A UML Model-Driven Approach to Efficiently Allocate Complex Communication Schemes

Andrea Enrici, Ludovic Apvrille, and Renaud Pacalet

Institut Mines-Telecom, Telecom ParisTech, CNRS/LTCI, Biot, France
{andrea.enrici,ludovic.apvrille,renaud.pacalet}@telecom-paristech.fr

Abstract. To increase the performance of embedded devices, the current trend is to shift from serial to parallel and distributed computing with simultaneous instructions execution. The performance increase of parallel computing wouldn't be possible without efficient transfers of data and control information via complex communication architectures. In UML/SysML/MARTE, different solutions exist to describe and map computations onto parallel and distributed systems. However, these languages lack expressiveness to clearly separate computation models from communication ones, thus strongly impacting models' portability, especially when performing Design Space Exploration. As a solution to this issue, we present Communication Patterns, a novel UML modeling artifact and model-driven approach to assist system engineers in efficiently modeling and mapping communications for parallel and distributed system architectures. We illustrate the effectiveness of our approach with the design of a parallel signal processing algorithm mapped to a multiprocessor platform with a hierarchical bus-based interconnect.

Keywords: Model Driven Engineering, Hardware-Software Co-Design, Design Space Exploration, Parallel Computing, Embedded Systems.

1 Introduction

Today's embedded systems are more and more realized as parallel systems where the processing and the control are distributed over a network of interconnected subsystems. Such systems are typically deployed to perform parallel computing for data-dominated applications where performance is driven by both data processing and data transfers. Currently, we find these parallel and distributed systems both at the chip level (e.g., Multi-Processors Systems on Chip) or in domains where the electronics components are physically distributed over the structure of the whole system (e.g., automotive and avionics systems). In this context, an important challenge is to efficiently program these complex architectures where interactions between computations and transfers, from both hardware and software points of view, significantly impact the software development (e.g., time-to-market of new products, development time and costs).
Among the possible approaches that can be taken to alleviate application software development, there is raising the level of abstraction at which these systems

are programmed, e.g., with the aid of Model Driven Engineering [1]. Thus, instead of manually programming a parallel and distributed system, a developer can separately model both the application(s) - i.e., the functional part of the system - and the candidate resources - i.e., the hardware architecture - therefore abstracting out low-level details (e.g., memory addressing modes) with the guidance of Electronic Design Automation tools. Then he/she selects the architecture units for executing the function's workload (*mapping*) and once a solution compliant with the predefined performance requirements is reached, the application code can be generated via automated model transformations. Finding a mapping solution compliant to some performance requirements (i.e., Design Space Exploration, DSE) is typically an iterative process: performance numbers are first extracted from mapping models. Then, according to these numbers, premapping models are improved and the process starts over until performance numbers converge to the desired performance requirements.

The performance of a data-dominated application executed on a parallel and distributed system is driven both by computations (i.e., processing) and by communications (e.g., data transfers). However, UML/SysML models intertwine both computation entities (i.e., classes/blocks) and communication entities (i.e., relationships/ports) aspects within the same diagrams. This lack of separation of concerns causes serious issues when models are to be modified due to DSE. As communications cannot be described separately from computations, input models must be re-designed from scratch each time a mapping alternative does not match the desired performance requirements. Thus, models mix information about the functionality of an application (i.e., the computations to be carried out) with information that is specific to a given architecture (i.e., how data can be transferred). This dramatically limits models' portability, transformations and impacts the time, costs and quality of a model-driven design.

In response to the above issues, this paper presents a novel approach and the corresponding artifacts to separately describe and map communications and computations, independently of the pair application-architecture. We apply our modeling approach to DiplodocusDF, a UML Model-Driven Engineering methodology for the rapid prototyping of data-dominated applications onto heterogeneous Multi-Processor System-on-Chip (MPSoC) architectures. In the scope of DiplodocusDF, we make use of its UML/SysML modeling facilities that are supported by the open source toolkit TTool [2]. Last, we show the benefits of our TTool/DiplodocusDF via the Design Space Exploration of a complete system composed of a signal processing application mapped onto an MPSoC architecture.

The rest of this paper is organized as follows. In Section 2 we describe in greater detail the problem statement accompanied by a clarifying example. Section 3 presents our systematic approach to separate computations and communications in the input specification models. Section 4 applies these principles to the modeling assets available in UML/SysML and demonstrates how we are able to solve the example problem of Section 2. The case study of Section 5 presents the implementation of our approach in TTool/DiplodocusDF, in the context of

a complete pair application-architecture. Section 6 discusses our contributions with respect to related works and Section 7 concludes the paper.

2 Problem Statement

In this section we will describe the problem statement in greater detail. We start with a discussion related to the application of the Y-chart approach [3] to map UML/SysML application models onto parallel and distributed architectures. Next, we extend the discussion with a practical example and state the problems that we aim at solving in this article.

2.1 Design Space Exploration with the Y-chart in UML/SysML

The problem that system engineers face when working with parallel and distributed architectures is the many design alternatives involved. The Y-chart approach (see Fig. 1) has been proposed as a methodology to help designers "to explore the design space of an architecture template in a systematic way, to design programmable embedded systems that are programmable and satisfy the design constraints" [3]. It has become a de facto standard approach underlying many Electronic Design Automation (EDA) tools and methodologies.
It is our belief, however, that applying the Y-chart of Fig. 1 to map applications modeled in UML/SysML, onto parallel and distributed architectures, is inefficient. This is due to a lack of separation of concerns in the application models between operations (i.e., processing and control operations), represented in UML diagrams as classes, and their dependencies, represented in UML diagrams as relationships. Indeed, the semantics associated by UML/SysML diagrams to relationships works well for applications mapped to architectures where operations are sequentially executed onto centralized units. Typically in these contexts, communications[1] have a small impact on performance and are executed on simple point-to-point paths (e.g., memory-bus-memory). On the other hand, relationships in UML/SysML diagrams are not suited to describe dependencies among operations when the latter are executed by parallel and distributed units that require intensive communications on complex paths, affecting performance.

2.2 A UML/SysML producer-consumer example in TTool

Fig. 2 depicts the scenario of our example modeled in TTool. Fig.2a shows a sample application made up of a pair of producer-consumer operations, interconnected by a relationship $ch1$ which represents the exchange of data from the producer to the consumer. Fig.2b illustrates a sample architecture where the producer-consumer application is mapped. The producer operation is mapped

[1] In this article we loosely use the word communications to refer to any transfer of data or control items

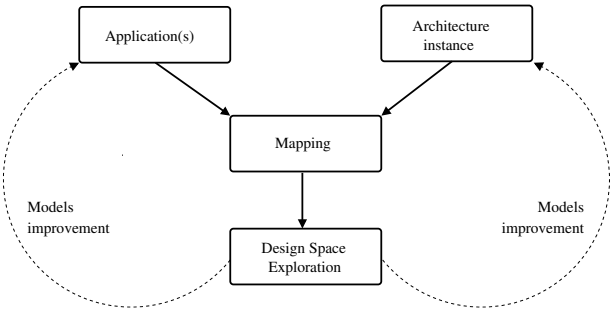

Fig. 1. The Y-chart approach for the design of programmable embedded systems

to the Digital Signal Processor (DSP) DSP1, the consumer operation to the Central Processing Unit (CPU) CPU1. Due to the capabilities of DSP1 and CPU1, we can imagine that DSP1 is able to directly store its output data only to Memory1 and CPU1 is able to directly retrieve its input data only from Memory2. Thus, in order to execute the consumer operation, a transfer is needed to move the producer's data from Memory1 to Memory2. Moreover, such a transfer can either be issued with a bus transaction or with a Direct Memory Access (DMA) transaction. So how to describe it in UML/SysML?

We encounter here a first problem, namely **a modeling problem**: a lack of expressiveness to describe at the same time (1) a specific transfer, (2) the architecture: units involved and (3) the way the transfer is performed. Typically what a system designer would do is to create a second instance of the application model of Fig. 2a, as depicted in Fig. 2c. In the latter an additional operation is injected between the producer and the consumer to imitate how data can be transferred with a DMA transaction. However, such an arrangement does not

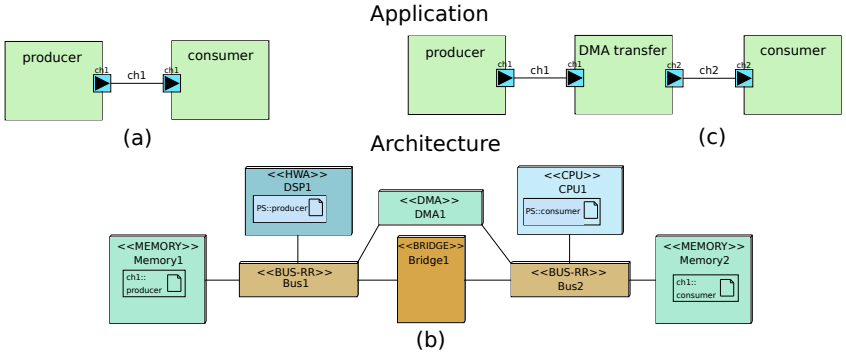

Fig. 2. A sample producer-consumer application mapped over an architecture in UML/SysML with TTool

prevent us from running into another issue if, for instance, it turns out that the DMA transaction is not efficient enough or if we want to map the application model to a different architecture that does not include any DMA engine. In either case the application model must be re-designed from scratch.
Thus, we face a second problem, namely **a mapping problem**: how to map a relationship ($ch1$ in this case) to the description of a transfer, in a portable way (i.e., in a way that prevents a designer from re-modeling the application)?

If the MARTE [4] profile has been specifically defined for the modeling of complex systems, and moreover supports the definition of scenarios for the usage of resources (BehaviorScenarios), it is unfortunately also not well adapted for reasons that are discussed in the related work section (Section 6).

3 The Approach

Fig. 3 shows a global view of the approach we propose in response to the modeling and mapping problems introduced in Section 2. The overall goal of our approach is to assist the system designer with a systematic methodology to separately define the modeling assets that are needed to describe a **triplet application-architecture-communications**. This separation of concerns aims at minimizing the intersection between modeling concepts that would otherwise be mixed in the application and the architecture descriptions. We believe that this lack of separation of concerns is at the root of the modeling and mapping problems of Section 2. In this section, we describe how the Y-chart approach of Fig. 1 is extended to accommodate for separate models for the application, the architecture and communications. At the same time we define the vocabulary of some key concepts that will be used throughout the paper.

In our vision, a **communication model** (Communication modeling box) acts as a an interface between the application and the architecture models. On one hand, the main purpose of an application model (Application(s) box), is to express the functionality of a given algorithm in terms of processing operations and control operations as well as in terms of the data and control dependencies among these operations. On the other hand, from the viewpoint of communication modeling, the main purpose of an architecture model (Architecture instance box) is to express the topology[2] of the system's architecture. The latter can be roughly defined as the structure of the interconnections of all the architecture units. For our purposes, the communications that are of interest are those needed to transfer data between a source and a destination storage units. Thus, the above topology must express all the possible *transfer paths*, defined as the set of interconnected architecture units that are involved in moving data from a source to a destination storage unit, as well as in exchanging the control information that configure such a data transfer. A communication model (Communication

[2] The topology includes performance parameters

modeling box) aims to match the needs expressed by data dependencies in the application, to the capacity of an architecture to serve such needs. In order to capture these concepts, we need a communication model to elegantly express the architecture units that participate in a data transfer, the *transfer components*, and to allow a system designer to describe the *transfer algorithm* that must be put in place to move data according to the dependencies of the application. More specifically, we define a *transfer algorithm* as the set of activities that must be executed by the *components* to move data in a *transfer path*. Also the mapping phase (Mapping box) must be defined to accommodate for our separation of concerns and it is composed of two stages. First, the workload of the application algorithm in terms of processing and control operations is projected over the specific architecture instance. In this stage, the designer selects the architecture units that will execute the processing and control operations. Next, the workload expressed by the control and data dependencies of the application is projected onto the *transfer paths* of the specific architecture instance through the communication models (*components* and *algorithm*). When exploring different mapping alternatives in the design space, we now dispose of separate modeling assets - i.e., application, communication architecture models - that can be individually modified without impacting on each other.

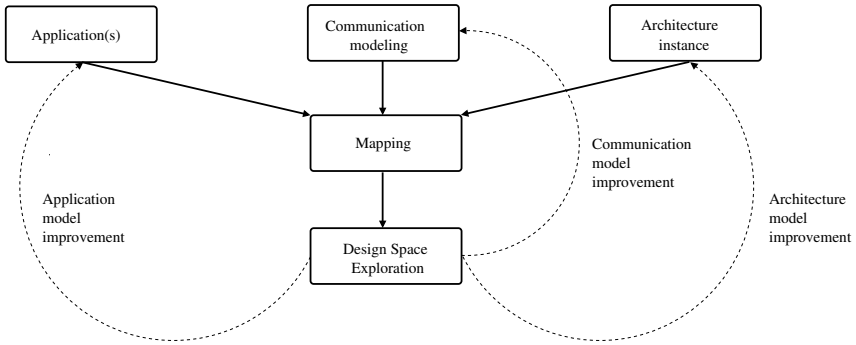

Fig. 3. The Y-chart approach extended with the separation of concerns between the application and the architecture models

4 Communication modeling and mapping in UML/SysML: Communication Patterns

In this section we put into effect the principles described in our extended Y-chart approach with the modeling assets available in UML/SysML. Next, we practically show how these assets can be used to solve the producer-consumer issues illustrated in Section 2. We regroup the UML/SysML modeling assets under the name of Communication Pattern, that defines a single modeling artifact used to

model one or more transfers.

4.1 Models for the Communication Pattern's transfer algorithm

In order to model a transfer algorithm, we first need to abstract out the *activities* that take place in the communication protocols or standards of the architecture instance. Further, we need to compose these *activities* and express their dependencies by means of some sort of structure or hierarchy. In the scope of UML/SysML, the diagrams that we estimated to be suitable for these purposes are Behavior Diagrams, and more specifically both Activity and Sequence Diagrams. The Activity Diagrams of a Communication Pattern capture the structure and dependencies of the simple and repeatable *activities* that are part of a transfer algorithm (e.g., program a DMA, execute a bus transaction). Each of these simple and repeatable *activities* is then described either directly by Sequence Diagrams or recursively via other Activity Diagrams. Within an Activity Diagram, *activities* are composed by operators to describe concurrency, sequencing, choice and iteration. The latter two are governed by control variables that are global to a given Activity Diagram and to all the diagrams that it references. An Activity Diagram is associated to a set of components which is global to all the diagrams it references.

The Sequence Diagrams of a Communication Pattern describe the way components interact in order to execute an activity, a well as the order in which these *interactions* are executed. The lifelines of Sequence Diagrams are associated to instances of components. *Interactions* are described via the exchange of parameterized messages (e.g., Read(), Write()) representing an abstraction of the signals wired on bus lines. In order to separate the control aspects from message exchanges, Sequence Diagrams are limited to asynchronous messages and Activity Diagrams are limited to above control operators, diagrams dependencies. Indeed, the latter are instantiated in Activity Diagrams when describing the dependencies among multiple Sequence Diagrams[3].

4.2 Communication Pattern's components

In order to describe the executors of a transfer algorithm, we classify the architecture units into three classes of components: *storage*, *transfer* and *controller*.

- A *storage component* is an architecture unit whose main functionality is to store input/output data produced or consumed by a processing operation, e.g., a RAM memory, a buffer.

[3] In analogy with computer science, we can see the messages exchanged in Sequence Diagrams as the low-level instructions of a given *transfer algorithm*. These instructions are grouped into the *activities* captured by both Activity and Sequence Diagrams. Activities can be thought of as routines in programming languages

– A *transfer component* is an architecture unit whose main functionality is to physically move data items between components, e.g., a AMBA bus, a CAN bus, a DMA.
– A *controller component* is an architecture unit whose main functionality is to coordinate a data transfer by configuring a *transfer component*, e.g., a Central Processing Unit, a microcontroller, a Digital Signal Processor.

When modeling a communication (Communication modeling box in Fig. 3) we use these three classes of components to describe a generic transfer algorithm independently of the architecture units of a given instance. This abstraction allows the transfer algorithm to be portable with respect to the system's architecture. Finally, when projecting the application workload expressed by data dependencies onto the architecture, (Mapping box in Fig. 3) these abstract components are mapped to the specific units of the architecture instance.

4.3 Modeling the consumer-producer problem with Communication Patterns

In this subsection we illustrate how a Communication Pattern can solve the modeling and mapping problems for the producer-consumer example of Section 2.

In the architecture model of Fig. 2b, according to our classification of components, we dispose of: two storage objects (i.e., Memory1 and Memory2), three transfer objects (i.e., Bus1, Bus2 and Bridge1) and three controller objects (i.e., DSP1, DMA1 and CPU1). In the application of Fig. 2a, we have two processing operations (i.e., producer and consumer) and one data dependency (i.e., channel ch1). The mapping of the application workload in terms of processing operations has already assigned the producer to DSP1 and the consumer to CPU1. The access capabilities of DSP1 and CPU1 force the producer output data to reside in Memory1 and the consumer input data to be accessible from Memory2. This scenario thus defines the need to have one transfer from Memory1 to Memory2. Such a transfer can be executed in two ways: either via a bus transaction or a DMA transaction. When modeling we know nothing about the performance numbers of the two transfer options, so we have to model the communication in the most generic possible way; thus, in terms of components we need two storage, one controller and three transfer components. The transfer algorithm is fairly simple given we only have one data dependency; it is illustrated in Fig. 4a: first the transfer component is programmed by the controller (ProgramTransfer box), then data is moved iteratively from the source to the destination storage by the transfer component (ExecuteTransfer box, loop operator) until an integer counter reaches the value zero. At this point the controller is informed of the completion (AcknowledgeTransfer box). Fig. 4b illustrates the Sequence Diagram corresponding to the activity ExecuteTransfer of the algorithm, Fig. 4a. Data is moved from the source storage to the destination storage via the transfer components.

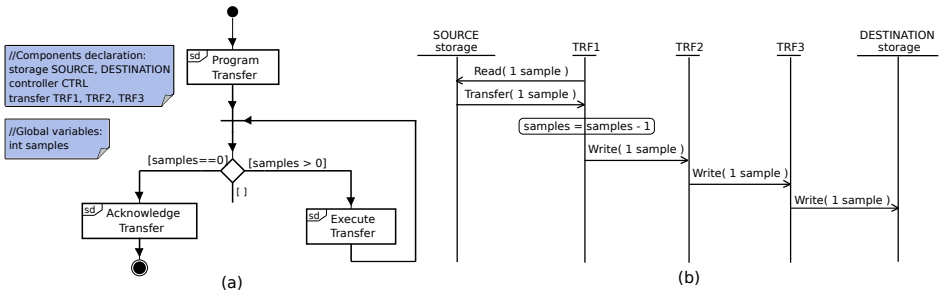

Fig. 4. The transfer algorithm for the consumer-producer example (a). The interactions among components corresponding to the ExecuteTransfer activity of the algorithm (b)

4.4 Mapping the producer-consumer problem with Communication Patterns

During mapping phase, the behavior diagrams of a Communication Pattern are arranged to match the capacity of the architecture instance. Since in our consumer-producer example we deal with only one data transfer we dispose of the full architecture capacity and we do not need to arrange the algorithm modeled in the Activity Diagram of Fig. 4a. However, in case the producer-consumer application required an algorithm to model multiple transfers, the algorithm would have been arranged to match the limited parallelism available in Fig. 2b. In the latter, there are two transfer paths from Memory1 to Memory2: (1) Bus1-Bridge1-Bus2 if a bus transaction is to be issued or (2) Bus1-DMA1-Bus2 in case a DMA transaction is to be issued. The choice between which of the two paths performs best is a matter of performance analysis and will not be treated in this paper as we are concerned with the pure modeling aspects. At this point, all we need to do is to map the Communication Pattern's components to the architecture units and individually arrange the algorithm's activities (Sequence Diagrams) accordingly. The ExecuteTransfer activity of the algorithm of Fig. 4a is shown in Fig. 5a for the transfer path corresponding to the DMA transaction and in Fig. 5b for the bus transaction. In Fig. 5a the three transfer components are mapped to units Bus1, DMA1 and Bus2, whereas in Fig. 5b they are mapped to units Bus1, Bridge1 and Bus2. In both cases the message exchanged within the activity ExecuteTransfer must be adapted to describe the exact operating mode of the architecture units, e.g., data passes through Bridge1 one sample per time, while 2 samples are stored in DMA1's internal buffer before they can be forwarded to Memory2.

5 Case Study

So far, we have showed to the reader the effectiveness of our works to the sample producer-consumer example of Section 2, demonstrating how our approach and

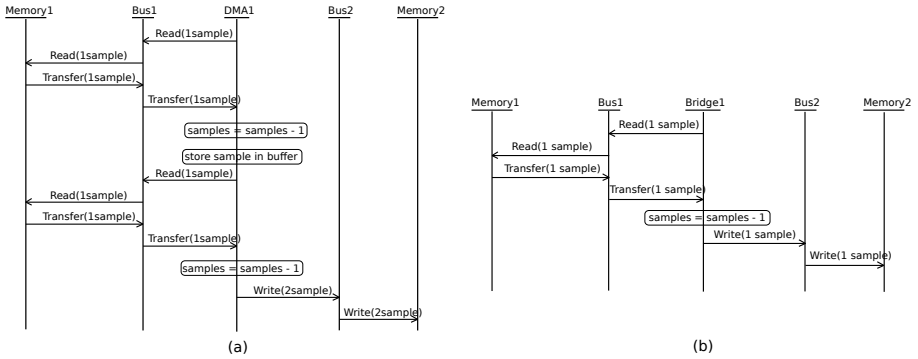

Fig. 5. The Sequence Diagram for the ExecuteTransfer activity of Fig. 4 mapped onto the transfer path Bus1-DMA1-Bus2 (a) and onto the transfer path Bus1-Bridge1-Bus2 (b)

Communication Patterns technically solve what we called the modeling and mapping problems for complex communication schemes. In this section we demonstrate the effectiveness of our approach and of Communication Patterns in the context of a complete system application-architecture, when Design Space Exploration comes into play.

5.1 TTool/DiplodocusDF

As part of our works, we integrated the approach presented in this article in DiplodocusDF [5], a UML Model-Driven Engineering methodology for the design and rapid prototyping of data-dominated applications on heterogeneous real-time embedded systems. To support this extension of DiplodocusDF, we implemented the diagrams needed by Communication Patterns into TTool [2], a toolkit for the edition, simulation and formal verification of UML/SysML diagrams supporting DiplodocusDF. An application model in TTool/DiplodocusDF is implemented as SysML block definition and Instance Diagram, where the behavior of each block is described by a SysML State Machine. An application model describes an algorithm from a functional view, with processing and control tasks interconnected by data and control dependencies. On the other hand, an architecture instance in TTool/DiplodocusDF is described by a UML Deployment Diagram made up of a set of generic interconnected units (e.g., bus, CPU, DMA) decorated with performance parameters. At mapping level an application is projected onto an architecture by respectively associating SysML blocks to nodes in the Deployment Diagram by means of artifacts. We have now integrated the mapping of the Communication Pattern diagrams and components onto a transfer path in the architecture Deployment Diagram as well as the association of a data dependency in the application to a Communication Pattern. Such a mapping has been implemented with an artifact within the architecture Deployment Diagram.

5.2 A parallel application: High Order Cumulants

The application for this case study is a classification algorithm, High Order Cumulants (HOC) as implemented in [6], that is used in cognitive radio by a transmitter to sense the spectrum and detect if another user is currently transmitting in the same frequency range. The SysML Instance Diagram for the application algorithm, as modeled in DiplodocusDF with TTool, is illustrated in Figure 6. For the sake of simplicity, in Fig. 6 the control operations and control dependencies are omitted and only the dataflow view (processing operations and data dependencies) of the model is displayed. The HOC algorithm operates on segments of the input stream (Source) that are independently processed (CWM1, CWM2, SUM, CWS) to extract a score. The occupancy of a specific frequency range is determined by accumulating scores (ACC) over a given classification period and by comparing the accumulated scores with a pre-computed threshold (Sink).

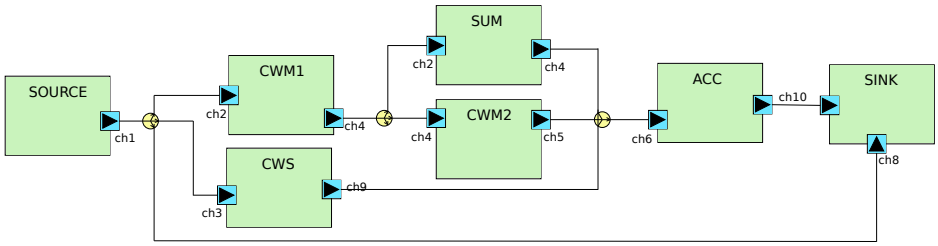

Fig. 6. Dataflow view of the SysML Block Instance Diagram for the HOC application, as modeled in DiplodocusDF with TTool

5.3 A parallel and distributed hardware architecture: Embb

The target hardware architecture for HOC is Embb [7]. Embb is a generic baseband architecture dedicated to signal processing applications. Figure 7a shows the architecture Deployment Diagram of the overall topology, as modeled in DiplodocusDF with TTool. Embb is composed of a processing subsystem and a control subsystem. In the former, left-hand side of Fig. 7a, samples coming from the air are processed in parallel by a distributed set of Digital Signal Processors (DSP1 through DSPn) interconnected by a crossbar (Crossbar). The control subsystem, right-hand side of Fig. 7a is where the control operations of the HOC application are executed. The latter run on a Control Processing Unit (MAINcpu) in charge of configuring and controlling both processing operations performed by the DSPs and the data transfers. The CPU of the control subsystem disposes of a memory unit (MAINmemory) and a bus interconnect (MAINbus). The latter is linked to the processing subsystem via a bridge (Bridge1). Fig. 7b illustrates the internal architecture of a DSP: each unit is equipped with a local control unit

(DSPcpu), a processing core (DSP) and a Direct Memory Access unit (DSPdma) to transfer data in and out of the local memory (DSPmemory).

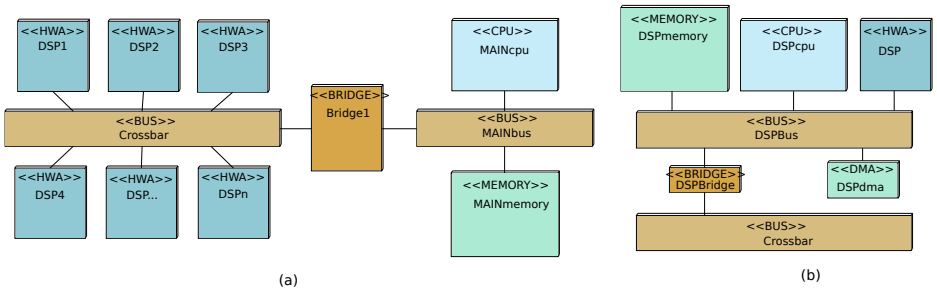

Fig. 7. The Deployment Diagrams of an architecture instance of Embb, a MPSoC platform dedicated to signal processing applications. Part (a) displays a global view of Embb with its processing subsystem (left-hand side) and control subsystem (right-hand side). Part (b) depicts the internal view of each Digital Signal Processor within the processing subsystem

5.4 Design Space Exploration with Communication Patterns

In the application graph of Fig. 6, we apply Communication Patterns to describe the transfers associated to channels ch1, ch2, ch3 and ch8 as the parallelism between CWM1 and CWS allows to describe two mapping scenarios. As a first scenario, we map Sink to the MAINcpu, Source to DSP1 and the pair CWM1,2 to DSP2. Given the topology of the architecture, Sink, Source and CWM1,2 store data respectively in MAINMemory, DSP1memory and DSP2memory. Thus, to move data produced by Source we need a Communication Pattern to model one transfer that serves CWM1,2 (ch2, ch3) and a second transfer that serves Sink (ch8). To do so, the ease of use of Communication Patterns allows us to extend the structure of the Activity Diagram of Fig. 4a with a second transfer as showed in Fig. 8. The latter illustrates two possible transfer algorithms: Fig. 8a models two *simultaneous transfers*, whereas Fig. 8b displays two *sequential transfers*. As a second mapping scenario, we associate CWM1,2 to two different DSP units, namely DSP2 and DSP3 in Fig. 7. Again we can re-use the Communication Patterns of Fig. 8, adapting them to model three independent transfers that each serve CWM1, CWM2 and Sink. Two of the possible transfer algorithms resulting from the combinations of parallel and sequential transfers are illustrated in Fig. 9.

Discussion Due to limits of space in this paper, we do not provide the Sequence Diagrams for the transfer algorithms of Fig. 8 and Fig. 9, nor the post-mapping

Activity Diagrams. Thanks to the separation of concerns between control aspects (Activity Diagrams) and message exchanges (Sequence Diagrams), different mapping alternatives are investigated by re-adapting only Activity Diagrams. This reduces considerably the efforts spent during Design Space Exploration as well as design time and costs. At the beginning of a design, it is inevitable to build from scratch the transfer algorithm, choose the components and arrange both of them according to specific transfer paths selected at mapping phase. However, when different mapping alternatives come out, diagrams can be reused with little changes: only the transfer algorithm in the Activity Diagrams has to be modified. Sequence Diagrams that are specific to transfer paths that have already been explored, are re-used without further modifications.

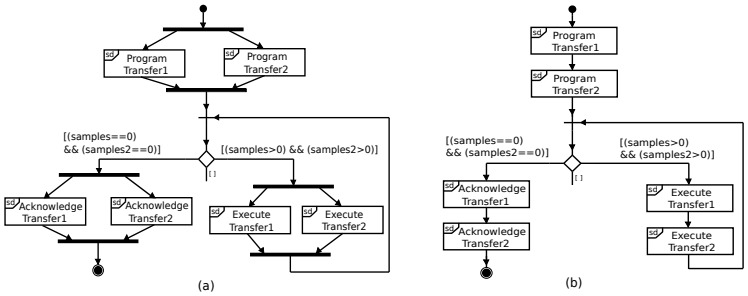

Fig. 8. Sample Activity Diagrams for algorithms modeling two data transfers

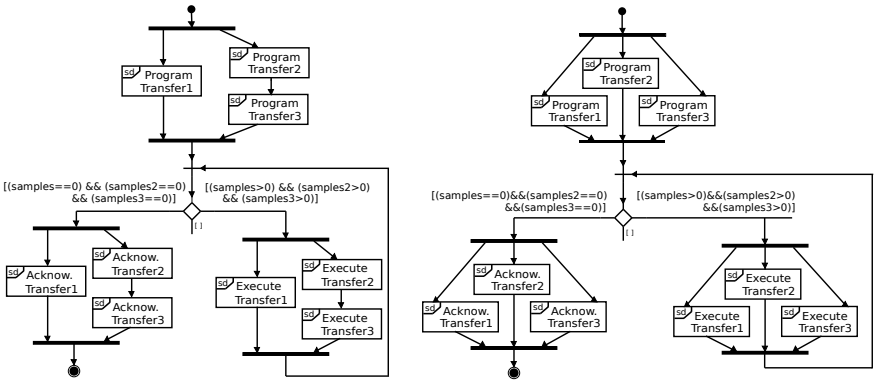

Fig. 9. Sample Activity Diagrams for algorithms modeling three data transfers

6 Related Work

In the literature, the problem of modeling and mapping complex communication schemes is tackled within the larger context of a complete system design. With respect to our contributions we roughly divide existing approaches in two categories: *manual* and *automatic*, based on the way Design Space Exploration is performed. We label as *automatic* an approach where Design Space Exploration is performed by Computer Aided Design (CAD) tools which automatically find a mapping solution and evaluate its performance numbers, from input specifications of a pair application-architecture. In this case, no separation of concerns between application and architecture is needed in input specifications, as most of the DSE efforts are charged to CAD tools. Consequently, input specifications are based on formalisms that can be easily handled by a computer, e.g., dataflow models, process networks. Examples of what we define automatic approaches are Daedalus [8], [9], Metropolis [10], Ptolemy [11], PeaCE/HoPES [12], SCE [13], SystemCoDesigner[14], DOL [15].
We call *manual* those approaches where it is up to the user to manually define a mapping solution whose performance numbers are then analyzed by CAD tools, given a pair application-architecture. Within this category we find works based on UML/MARTE such as GASPARD [16], MOPCOM [17], Koski [18] and [19], [20] dedicated to both hardware and software synthesis. These approaches rely on a refinement process that progressively lowers the level of abstraction of input models. However, such a refinement does not completely separate the application (software synthesis) or the architecture (hardware synthesis) models from the communications, as we defined in Fig. 3. MARTE [4] shares many commonalities with our approach, in terms of the capacity to separately model communications from the pair application-architecture. For such a purpose, MARTE proposes Behavior Scenarios and Steps (Communication Steps). However, these assets are designed for performance and timing analysis, rather than DSE. Consequently, they intrinsically lack a separation between control aspects and message exchanges as we proposed in Activity and Sequence Diagrams. MARTE does not integrate a systematic methodology for DSE and does not define the necessary abstraction levels, proposing only a distinction between logical and physical level. In the context of our Communication Patterns, these levels of abstraction have been the subject of a previous publication [21].
With respect to the above classification, we can place TTool/DiplodocusDF in the category of the manual approaches as it is up to the user to define a Communication Pattern. Performance analysis can be automated in TTool/DiplodocusDF by means of scripting facilities but it is not comparable to the solutions proposed by the above *automatic* approaches.
Independently of the works we presented, in the past edition of MODELS Arkin et al. [22] proposed a model-driven approach and tool, to automate the mapping of parallel algorithms onto parallel platforms. Interestingly enough, the authors introduce their definition of a Communication Pattern to describe the dynamic behavior of the nodes of a parallel platform via communication paths made up of a pair source-destination nodes and a route between the two. Their Commu-

nication Patterns target larger systems and are presented in the frame of an approach where separate steps define the architecture, communication and application similarly to Fig. 3. From the description available in [22] the aim and context of their Communication Patterns is clear and similar to ours. However, with respect to our works, it is not clear what the effective expressive power of such Communication Patterns is, what can be exactly represented in terms of architecture units, transfer algorithm and how they are employed during DSE.

7 Conclusion

In this paper we have provided a systematic approach and its implementation to separately model and map communications from a pair application-architecture, in the frame of the Y-chart approach. In response to the modeling problem, we introduced Communication Patterns and their implementation with UML/SysML modeling diagrams. In the latter, we further introduced an additional separation of concerns between control aspects (Activity Diagrams) and message exchanges (Sequence Diagrams). In response to the second problem, we defined the mapping of data dependencies in the application onto transfer paths in the architecture. We illustrated our solution, first, by means of a simple producer-consumer example and secondly, within the context of a complete application (HOC) and architecture (Embb). Moreover, we provided an implementation of the overall approach we propose in TTool.
Although we have applied Communication Patterns to a MPSoC architecture and a signal processing application, we believe that our contribution is general. We believe it can be applied to other data-dominated applications (e.g., video and image processing) and to other types of distributed architectures (e.g., automotive). In the approach we presented, we have proposed that input specification are manually modeled and mapped. Such a manual approach may constraint the applicability of our solution to systems with a limited number of components. Nevertheless, it is our intuition that our Communication Patterns may scale well also for larger systems (i.e., hundreds of components) via the creation of libraries of communication models.
In our future works we will focus on generating application code from automatic transformation of models that result from the approach we proposed in this paper. Additionally, we will complete the implementation of our approach by extending the simulator of TTool/DiplodocusDF with the support for performance analysis with Communication Patterns.

Acknowledgements. The research leading to these results has been conducted in the framework of the Celtic-Plus project SPECTRA (CP07-013) which has been partially funded by the French "Direction générale de la compétitivité, de l'industrie et des services (DGCIS)", the "Ministry of Finance and Economy / Business Development Agency" of the Monaco Principality and the "AVANZA2" framework of Spanish Industry, Tourism and Commerce Ministry (Ministerio de Industria, Turismo y Comercio).

References

1. Schmidt, D., C.: Model-Driven Engineering, In: IEEE Computer 39(2), (2006)
2. TTool, http://ttool.telecom-paristech.fr
3. Kienhuis, B., Deprettere, E., F., van der Wolf, P., Vissers, K.: A Methodology to Design Programmable Embedded Systems - The Y-chart Approach. In: Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS, pp. 18-37 (2002)
4. Object Management Group. A UML profile for MARTE (2014), http://www.omgmarte.org
5. Gonzalez Pina, J. M.: Application Modeling and Software Architectures for the Software Defined Radio. PhD Dissertation, Telecom ParisTech (2013)
6. SACRA, Spectrum and Energy efficiency through multi-band Cognitive Radio: D6.3, Report on the Implementation of selected algorithms. http://www.ict-sacra.eu/public_deliverables/.
7. Muhammad, N.-u.-I., Rasheed, R., Pacalet, R., Knopp, R., Khalfallah, K.: Flexible Baseband Architectures for Future Wireless Systems. In: EUROMICRO Digital System Design, pp. 39-46. (2008)
8. Thompson, M., Nikolov, H., Stefanov, T., Pimentel, A.D., Erbas, C., Polstra, S., Deprettere, E.F.: A Framework for rapid system-level exploration, synthesis and programming for multimedia MP-SoCs. In: CODES-ISSS, pp. 9-14. (2007)
9. Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A., Polstra, S., Bose, R., Zissulescu, C., Deprettere, E.: Daedalus: Toward composable multimedia MP-SoC design. In: Design Automation Conference (DAC), pp. 574-579. (2008)
10. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.: Metropolis: An integrated electronic system design environment. IEEE Computer, 36(4), 45-52 (2003)
11. The Ptolemy Project (2014), http://ptolemy.eecs.berkeley.edu
12. Ha, S., Kim, S., Lee, C., Yi, Y., Kwon, S., Joo, Y.-P.: PeaCE: A hardware-software codesign environment for multimedia embedded systems. ACM Transactions on Design Automation of Electronic Systems, 12(3), 1-25 (2007)
13. Dmer, R., Gerstlauer, A., Peng, J., Shin, D., Cai, L., Yu, H., Abdi S., Gajski D.: System-on-chip environment: A SpecC-based framework for heterogeneous MPSoC design. EURASIP Journal on Embedded Systems, 2008(3), 1-13 (2008)
14. Keinert, K., Streubhobar, M., Schlichter, T., Falk, T., Gladigau, J., Haubelt, C., Teich, J., Meredith, M.: SystemCoDesigner - An automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. ACM Transactions on Design Automation of Electronic Systems, 14(1), 1-23 (2009)
15. Thiele, L., Bacivarov, I., Haid, W., Huang, K.: Mapping Applications to Tiled Multiprocessor Embedded Systems. In: 7th International Conference on Application of Concurrency to System Design (ACSD), pp. 29-40. (2007)
16. Gamatie, A., Le Beux, S., Piel, E., Ben Atitallah, R., Etien, A., Marquet, P., Dekeyser, J.L.: A Model-Driven Design Framework for Massively Parallel Embedded Systems. ACM Transactions on Embedded Computing Systems 10(4), 1-36 (2011)
17. Lecomte, S., Guillouard, S., Moy, C., Leray, P., Soulard, P.: A co-design methodology based on model driven architecture for real time embedded systems. Mathematical and Computer Modelling 53(3-4), 471-484 (2011)
18. Kangas, T., Kukkala, P., Orsila, H., Salminen, E., Hnnikinen, M., Hmlinen, T.D.: UML-based multiprocessor SoC design framework. ACM Transactions on Embeded-ded Computing Systems 5(2), 281-320 (2006)

19. Vidal, J., de Lamotte, F., Gogniat, G., Soulard, P., Diguet, J.-P.: A co-design approach for embedded system modeling and code generation with UML and MARTE. In: Design and Automation Test in Europe (DATE), pp. 226-231 (2009)
20. Vidal, J., de Lamotte, F., Gogniat, G., Diguet, J.-P., Soulard, P.: UML design for dynamically reconfigurable multiprocessor embedded systems. In: Design and Automation Test in Europe (DATE), pp. 1195-1200. (2010)
21. Enrici, A., Apvrille, L., Pacalet, R.: Communication Patterns: a Novel Modeling Approach for Software Defined Radio Systems. In: 4th International Conference on Advances in Cognitive Radio (COCORA), pp. 35-40. (2014)
22. Arkin, E., Tekinerdogan, B., Imre, K., M.: Model-Driven Approach for Supporting the Mapping of Parallel Algorithms to Parallel Computing Platforms. In: MODELS, pp. 757-773 (2013)