Model Transformations for the Compilation of Multi-processor Systems-on-Chip

Éric Piel, Philippe Marquet, and Jean-Luc Dekeyser

INRIA Lille – Nord Europe & LIFL, University of Lille, France
e.a.b.piel@tudelft.nl, {philippe.marquet,jean-luc.dekeyser}@lifl.fr

Abstract. With the increase of amount of transistors which can be contained on a chip and the constant expectation for more sophisticated applications, the design of Systems-on-Chip (SoC) is more and more complex. In this paper, we present the use of model transformations in the context of SoC co-design. Both the hardware part and the software part of a SoC can be represented as a model using the MARTE standard from the OMG. We introduce the use of Model-Driven Engineering in order to generate executable code from a self-contained model of SoC.

First, we detail the restrictions and extensions we have brought to the MARTE profile in order to permit the complete description of the SoC as a model.

The compilation is a sequence of small and maintainable transformations that allows to pass gradually from a high-level description into models closer in abstraction to the final model, which is then converted into code. An in-depth view of one of the several transformation chains composing our tool is given. The implementation relies on the use of our experimental Java-based transformation engine which uses a hybrid declarative-imperative language.

We later discuss why model transformations fit better the compilation of the SoCs than traditional compilers. In particular, the re-use of transformations can greatly help with the fast evolution of SoC design, allowing development time reduction. Additionally, as each rule is small and relatively self-contained, their correctness is easier to ensure, which leads to more reliable compilation and indirectly more reliable SoCs.

1 Introduction

At the same time as advances in technology allow to integrate more and more transistors on a single chip, the embedded system applications get always more sophisticated. Although these two evolutions fit well in term of computation power, the combination put a strong pressure on the designers' capacity to design and verify the resulting very complex systems. As the International Technology Roadmap for Semiconductors has highlighted [1], there is a strong need for enhancing the design productivity. New design methodologies have to be adopted for the development of these large and complex Systems-on-Chip (SoCs).

A SoC contains on a single chip all the components of a computer: memory, processor, interconnection network, A/D and D/A converters... With the size

R. Lämmel, J. Visser, and J. Saraiva (Eds.): GTTSE 2007, LNCS 5235, pp. 459–473, 2008.

increase of chips, it is possible to put more components in a SoC. Additionally, due to physical restrictions in terms of frequency and voltage, to expand the processing power it is not possible to simply increase the size of the processor. It is necessary to put *several* processors in the system. Requiring from the software developers to handle the parallel programming paradigm in addition to the traditional concerns. Typically, the embedded systems are used in areas like multimedia (such as video encoding/decoding, HDTV), detection systems (such as radars, sonars), or telecommunications (such as mobile phones, antennas). All these applications are inherently multidimensional data flow applications.

In this paper, after introducing the specificities of SoC design, we will mention different approaches proposed until now for improving the productivity. Then we will give a brief description of a possible usage of Model-Driven Engineering in this context by presenting our development environment. The description of the metamodel for the specification will be followed by a close look at several model transformations allowing the compilation of a SoC model into simulation code. Then, mainly based on the acquired experience during the development of the presented transformations, we will highlight the benefits of model transformations for this particular purpose.

1.1 SoC Co-Design

One of the particularities of SoCs is that they are built for one specific application. Each new application leads to the design of a new architecture and new software, both exactly fitted to the task and specifically adapted to each other. Another particularity is that the initial cost for realization on the silicon (the creation of the mask of the chip) is very high, this mostly forbids the usage of prototypes. The SoC developers have to rely on simulations to test and verify their design.

The development of a SoC usually consists in the concurrent design of both the application and the hardware architecture, as illustrated in Figure 1. Each part is handled by different people, specialized on one of these domains. Then the application is mapped on the hardware, during the phase of association. This leads to generations of simulations of the full system. These simulations of both the hardware and the software together vary depending on the level of

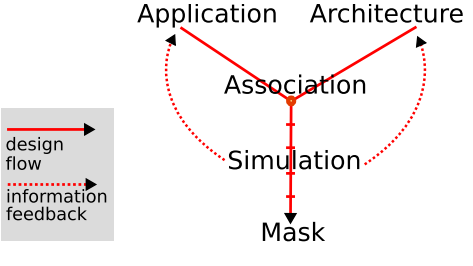


Fig. 1. Overview of the usual SoC development organization

abstraction. From the simulation results, the SoC designers can correct the SoC specification (the application, the architecture or the association) and obtain a new simulation. This is represented by the *information feedback* arrows on the figure. Gradually, the abstraction levels used for the description and the simulation are reduced in order to obtain more accurate observations.

At first, simulations at high levels of abstraction are used in order to rapidly obtain results about the system behaviour. Although there do not exist universally accepted levels of abstraction, typically those levels highly abstract the communications between the hardware components as well as their inner behavior between each cycle [2]. In some cases, the application is also abstracted such as proposed in [3] by using a special OS layer designed for the simulator or as proposed in [4] by only executing once each phase composing the application execution. The lowest abstraction level is usually the RTL (Register Transfer Level), from which the SoC can be synthesized. This journey through the representation of the same system at successive abstraction levels is typical of the SoC development. Classically, each time the abstraction level is reduced, the simulation has to be entirely re-written.

1.2 Related Works

In order to speed up the design flow, several methodologies have been proposed. The *system synthesis* methodology consists in transforming through successive refinements the original sequential specification into a concurrent specification defining all the implementation details. This relies on the use of high-level languages. At the beginning the system is represented only as a network of processes following a specific Model of Computation (MoC) such as KPN [5] (Kahn Process Network) or SDF [6] (Synchronous DataFlow). It is then rewritten in languages targeted toward hardware specification such as SystemC [7] or SpecC [8]. These languages allow to gradually specify the architecture part of the system with different levels of refinements down to a physical description. Such approaches have been proposed though projects such as COSMOS [9], Chinook [10], or Specsyn [11]. They have been the first approaches proposing automated transformations from a high abstraction level to a lower one. However, one of the main drawback was that the transformations were not complete, various tasks had to be done manually.

Another approach called *platform-based design* [12] aimed at reducing the work of the designers by supplying a parametrizable architecture. A specific tool allows the SoC designer to compose and configure a hardware platform in order to adapt it to the specific requirements of the application. The software is expressed using a high-level API. Tools such as VCC by Cadence [13], or N2C by Coware [14] offer such kind of platform. However, the platform provided by the tool is often specific to a particular application and targeting a different domain requires the costly introduction of a new platform into the tool by its makers.

The approach called *component-based design* [15,16] strives to provide the advantages of the two previous presented methodologies. The designers describe the whole SoC as a hierarchical network of virtual components and communication

channels. Each virtual component is a primitive internally specified at a low level of abstraction, typically at the RTL level for the hardware components and C or assembly language for the software components. The interconnection of the components is done via *wrappers*. This bottom-up approach allows designers to reuse efficient custom solutions with best performances. Unfortunately, the abstraction level at which the system is initially defined is limited, requiring from the developer to have already defined which part is hardware and which one is software and to write the system as a code.

Recently some propositions have appeared suggesting the usage of *Model-Driven Engineering* [17] in the specific context of SoC design. Early propositions have focused on the *modeling* side of this approach. In particular, several UML profiles dedicated to the representation of such systems have emerged. However, either they display the same problem as UML which has too many variation points to allow a complete specification only at the model level, such as SysML [18], or they tend to be very specific to one given implementation language, such as the standardization proposal by Fujistu [19] which is very tied to SystemC.

Some works are starting to appear on the usage of the second side of this approach: the *model transformations*. They highlight the need for model notation to have an entirely *executable* semantics [20], that is not having any semantic variation points and containing information so that each part of the system can be completely realized. Unfortunately, so far the propositions [20,21,22] have been limited as direct transformations from UML profiles to SystemC simulations. Similarly, in [23] a model transformation is used to pass from a high level model of the software to a compilable level. In our project called Gaspard [24], we went further in the usage of MDE by abstracting more the level of specification of the SoC and leveraging the model transformations to target the multiple types of outputs that might be needed during the SoC design. Moreover, each target is obtained not by the execution a one big transformation but by a chain of smaller and maintainable transformations.

2 Executable Models of SoC

In our project Gaspard the specification of the SoC is done exclusively via models. These models conform to the new UML profile called MARTE [25] standardized by the OMG. The MARTE profile *consists in defining foundations for model-based description of real-time and embedded systems*. It is specifically designed to permit description of both the hardware and the software parts of those systems. From the user point of view, using this profile has the benefits of using a standard representation (in other words, not being tied to a vendor specific representation) and of providing a high abstraction level, which is not directly associated to an implementation and more closely fits the generic concepts manipulated during SoC design. Additionally, one particular point of interest in our case for developing multi-processor SoCs (MPSoC) is the introduction of concepts

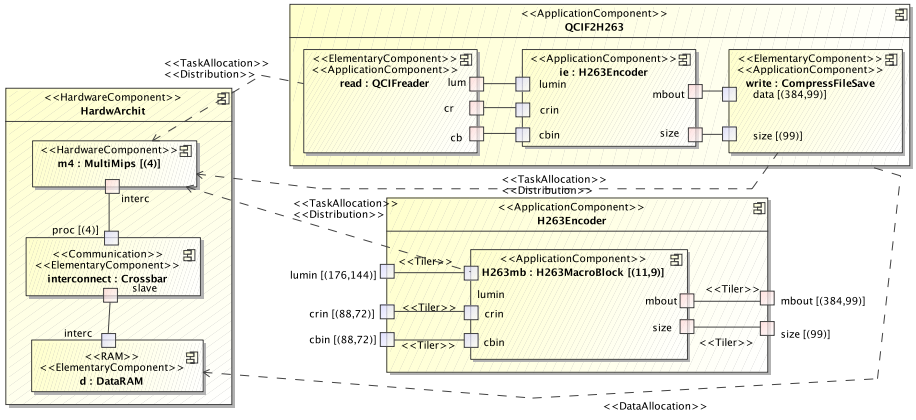


Fig. 2. Overview of a quadri-processor SoC with an association of an H.263 encoder application

for Repetitive Structure Modeling [26]. It allows to represent in a compact way both parallel architectures and parallel applications.

To use the MARTE profile as a means to represent a fully *executable system*, additional concepts and restrictions have been introduced as a supplementary profile. The necessity is twofold: first, the semantic variation points have to be eliminated to allow a non-ambiguous interpretation by the model transformations, and second, all the implementation details (down to the complete code of each function) have to be specified. In the Gaspard supplementary profile, the first point has been addressed by defining additional strict semantics on each package of the profile. For instance, the hardware components are defined to be represented only via the notions of components, ports and connectors, each of them having a precise meaning on the implementation. On the application side, while in MARTE the behaviour can be represented in ways nearly as broad as in UML, in Gaspard we have restrained the set of notions to a model of computation (MoC) based on ArrayOL [27]. It focuses on the expression of data flow applications with all their data and task parallelism. It is particularly well suited to the context of Gaspard that aims to design intensive signal processing applications. This is not a restriction of MDE: if the need arises to support a more generic context, another, less specific, MoC would have to be employed.

As an example, an overview of an MPSoC model using this profile is provided in Figure 2. On the left side is the main component of the architecture (four processors, a memory, and a crossbar). On the right side are the first two levels of an application dedicated to H.263 video encoding. Multiplicity (the numbers between brackets) allows to specify the repetition of a component in a compact and explicit way. In this example it is used for representing the four processors, as well as the 11×9 repetitions of the task *H263mb* (each repetition processing a different part of the picture). Not only this allows the designer to easily modify the configuration of the system, but it also permits to express the parallelism of

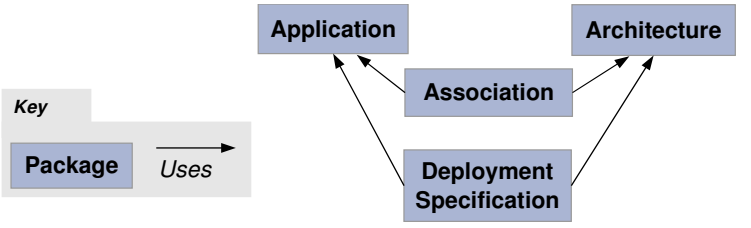


Fig. 3. Main packages used during an MPSoC design with Gaspard

the application. Stereotyped dependencies specify the distribution of tasks of the application on the processors and the distribution of the data on the memory.

To address the second point required to specify an executable system, a package of the Gaspard profile called *deployment specification* permits to link the elementary components (which can be considered as "black boxes") to source code. This mechanism can be used both for the architecture and the application components. In addition, special concepts allow to precisely map the interfaces of the components to the input and output of the functions. For a given functionality (e.g., Fast Fourier Transform, MIPS processor) several implementations can be provided, in different languages, or abstraction levels. With the creation of component libraries, this simplifies the SoC designer work as the deployment specification can be selected only once for each component and reused for each compilation target.

Figure 3 illustrates the relationships between the main packages used for the specification of an MPSoC in Gaspard. The two packages at the top define the architecture and the application. The deployment specification package introduces additional information concerning the implementation details, while the association package permit the mapping of the application on the architecture. Each of these packages correspond to a specific task during the design of the SoC. In particular, there is no dependency between the architecture and the application: it is possible to work on them concurrently.

3 Model Transformations for MPSoC Compilation

From the MPSoC model Gaspard provides several transformation chains. As output of a transformation chain, the user expects compilable code which can be used in already available tools. The Gaspard environment permits to select a *target* into which the SoC should be transformed. The most obvious target is a synthesizable hardware description and application code compilable for this particular hardware. As shown in Figure 4, other target possibilities encompass synchronous specification for formal verification, and simulations of the MPSoC at various abstraction levels. Each chain is a sequence of several model transformations separated by metamodels and finished by a code generation. For now the two code generations leading to *SystemC/PVT* and *SystemC/CABA* have not yet been fully implemented.

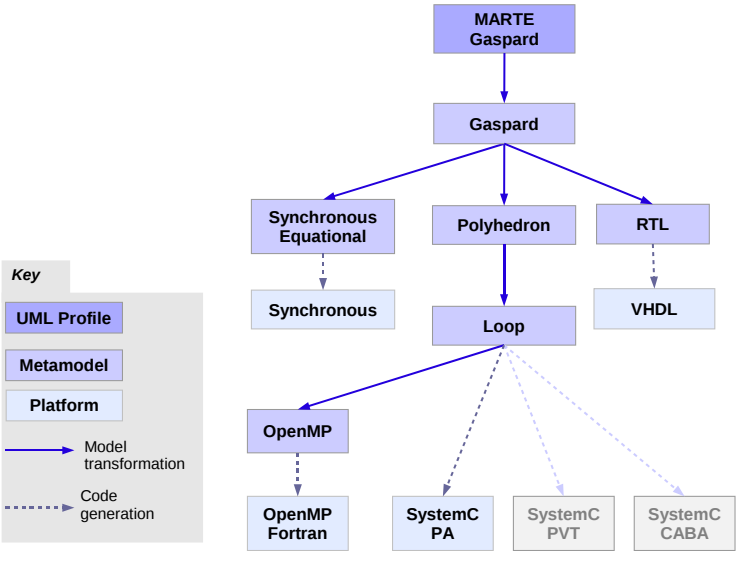


Fig. 4. The Gaspard compilation chains. From an MPSoC model in UML, indicated here as *MARTE/Gaspard*, each chain leads to a different target.

In the following subsections we will give an overview of the implementation of the transformations and the transformations chains. Interested readers are invited to refer to the Gaspard website [24] for downloading the environment (with the four transformation chains working) and examples of SoC models.

3.1 Implementation of the Transformations

Following the MDE recommendations, most of the transformations have a model as input and produce a model as output. They are called *model-to-model transformations*. In order to generate code, the final transformation of a chain takes a model as input but produces text as output. Such a transformation is are called *model-to-text transformation* (we sometimes use simply *code generation*).

Figure 5 presents the organization of a model-to-model transformation. A point to emphasize is that both the input and output are clearly defined by the metamodels to which they conform to. Using the declarative language approach, each transformation is actually an organized set of rules. Each rule works on a small part of the input metamodel specified as a pattern called the *left hand side* and produces a small part of the output metamodel called the *right hand side*.

The transformation is executed using a transformation engine. All the model-to-model transformations in Gaspard have been implemented for MoMoTE, a transformation engine developed by the team based on EMF [28] (Eclipse Modeling Framework). Each transformation has one top-level rule which is called initially. It usually matches the root component of the input model. From this rule, all the other rules are called (directly or indirectly).

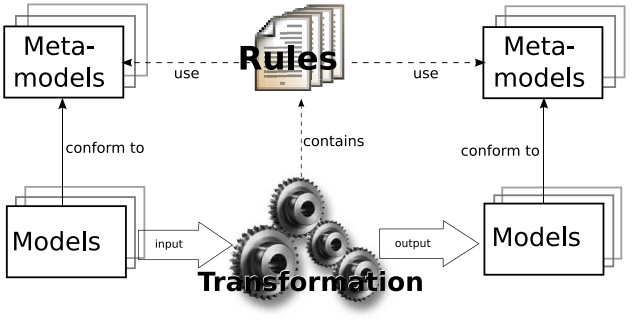


Fig. 5. Organization of a model-to-model transformation

Figure 6 presents an example of a transformation rule. It converts the concept of `HardwareComponentInstance` of the *Gaspard* metamodel into a equivalent concept of the *Polyhedron* metamodel. In MoMoTE, each rule is expressed as a Java class which contains five methods:

- The constructor for defining the sub-rules, and where the elements created by the sub-rules should be added. This is done via the `addRule()` method. In the example, it delegates the transformations of the *shape* and the *portInstances* to sub-rules.
- `getCondition()` for defining the left hand side using EMFT Query syntax (a part of EMF). In the example, the query looks for every `HardwareComponentInstance`.
- `create()` for defining the base element of the right hand side. In the example, it defines the right hand side as a `HardwareComponentInstance` of the *Polyhedron* metamodel.
- `process()` expresses how the new elements have to be created depending on the input. It is called once for each input element processed. In the example rule, it copies the name.
- `processReferences()` expresses how the references between the elements are produced depending on the input. In the example, it creates an equivalent reference as in the original model but pointing to the element created via another rule.

The last two methods are written with the usual Java imperative syntax, which confers a strong expressivity power to the rules. That is the place where the domain specific algorithms are situated. Even if for brevity, we have presented a rather straightforward rule, in the transformations that we have developed, one can find rules doing graph re-organization, static task scheduling, conversion between linear system representations, etc. Using Java as the underlying platform also proved to be advantageous for easily interfacing with external libraries or programs implementing a specific conversion.

These points are one of the reasons we used a Java-based transformation engine. Another reason is that at the beginning of this work, higher level engines

```
public class GaspardHI2PolyhedronHI extends Rule{

    public GaspardHI2PolyhedronHI()
    {
        super();
        setQueryFromContext(true);
        addRule(PolyhedronPackage.eINSTANCE.getComponentInstance_Dim(),
                new Gaspard2Shape2PolyhedronShape());
        addRule(PolyhedronPackage.eINSTANCE.getComponentInstance_PortInstance(),
                new Gaspard2PortInstance2PolyhedronPortInstance());
    }

    protected EObjectCondition getCondition(EObject srcElementContext)
    {
        return new EObjectTypeRelationCondition(
                    Gaspard2Package.eINSTANCE.getHardwareComponentInstance());
    }

    protected EObject create(EObject srcElement)
    {
        return PolyhedronFactory.eINSTANCE.createHardwareComponentInstance();
    }

    protected void process(EObject srcElement, EObject tgtElement)
    {
        HardwareComponentInstance hci=(HardwareComponentInstance) srcElement;
        gaspard2.metamodel.polyhedron.HardwareComponentInstance shci=
          (gaspard2.metamodel.polyhedron.HardwareComponentInstance) tgtElement;
        shci.setName(hci.getName());
    }

    protected void processReferences(EObject srcElement, EObject tgtElement)
    {
        HardwareComponentInstance hci=(HardwareComponentInstance) srcElement;
        gaspard2.metamodel.polyhedron.HardwareComponentInstance shci=
          (gaspard2.metamodel.polyhedron.HardwareComponentInstance) tgtElement;
        shci.setComponent(((gaspard2.metamodel.polyhedron.HardwareComponent)
          getTransformation().getGlobalRefs().get(hci.getComponent())));
    }
}
```

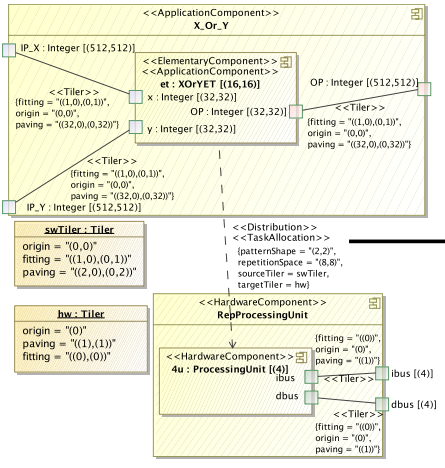
Fig. 6. Example of transformation rule using MoMoTE

such as TrML [29], or QVT [30] (Query, View, Transform) were not ready to be used. On the long term these kind of transformation engines will likely ease the development of the transformations.

In Gaspard, the code generations are executed using another engine, called MoCodE. This engine, also developed internally, can be seen as a layer over JET [31] (Java Emitter Templates). It allows to associate one (or several) template for each class of the input metamodel. The transformation is then represented as a set of templates which are called depending on the type of the elements in the input model. Each template is the text as it should appear in the output intermixed with Java code.

3.2 Example of Transformation Chain

As an example, we will describe here the transformation chain towards SystemC/PA. The PA level is a very high-level of abstraction of the simulation which permits a quick simulation and allows the user to see the execution of the program in



$$
\begin{cases}
p_0 \leq 0,3 - p_0 \leq 0 \\
4 \times mh_0 - p_0 + 1 \times q_0 + 1 \times q_1 \\
\qquad + 0 \times d_0 + 0 \times d_1 + 0 = 0 \\
-16 \times ms_0 - x_0 + 2 \times q_0 + 0 \times q_1 \\
\qquad + 1 \times d_0 + 0 \times d_1 + 0 = 0 \\
-16 \times ms_1 - x_1 + 0 \times q_0 + 2 \times q_1 \\
\qquad + 0 \times d_0 + 1 \times d_1 + 0 = 0 \\
q_0 \leq 0,7 - q_0 \leq 0 \\
q_1 \leq 0,7 - q_1 \leq 0 \\
d_0 \leq 0,1 - d_0 \leq 0 \\
d_1 \leq 0,1 - d_1 \leq 0 \\
x_0 \leq 0,15 - x_0 \leq 0 \\
x_1 \leq 0,15 - x_1 \leq 0
\end{cases}
$$

Fig. 7. Example of input and output of one of the rules of the Gaspard to Polyhedron transformation

term of pattern usages, the data element of a Gaspard program, instead of reads and writes of bytes. More information on this level can be found in [32].

The first transformation of the chain is one common to all the chains. It takes as an input the UML model conform to the MARTE/Gaspard profile, created by the SoC designer, and outputs a model conform to a metamodel featuring the same concepts than in the profile. Deliberately, this transformation is limited to a simple *translation*: this allows to convert the standardized view in UML which benefits the user into the representation specific to the domain in the metamodel which is much easier to manipulate in the transformations (because the only required concepts are present, there is no need to go through stereotype mechanism of UML, etc.).

The second transformation, from the Gaspard metamodel to the Polyhedron metamodel is composed of approximately 60 rules. They mainly allow us: to express the repetitions as polyhedrons, to separate the application tree following the association specification, to map the data arrays on the memories, and to simplify the deployment specifications. A polyhedron is a set of linear equations and inequations. Works on parallel scheduling often rel because –explained very rapidly– they permit to compute which iteration of a loop should be executed on which processor. A whole set of theories *and tools* are already available.

Figure 7 illustrates the effect of the rule producing a polyhedron from a distribution of a task on a processor. On the left side is the information as defined by the user, on the right side is a text representation of the polyhedron computed by the rule. The rule is called for every *ApplicationComponent* which has one (or several) *Distribution* starting from it. Using the various information specified on the *ApplicationComponent*, the *HardwareComponent*, the *Tilers*, and the *Distribution*, a set of equations and inequations are generated. It is then inserted as a specific attribute of the element generated out of the *ApplicationComponent*.

The third model transformation of the chain, from the Polyhedron metamodel to the Loop metamodel, converts the mapping expressed by the polyhedrons into pseudo-code expressions, as used by the code implementations. Each polyhedron is transformed into nested-loops. Even if theoretically this transformation could be merged with the previous one, we have decided to separate them for technical reasons: the computation is done via the call to an external program, CLooG [33], it is easier to maintain and troubleshoot this call independently from the rest of the transformations.

The final transformation of the chain is a SystemC code generation from the Loop metamodel. Based on the usage of templates as described previously, it generates both the simulation of the hardware components and the application components. Each hardware component is transformed into a SystemC module with its ports linked. For each processor, the part of the application which has to be executed on this processor is generated as a set of *activities* dynamically scheduled and synchronised, following the model of execution defined for the Gaspard applications on MPSoCs. Additionally, the framework needed to automatically compile all the simulation code is also generated (as a Makefile).

4 Advantages of Model Transformations for SoC Compilation

The usage of models for the design of multi-processor SoC is on its own a great improvement over current practice because it provides a higher abstraction level that especially helps for component reuse and parallel coding. The graphical representation also facilitates the global vision of complex systems and of interactions between the parts of of the system. As shown in [34], model-based approaches help the system designers to reuse preexistent works and to adapt it to new applications, increasing their productivity. Nonetheless MDE transformations also bring their batch of benefits into the SoC co-design when used for the compilation flow.

In Gaspard, the use of transformations permits to generate the various abstraction levels of the SoC out of the same model. This relieves the designer from manually re-writing the system each time a lower level of abstraction is targeted. In turn, this allows the designer to explore more configurations of the system to find one as close as possible from the optimal.

As Alanen et al. have emphasized in [21], one benefit of the introduction of transformations is the break down into small parts of the SoC compilation which can be more easily understood by the *SoC designer*. In contrast to custom monolithic tools where the meaning of the SoC model is only associated to the final output of the code generator, the compilation chains bring transparency. This transparency is useful for the users, as it helps to grasp the meaning of the concepts used for the SoC design at the different levels of compilation and see their evolution until the generated code.

The compilation flow is a chain of several small transformations. Of course, this can also be achieve with traditional methods, but the MDE simplifies the

separation between each transformation. Transformation inputs and outputs are formally and explicitly described by metamodels. Each intermediate metamodel is a strongly documented "synchronization point" of the compilation flow, which is complex to conceive within traditional compilers. The ease of expressing intermediate representations leads to the benefit of maximizing the *reuse* of transformations while compiling toward different targets. As in SoC compilation several targets are always necessary, at least for the different abstraction levels of simulation, and each compilation may have strong common points with the other compilations, the reuse possibility is very high. This is illustrated in the overview of our compilation chain available in Figure 4. The first transformation is reused across all the six chains and, similarly, the two successive transformations between the Gaspard metamodel and the Loop metamodel are shared by four chains. For the different SystemC targets, the chains vary only on the code generation. This minimizes the work required to create a transformation chain towards a new target and favors the use of already-validated transformations.

Additionally, the *explicit* separation between the various stages of compilation permits to easily share the development work of the compiler among several developers while maintaining the coherency of the global project. For instance, in our case, the current compilation flow of Gaspard have been carried out simultaneously by up to seven persons.

Moreover, model transformations can be written in a declarative way: the transformation is a set of mostly independent rules which have explicit declarations of their input and output patterns. In SoC design, the hardware is as versatile as the software. Likewise, the employed technologies evolve very quickly between the development of two products. For example, a few years ago all the SoCs were mono-processor. With the need to handle multi-processors, tools had to be adapted to manage the additional specificities introduced both on the hardware and the software parts. As another example, new abstraction levels for the simulation are proposed in order to accelerate the simulation as the size and complexity of the systems increase, such as described in [2]. The compilation flows have to be modified to also permit code generation at those higher abstraction levels. This fast evolution is typical of the SoC design. During its existence, the compilation flow is not static but must constantly adapt to the technology evolutions, be updated with the evolution of the standards used to represent the SoC, and has to integrate the improvements proposed by researchers on the already existing algorithms. Writing transformations in a declarative language increases their maintainability and simplifies their modification because it is easier to identify which part of the model they affect. This flexibility of evolution of the compiler speeds up the development of a SoC which is based on novel technologies.

In addition, independence in-between the rules is advantageous for reuse as they can be easily separated and regrouped. In Gaspard, the rules of the transformation from the Gaspard metamodel to the Polyhedron metamodel used to simplify the deployment specification have been integrated mostly as-is in the compilation chains towards VHDL and Synchronous languages.

Furthermore, correctness and reliability of the compiler are important factors in SoC design, especially in regard to the high cost of errors in its output. The constant evolution imposed on the compiler impedes on its quality. Model transformations permit to improve the quality for two reasons. First, this is due to the increased reuse and maintainability provided at the fine grain level of the rules and at the coarse grain level of the transformations. Second, the fact that each transformation has its output conform to metamodels allows to *formally* verify the structure of the models, at each stage of the compilation chain.

5 Conclusions

In this article we have first presented the various needs and constraints existing in the context of SoC co-design. The usage of Model-Driven Engineering in this context have been detailed using an example, our project Gaspard. First, the SoC designer creates a model of the SoC with all the information needed for the implementation, that is: without ambiguity and with all the details concerning the realization of even the most elementary components. Second, the model is used as the input of a chain of transformations taking the role of a compiler. Depending on the target selected by the user, a specific sequence of transformations is executed until the generation of code is reached. Each transformation has its input and output conforming to specific metamodels and is actually a set of rules. In addition, each rule is written in a declarative way, having an input pattern and an output pattern.

Then, we have emphasized the benefits of model transformations in the particular context of SoC compilation. The transparency brought by the transformations to the compiler contrasts with the usual monolithic tools and helps the user to better understand the concepts he has to manipulate. The strong separation between each transformation facilitates the simultaneous development of the compiler by different persons. Model transformations also ease the reuse between the numerous compilation chains present in the SoC design. Moreover, the separation of each transformation in rules permits not only to reuse them easily in different chains but also smooth the necessary constant evolution of the SoC compiler. Finally, another advantage is that the reliability of the compilation is improved thanks to better maintainability and the formal representation of each stage of the compilation as a metamodel.

The Gaspard environment already produces simulations capable of providing performance and consumption estimations. One of the topics we are now investigating is the automatization of the design space exploration. In order to remove a bottleneck pointed out by the generated simulation, the environment should be able to automatically find out which part of the original model has to be modified, and in which way. The traceability of a whole chain of transformations is an important step to achieve this goal. In future works we will explore additional features of the usage of MDE. In particular we will consider transformation composition as a way to facilitate even more the reuse within the compilation chains. Another perspective is the description of the transformations as models, which would allow a graphical representation of the transformation rules.

References

1. ITRS, International Technology Roadmap for Semiconductors: Design, 2005 edition (2005), http://www.itrs.net/
2. Donlin, A.: Transaction level modeling: flows and use models. In: Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Stockholm, Sweden, pp. 75–80 (2004)
3. Honda, S., Wakabayashi, T., Tomiyama, H., Takada, H.: RTOS-centric hardware/-software cosimulator for embedded system design. In: Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS 2004), Stockholm, Sweden (September 2004)
4. Hamerly, G., Perelman, E., Lau, J., Calder, B.: Simpoint 3.0: Faster and more flexible program analysis. In: Workshop on Modeling, Benchmarking and Simulation, Madison, Wisconsin, USA (June 2005)
5. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information Processing 1974: Proceedings of the IFIP Congress 1974, pp. 471–475. North-Holland, Amsterdam (1974)
6. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. on Computers (January 1987)
7. Grotker, T., Liao, S.: Al: System Design with SystemC. Kluwer Publishers, Dordrecht (2002)
8. Gerstlauer, D., Peng, G.: System Design: A Practical Guide with SpecC. Kluwer Academic Publishers, Dordrecht (2001)
9. Ismail, T.B., Abid, M., Jerraya, A.: COSMOS: a codesign approach for communicating systems. In: CODES 1994: Proceedings of the 3rd international workshop on Hardware/software co-design, Los Alamitos, CA, USA, pp. 17–24. IEEE Computer Society Press, Los Alamitos (1994)
10. Chou, P., Ortega, R., Borriello, G.: The chinook hardware/software co-synthesis system. Technical Report TR-95-03-04 (1995)
11. Gajski, D., Vahid, F., Narayan, S., Gong, J.: Specsyn: An environment supporting the specify-explorerefine paradigm for hardware/software system design. IEEE Transactions on Very Large Scale Integration Systems 6(1), 84–100 (1998)
12. Chang, H., Cooke, L., Hunt, M., Martin, G., McNelly, A.J., Todd, L.: Surviving the SOC revolution: a guide to platform-based design. Kluwer Academic Publishers, Norwell (1999)
13. Schirrmeister, F., Sangiovanni-Vincentelli, A.: Virtual component co-design – applying function architecture co-design to automotive applications. In: Proceedings of the IEEE International Vehicle Electronics Conference, Tottori, Japan (September 2001)
14. CoWare inc.: CoWare N2C (2001), http://www.coware.com/cowareN2C.html
15. Cesário, O.W., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S., Jerraya, A. A., Gauthier, L., Diaz-Nava, M.: Multiprocessor SoC platforms: A component-based design approach. IEEE Des. Test 19(6), 52–63 (2002)
16. Jerraya, A.A., Yoo, S., Bouchhima, A., Nicolescu, G.: Validation in a component-based design flow for multicore SoCs. In: ISSS, pp. 162–167. IEEE Computer Society, Los Alamitos (2002)
17. Planet MDE: Model Driven Engineering (2007), http://planetmde.org
18. Object Management Group, Inc., ed.: Final Adopted OMG SysML Specification (May 2006), http://www.omg.org/cgi-bin/doc?ptc/06-0504

19. Object Management Group, Inc., ed.: UML Extension Profile for SoC RFC (March 2005), http://www.omg.org/cgi-bin/doc?realtime/2005-03-01
20. Nguyen, K.D., Sun, Z., Thiagarajan, P.S., Wong, W.F.: Model-driven SoC design via executable UML to SystemC. In: RTSS 2004: Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004), Washington, pp. 459–468. IEEE Computer Society, Los Alamitos (2004)
21. Alanen, M., Lilius, J., Porres, I., Truscan, D., Oliver, I., Sandstrom, K.: Design method support for domain specific soc design. In: MBD-MOMPES 2006: 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 2006), pp. 25–32. IEEE Computer Society, Washington (2006)
22. Riccobene, E., Scandurra, P., Rosti, A., Bocchio, S.: A model-driven design environment for embedded systems. In: DAC 2006: Proceedings of the 43rd annual conference on Design automation, pp. 915–918. ACM, New York (2006)
23. Szemethy, T., Karsai, G., Balasubramanian, D.: Model transformations in the Model-Based Development of real-time systems. In: ECBS 2006: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, Washington, pp. 177–186. IEEE Computer Society, Los Alamitos (2006)
24. WEST Team LIFL, Lille, France: Graphical array specification for parallel and distributed computing (GASPARD-2) (2005), http://www.lifl.fr/west/gaspard/
25. ProMarte partners: UML Profile for MARTE, Beta 1 (August 2007), http://www.omg.org/cgi-bin/doc?ptc/2007-08-04
26. Boulet, P., Marquet, P., Piel, E., Taillard, J.: Repetitive Allocation Modeling with MARTE. In: Forum on specification and design languages (FDL 2007), Barcelona, Spain, (September 2007) (Invited Paper)
27. Boulet, P.: Array-OL revisited, multidimensional intensive signal processing specification. Research Report RR-6113, INRIA (February 2007)
28. Eclipse Consortium: EMF (2007), http://www.eclipse.org/emf
29. Etien, A., Dumoulin, C., Renaux, E.: Towards a unified notation to represent model transformation. Research Report RR-6187, INRIA (May 2007)
30. Object Management Group, Inc.: MOF Query / Views / Transformations, OMG paper (November 2005), http://www.omg.org/docs/ptc/05-11-01.pdf
31. Eclipse Consortium: JET, Java Emitter Templates (2007), http://www.eclipse.org/modeling/m2t/?project=jet
32. Atitallah, R.B., Piel, E., Niar, S., Marquet, P., Dekeyser, J.L.: Multilevel MPSoC simulation using an MDE approach. In: IEEE International SoC Conference (SoCC 2007), Hsinchu, Taiwan (September 2007)
33. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 13 IEEE International Conference on Parallel Architecture and Compilation Techniques, Juan-les-Pins, France, pp. 7–16 (September 2004)
34. Bunse, C., Gross, H.G., Peper, C.: Applying a model-based approach for embedded system development. In: EUROMICRO 2007: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007), Washington, pp. 121–128. IEEE Computer Society, Los Alamitos (2007)