identify those which minimise the execution time and the area cost.

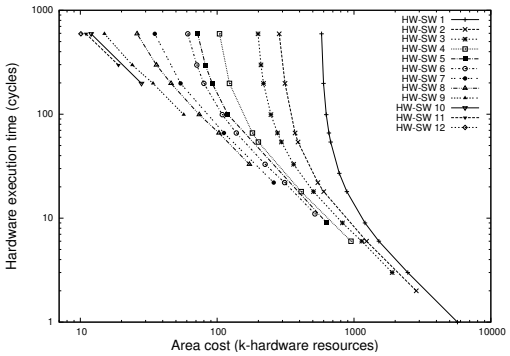
Few minutes were necessary to explore the design space using our HLS flow, while several days are necessary for an exploration at low abstraction levels. The exploration in our HLS flow relies on a single high abstraction model of the H.263 encoder application. The refactoring functions, the automatic model transformations and the estimation process thus allow one to efficiently explore the design space starting from this UML model. The selection of the hardware accelerator depends on the execution performance and the area cost criteria. An example of criteria specification is illustrated in Figure 25(b) and Figure 25(c) according to the satisfying solutions area box. The solutions inside this box satisfy the specified criteria.

In order to even more accelerate the exploration of the design space, the global and the local strategies are used. They are applied for an efficient and a less efficient software execution context (Figure 25(b) and Figure 25(c)). The global strategy starts with a software execution of the overall data parallelism. The corresponding generated hardware accelerator is represented with the mark A in the figures. The user then successively increases the data parallelism executed in parallel by the hardware accelerator. Once a solution satisfies the execution performance (mark B), the local strategy aims at minimizing the area cost. For this purpose, the heuristic successively increases the data parallelism to execute in sequential by the hardware accelerator. As result, the heuristic validate the solution marked C.

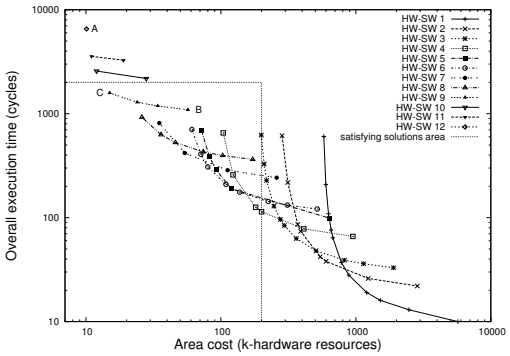
6.2.1 Integration of the Hardware Accelerator in an Embedded System

The RTL2VHDL model to text transformation ensures the automatic VHDL code generation of the selected solution. This code corresponds to a hardware accelerator able to executed the DCT part of the H.263 encoder application. This accelerator has been successfully integrated in an embedded system able to execute the overall H.263 encoder application [18].

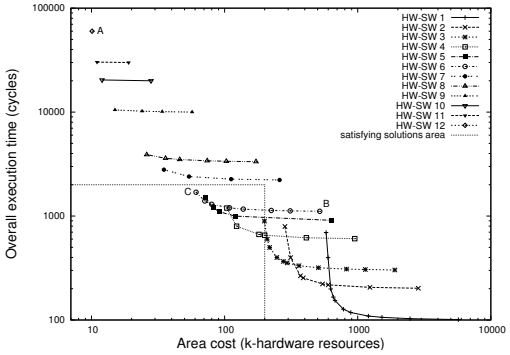
Our HLS flow thus enables to design an efficient hardware accelerator for an intensive signal processing application. Following our design space exploration strategies, we explored the design space at a high abstraction level using information resulting from the intermediate models generated by the transformation. The most effective partitioning has then been selected, without the inconvenience of low level implementations. The ease of modifications at high abstraction level



(a) Execution time of the hardware part.



(b) Overall execution time with a high performance software execution.



(c) Overall execution time with a non efficient software execution.

Figure 25: Execution performances and area costs of the hardware accelerators automatically generated by our flow.

coupled with the fast evaluations lead to a very powerful design space exploration framework.

7 Related works

During the last few years, the trend in HLS research field aims to generate HDL code from C or C-like language [17, 21, 22, 41]. Using C code to generate hardware design offers opportunity to work with a well known language and at higher abstraction level than RTL. However, this is not reliable without a major inconvenience. C-like description becomes difficult to handle for hierarchical applications that manage both tasks and data parallelism. The hierarchy appears like functions, the task parallelism is expressed using C extensions (*i.e.* pragma) and the data parallelism has to be extracted from loop statements. Moreover, just subsets are supported (pointers usage is often forbidden). Therefore, the users manipulate a subset of C syntax extended by annotations and lost interest in using standard. Some other approaches aim to specify application with polyhedrons. This offers the opportunity to use existing libraries for powerful optimizations in the IR [3, 14, 20]. Such textual description has the same inconvenience like the previously highlighted one.

An interesting high level language is the data-parallel formalism Alpha [47]. It manipulates polyhedra instead of arrays. This leads to different specification styles. Alpha particularly suits for the specification of *systolic* architectures. As a result, it does not offer a satisfactory solution to the design of other types of application models as it is needed here for intensive signal applications.

MDE has been increasingly adopted for the design of embedded systems in general [42]. 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 [37], UML SPT [36], UML-RT [44], TUT Profile [24], ACCOR/UML [26] and Embedded UML [31]. Because all these profiles may potentially overlap, significant standardization efforts have been recently realized by the OMG, resulting in the single unified and effective MARTE standard profile [35], on which our HLS flow relies. MARTE stands for Model and Analysis Real-Time Embedded system. Among other things, MARTE provides mechanism 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.

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 propositions use specific notations, defining an entirely *executable* model semantics [1, 34, 40]. Such expressive notations allow one to define models with sufficient information so that the specified system can be completely generated. However, the code is directly generated from the specifications, without any intermediary representation. The same is observed in the VHDL code generation from UML [5, 10], where the code is obtained directly by mapping the UML concepts with the VHDL syntax. More generally, this 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 directly being dependent on target languages or abstraction levels. Moreover, these tools focus on the finite state machines, they do not address ISP applications.

8 Conclusion

This paper advocates the use of the MDE methodology for the high level synthesis and the design space exploration. Indeed, in order to demonstrate the MDE advantages, we developed a model based HLSflow. This flow relies on a precise definition of sensitive features such as data parallelism and data dependencies that are suitable to support 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 designed at high abstraction levels in UML, the flow automatically performs successive refinements and generates the corresponding VHDL code. Such refinements rely on clear identification of concepts in the different abstraction levels and on a suitable decomposition of the model transformations into rules. We have also integrated an estimation process that allows one to evaluate the generated hardware accelerators. This accelerates the exploration of the design space which relies on strategies modifying high level models in order to meet the performance requirements in low level models. We have validated the relevance of our HLS flow for the design of a video processing application.

MDE abilities could also be used to extend the flow in order to enlarge the scope of managed applications, to target other implementation languages or to target 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 *MBD-MOMPES '06: 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, Washington, DC, USA, 2006. IEEE Computer Society.

[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] A. Ben Atitallah, P. Kadionik, F. Ghozzi, P. Nouel, N. Masmoudi, and H. Levi. Hw/sw codesign of the h. 263 video coder. pages 783–787, May 2006.

[5] Dag Björklund and Johan Lilius. From UML Behavioral Descriptions to Efficient Synthesizable VHDL. In *Proceedings of the 20th IEEE Norchip Conference*, November 2002.

[6] Pierre Boulet. Array-OL revisited, multidimensional intensive signal processing specification. Research Report RR-6113, INRIA, February 2007.

[7] 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.

[8] Peter Pin-Shan Chen. The entity-relationship model–toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[9] Guy Cote, B. Erol, M. Gallant, and F. Kossentini. H.263+: video coding at low bit rates. *IEEE Trans. On Circuits And Systems For Video Technology*, November 1998.

[10] 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.

[11] 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.

[12] DaRT Team LIFL/INRIA, Lille, France. Graphical array specification for parallel and distributed computing (GASPARD2). `https://gforge.inria.fr/projects/gaspard2/`, 2008.

[13] Tata Research Development and Design Centre. Modelmorf – a model transformer. `http://www.tcs-trddc.com/ModelMorf`.

[14] 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.

[15] Eclipse Consortium. EMF. `http://www.eclipse.org/emf`, 2007.

[16] Anne Etien, Cedric Dumoulin, and Emmanuel Renaux. Towards a unified notation to represent model transformation. Research Report RR-6187, INRIA, May 2007.

[17] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the streams-c c-to-fpga compiler: an applications perspective. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 134–140, New York, NY, USA, 2001. ACM Press.

[18] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Anne Etien, Rabie Ben Atitallah, and Philippe Marquet Jean-Luc Dekeyser. A Model Driven Design Framework for High Performance Embedded Systems. Research Report 6614, INRIA Lille - Nord Europe, August 2008.

[19] L. Grunske, L. Geiger, and M. Lawley. A graphical specification of model transformations with triple graph grammar. In Lecture Notes in Computer Science, editor, *First European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA*, volume 3748, pages 284–298, November 2005.

[20] 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.

[21] 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.

[22] 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.

[23] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proceedings of the Model Transformation in Practice Workshop*, Montego Bay, Jamaica, October 2005.

[24] 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.

[25] Linda Kaouane, Mohamed Akil, Thierry Grandpierre, and Yves Sorel. A methodology to implement real-time applications onto reconfigurable circuits. *J. Supercomput.*, 30(3):283–301, 2004.

[26] P. Lanusse, S.Gérard, and F.Terrier. Real-time modeling with UML : The ACCORD approach. In *UML 98 : Beyond the notation*, Mulhouse, France, 1998.

[27] Sébastien Le Beux. *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles.* Thèse de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, December 2007.

[28] Sébastien Le Beux, Vincent Gagne, El Mostapha Aboulhamid, Philippe Marquet, and Jean-Luc Dekeyser. Hardware/software exploration for an anti-collision radar system. In *The 49th IEEE International Midwest Symposium on Circuits and Systems*, San Juan, Puerto Rico, August 2006.

[29] Sébastien Le Beux, Philippe Marquet, and Jean-Luc Dekeyser. Multiple abstraction views of FPGA to map parallel application. In *3rd International Workshop on Reconfigurable Communication Centric System-on-Chips, ReCoSoC*, Montpellier, France, June 2007.

[30] 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.

[31] G. Martin, L. Lavagno, and J. Louis-Guerin. Embedded uml: a merger of real-time uml and co-design. pages 23–28, 2001.

[32] S. Meliá, J. Gomez, and J. L. Serrano. UPT: A Graphical Transformation Language based on a UML Profile. In *Proceedings of European Workshop on Milestones, Models and Mappings for Model-Driven Architecture (3M4MDA 2006). 2nd European Conference on Model Driven Architecture (EC-MDA 2006)*, July 2006.

[33] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, October 2005.

[34] 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.

[35] Object Management Group. A UML profile for MARTE, 2007. `http://www.omgmarte.org`.

[36] 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.

[37] Object Management Group, Inc., editor. *Final Adopted OMG SysML Specification.* `http://www.omg.org/cgi-bin/doc?ptc/06-0504`, May 2006.

[38] Object Management Group, Inc. MOF Query / Views / Transformations. `http://www.omg.org/docs/ptc/07-07-07.pdf`, July 2007. OMG paper.

[39] ProMarte partners. UML Profile for MARTE, Beta 1. `http://www.omg.org/cgi-bin/doc?ptc/2007-08-04`, August 2007.

[40] 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.

[41] 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.

[42] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):41–47, February 2006.

[43] Ed Seidewitz. What models mean. *IEEE Softw.*, 20(5):26–32, 2003.

[44] 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.

[45] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, fourth edition, May 1998.

[46] INRIA Triskell. Kermeta. `http://www.kermeta.org/`.

[47] Doran K. Wilde. The ALPHA Language. Technical Report 827, IRISA, France, 1994.

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399