A Model-Driven Engineering Methodology to Design Parallel and Distributed Embedded Systems

ANDREA ENRICI, LUDOVIC APVRILLE, RENAUD PACALET,
LTCI, CNRS, Telecom ParisTech, Université Paris-Saclay, 06904 Biot Sophia Antipolis, France

In MDE system-level approaches, the design of communication protocols and patterns is subject to the design of processing operations (computations) and to their mapping onto execution resources. However, this strategy allows to capture simple communication schemes (e.g., processor-bus-memory) and prevents to evaluate the performance of both computations and communications (e.g., impact of application traffic patterns onto the communication interconnect) in a single step. To solve these issues we introduce a novel design approach - the Ψ-chart - where we design communication patterns and protocols independently of a system's functionality and resources, via dedicated models. At the mapping step, both application and communication models are bound to the platform resources and transformed to explore design alternatives for both computations and communications. We present the Ψ-chart and its implementation (i.e., communication models and Design Space Exploration) in TTool/DIPLODOCUS, a UML/SysML framework for the modeling, simulation, formal verification and automatic code generation of data-flow embedded systems. The effectiveness of our solution in terms of better design quality (e.g., portability, time) is demonstrated with the design of the physical layer of a ZigBee (IEEE 802.15.4) transmitter onto a multi-processor architecture.

CCS Concepts: •**Computer systems organization** → **Embedded systems;** •**Hardware** → **Methodologies for EDA; Software tools for EDA;**

General Terms: Design, Experimentation

Additional Key Words and Phrases: Model Driven Engineering, Hardware/Software Co-Design, Design Space Exploration, UML, SysML

ACM Reference Format:
Andrea Enrici, Ludovic Apvrille, Renaud Pacalet, 2015. A Model-Driven Engineering Methodology to Design Parallel and Distributed Embedded Systems - the Ψ-chart approach. *ACM Trans. Embedd. Comput. Syst.* V, N, Article XXXX (XXXX 2015), 25 pages.
DOI: 0000001.0000001

1. INTRODUCTION AND PROBLEM STATEMENT

Today's embedded systems are more and more realized with architectures where the processing operations, i.e., data and control information, are executed in parallel over a network of interconnected subsystems (e.g., Multi-Processors Systems on Chip, MP-SoC, electronic equipments in automotive and avionic systems). The performance, cost and time-to-market of these systems is not only driven by the design of data-processing operations but also by the design of data-transfer operations. Therefore, it is of utmost importance to account for the design of these data-transfer operations, *communications*, in the early phases of a design process.
Since the late nineties, the Y-chart [Kienhuis et al. 1997; Kienhuis et al. 2002], Fig. 1, is one of the dominant Model Driven Engineering (MDE) approaches that guides the

The research work presented in this paper complements the work first published in [Enrici et al. 2014]. The research work leading to this paper has been supported by the French FUI project NETCOM under grant agreement no. F1405046 U.
Author's addresses: A. Enrici, L. Apvrille, R. Pacalet, System-on-Chip laboratory at Institut EURECOM, Campus SophiaTech, 450 Route des Chappes, 06904 Biot - Sophia Antipolis cedex, FRANCE.

DOI: 0000001.0000001

automation of design and Design Space Exploration of embedded systems for data-dominated applications. In this approach, communications are typically described both in an application model (i.e., the system's functionality, box 1.1 in Fig. 1) and in an architecture/platform model (i.e., the system's resources, box 1.2 in Fig. 1). In the application model, communications are represented in the form of logical dependencies between computations (e.g., channels, events). In the architecture (platform) model, communications are described in the form of the services offered by hardware/software resources (e.g. CPU and its Operating System, DMA engines, buses). A design is then evaluated (Design Space Exploration, box 3 in Fig. 1) based on a mapping model (box 2 in Fig. 1) that captures a selection of the architecture resources that execute the functionality of the application model.
However, when creating a mapping model it is frequent to incur into a *communication mismatch* between the description of communications contained in the application and in the architecture models. This is due to the mismatch between the primitives and operational semantics used to describe communications in the Model of Computation, MoC, of the application (e.g., point-to-point data channels with blocking read()/write() operations) and those in the Model of Computation of the architecture (e.g., configuring and executing a DMA data transfer or a series of bus transactions).

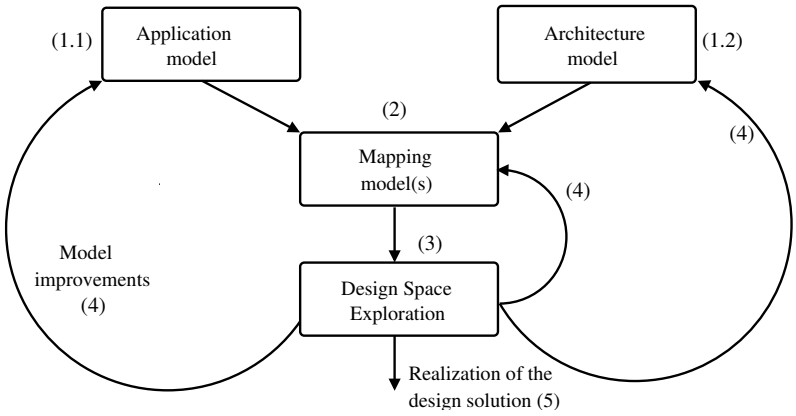

Fig. 1. The Y-chart approach for the design of programmable embedded systems

Several implementations of the Y-chart approach exist that differ in terms of the semantics of the modeling language used to specify a system and or in terms of the purposes and techniques of the DSE phase. Nevertheless, this communication mismatch still remains an open issue. Typically, it is circumvented by designing communications *after* mapping the application onto the architecture, as the mapping of communications depends on the mapping of computations. However, this strategy heavily impacts the modeling phase (i.e., portability) and the number of iterations (Model Improvements in Fig.1) that occur after DSE when a mapping configuration does not satisfy the design requirements. Communications that are modeled after the mapping of computations cannot be ported to other target platforms, without additional re-design steps that are time consuming and error-prone (labels 4 in Fig. 1). The DSE of communications occurring after the DSE of computations, local optima are much more likely to be found as a comprehensive view of all design constraints is missing. This further increases the number of iterations in the Model Improvements phase (step 4 in Fig. 1).
To solve the communication mismatch in the frame of UML/SysML-based design, we proposed a novel design approach: the Ψ-chart approach [Enrici et al. 2014], where

communication protocols and patterns are modeled independently of the application and architecture models, before mapping. In this publication, we present a complete description of the following research work:

(1) A description of the Ψ-chart design approach, regardless of the specific semantics issues of UML and SysML.
(2) A complete implementation of the Ψ-chart approach in TTool/DIPLODOCUS, a UML/SysML toolkit for the hardware/software co-design of data-dominated embedded systems, in terms of:
  (2.1) The syntax and semantics of dedicated models for communication protocols called Communication Patterns.
  (2.2) The model transformations that integrate Communication Patterns into the Design Space Exploration facilities offered by TTool/DIPLODOCUS. These transformations target the simulation, formal verification and rapid prototyping via the automatic generation of executable control code (i.e., the code that configures and triggers the platform units onto which computations and communications are mapped). The goal of these model transformations is to simplify the design space that results by simultaneously accounting for communication and computation related constraints.

Section 2 provides a complete description of the Ψ-chart design approach and Section 3 introduces TTool/DIPLODOCUS. Section 4 and Section 5 describe our implementation of the Ψ-chart approach in TTool/DIPLODOCUS: first we illustrate our UML/SysML models for communication protocols and secondly we describe the model transformations. Section 6 shows the Ψ-chart design of the physical layer of a ZigBee transmitter in TTool/DIPLODOCUS and compares it with design in the Y-chart implementation of TTool/DIPLODOCUS. Section 7 discusses our contributions with respect to related work and Section 8 concludes the paper.

2. A NOVEL DESIGN APPROACH

The Ψ-chart, Fig. 2, is an extension of the Y-chart, Fig. 1, where a third input is added to capture communication protocols and patterns independently of the description of communications that is present in the application and architecture models. This design paradigm is based on the following considerations.

A communication protocol can be defined as a set of rules for the exchange of information between abstract components (e.g., master, slave, controller). These rules are specified regardless of the particular characteristics of an implementation of these abstract components (e.g., a master being an ARM CPU or an Intel CPU). Therefore, a communication protocol can be modeled regardless of the specific resources of a platform model. A communication protocol is also specified regardless of the algorithm and the processing operations that produce/consume the information to transfer (e.g., Fast Fourier Transforms, vector operations). Therefore, a communication protocol can be described independently of an application model.

From the viewpoint of the total design time, the Ψ-chart reduces the number of iterations that occur when, after DSE, a mapping solution results not to satisfy the desired requirements. Many iterations are, in fact, caused by the re-design of the communications that occurs as a consequence of the communication mismatch, after the mapping of computations. This pitfall is avoided in the Ψ-chart by positioning the design of communications before the mapping phase.

From the viewpoint of the Design Space Exploration phase, the Ψ-chart allows to jointly evaluate the performance of both communications and computations, without the need to decouple their exploration in several steps. In the Ψ-chart, the complexity of the design space increases with respect to the Y-chart approach. In the latter case,

because of the communication mismatch, it is common to separate the design (i.e., model and explore) of a system in two steps: first computations and then communications. This leads to situations where only partial architectural trade-offs (i.e., local optima) can be evaluated, that do not account for a global performance impact of both communications and computations.

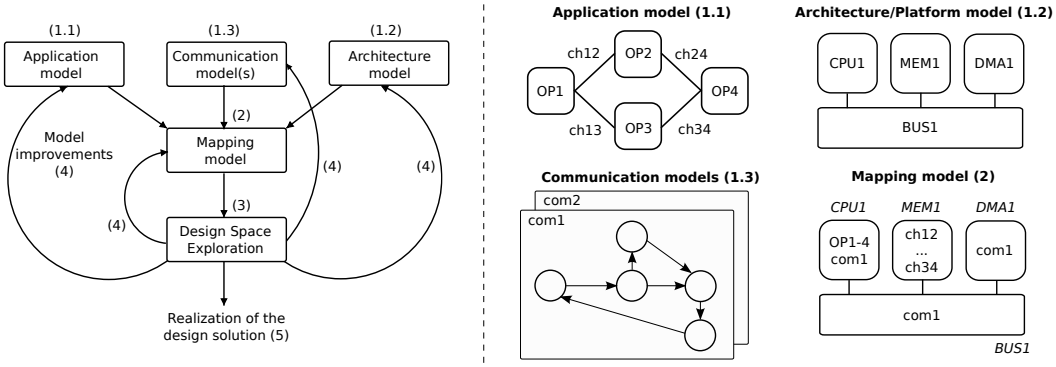

Fig. 2. The Ψ-chart approach (left side) and a graphical visualization of its constituent models (right side).

The Ψ-chart approach is described by the following enumerated steps, where each number points to the corresponding label in Fig. 2. In spite of the ordering associated by the numbers, steps 1.1-1.3 can be conducted in any order as the corresponding models are created independently of each other. On the contrary, the numbering of steps 2-5 coincides with the order dependencies of the design flow.

1.1 Application: a model of the system's functionality (e.g., a video-compression algorithm). This model must be created regardless the resources that are available for execution purposes (i.e., hardware or software resources) and must express all potential parallelism between operations. This model must express both the processing of information (e.g., computations) and the dependencies (e.g., communications) between these processing operations.

1.2 Architecture or platform: a model of the resources (e.g., bus, CPU, memory, middleware, OS) to support the execution of the applications' functionality. This model must express the topology of the available hardware/software resources, the services that these resources offer (e.g., a bus transaction, and operating system call) as well as their costs (e.g., in terms of silicon area, power consumption, computational power).

1.3 Communication: a model describing the communication protocols and patterns used/needed by the target platform to transfer information between processing operations. Such protocols must be expressed independently of the semantics associated with the dependencies between computations in the application model. Similarly, they must also be expressed independently of the specific semantics of the services offered by the platform model's resources and services (e.g., bus, CPU, memory, middleware, Operating System).

2 Couple each application and communication models to the architecture model (mapping). In this step, each entity requesting a processing service in the application models (e.g., a data-processing operation, a control task) is associated to a resource providing the corresponding service in the architecture model (e.g., an

Application Specific Integrated Circuit, a general-purpose CPU and its Operating System). Also, a dependency between processing entities of the application model is associated to a communication model. Subsequently, each entity requesting a service (e.g., configure a transfer) in the communication models (e.g., a master) is mapped onto a resource providing the corresponding service in the architecture model (e.g., a DMA controller, a CPU).

3 Explore the design space (e.g., via simulation, formal verification) by associating performance numbers to the mapping model and evaluate the compliance of the mapping model to a set of pre-defined requirements (e.g., silicon area, power consumption, computational power).

4 In case the mapping model does not provide the desired performance, the system models are modified in order to find alternative solutions (labels number 4 in Fig. 2). In the architecture model, existing resources are re-structured or new resources are introduced. The application and communication models are arranged so as to express the same functional behavior in a different way. In the mapping model, new associations between the application, the communications and the architecture are explored.

5 The above steps are repeated iteratively until a solution is found that satisfies all design requirements. At this point, the resulting design is passed to the implementation teams that realize it in terms of hardware and software components.

We further underline here that the Ψ-chart is a generic approach that defines a set of design guidelines (i.e., separation of concerns between application, communication and platform models) regardless of the specific semantics of the modeling language (e.g., state machines, data-flow MoCs, UML/SysML) and the Design Space Exploration techniques (e.g., simulation, formal verification) used by a given design framework.

2.1. The mapping methodology

Fig. 3 details the ordered steps that are necessary to create a complete mapping model (step 2 in Fig. 2) in the Ψ-chart approach. We specify that Fig. 3 shows how to *bind* information that is expressed in separate application, communication and platform models.

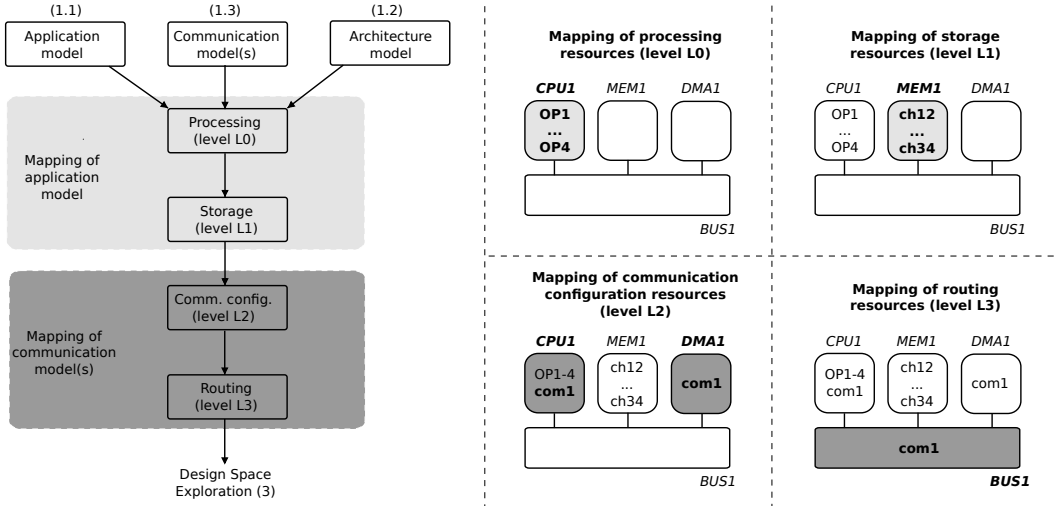

Fig. 3. The mapping methodology of the Ψ-chart (left side) and the models for each step (right side).

Computation (level L0). The computational parts of the application model are mapped to the platform model. For instance, a node in the application MoC that models an FFT operation is mapped to a Digital Signal Processor (DSP) unit; a node modeling a control task is mapped to a Central Processing Unit (CPU). Similarly, for variables which are dependent on the specific characteristics of the mapped unit, a value is assigned accordingly. For instance, variables describing abstract data types (e.g., complex numbers) are assigned a value (e.g., `cpx32` in case the mapped DSP unit represents complex numbers with 16 bits for the imaginary part and 16 bits for the real part).
Storage (level L1). Any behavior and variable in the application model that is related to the storage of data or control information is mapped. A system engineer selects the architecture units (e.g., memories, buffers) that will store the data and/or control information produced or consumed by the computations mapped at level L0. According to the selected units and their characteristics, parameters are assigned a value, e.g., the size of a buffer.
At this point, data dependencies in the application model must be associated to the communication models that describe the corresponding transfer of data. In our implementation of the Ψ-chart in TTool/DIPLODOCUS this is performed by the user who explicitly associates a data channel between computations to a Communication Pattern (Section 4). We specify that this solution is not imposed by the design principles of the Ψ-chart approach. Other types of relations may be deployed according to the characteristics of the specific design framework into which the Ψ-chart is implemented (e.g., matching signature operations).
Communication configuration (level L2). The behavior and parameters of a communication model are mapped to the platform model. A system engineer selects the architecture units that will be in charge of configuring the data transfers that move data from the source to the destination storage units mapped at level L1.
Routing (level L3). The route that data will take to be transferred between a source and a destination storage is chosen in terms of transfer units (e.g., bus, bridge) according to the topology of the architecture model.

3. AN OVERVIEW OF DESIGN IN TTOOL/DIPLODOCUS

TTool/DIPLODOCUS [Apvrille et al. 2006; Apvrille 2008] is a UML/SysML framework for the hardware/software co-design of data-dominated embedded systems.
In TTool/DIPLODOCUS, an *application* model is denoted with SysML Block Definition and Block Instance diagrams that are composed by a set of blocks interconnected by data and control dependencies via ports and channels. The internal behavior of each block is described by a SysML Activity Diagram. An application is described in terms of the two following abstraction principles:

— *Data abstraction:* only the amount of data exchanged between application blocks is modeled. Internal decisions that depend on the value of data are expressed in terms of non-deterministic and static operators (i.e., conditional choice based on the value of a random variable).
— *Functional abstraction:* algorithms are described using abstract cost operators that express the complexity of processing data in terms of the number of operations required to execute them (e.g., number of integer operations).

A *platform* model is denoted using a UML Deployment Diagram that represents a set of interconnected resources, e.g., bus, CPU and its operating system, DMA, memory. These resources are characterized by performance parameters (e.g., the scheduling policy and the number of cores for a CPU) that are used for DSE (e.g., simulation, formal verification) and by implementation characteristics (e.g., addresses of memory areas) that are used for rapid prototyping (i.e., control code synthesis). A *mapping* model is

created from an instance of the platform model where dedicated UML artifacts are added to map the computations and their dependencies. The abstract cost operators are assigned a value according to the performance characteristics (e.g., operating frequency) of the platform's units. TTool/DIPLODOCUS allows a user to map functions that belong to different functional views, namely from different application models.
Design Space Exploration in TTool/DIPLODOCUS evaluates the performance of a mapping solution by simulating the workload of computations and data-transfers [Knorreck 2011]. A formal verification engine [Knorreck 2011] is also available to verify system properties (e.g., liveness, reachability, scheduling). DSE can be performed both manually via the tool's GUI or automatically via a set of scripts that configure the DSE engine to evaluate different mapping alternatives.
The above abstraction principles have been defined as TTool/DIPLODOCUS targets early design and DSE, when not all the details about a system's application (e.g., value and type of data) and platform (e.g., Operating System, size and policy of cache memories for a CPU) are known. The validation of the effectiveness of these abstractions has been described in [Jaber 2011], where TTool/DIPLODOCUS was used for the design of the physical layer of a LTE base station jointly with Freescale Semiconductors. The resulting design in TTool/DIPLODOCUS lead to performance results that differed by only 10% with respect to the final implementation. To obtain these performance figures, design in TTool/DIPLODOCUS required only a few weeks, whereas manual development of a functionally equivalent system amounted to 6 months.
In the following, we describe our implementation of the Ψ-chart in TTool/DIPLODOCUS.

4. COMMUNICATION MODELS IN TTOOL/DIPLODOCUS: COMMUNICATION PATTERNS

The communication models of the Ψ-chart that we implemented for TTool/DIPLODOCUS are called Communication Patterns (CPs). These models describe communication protocols at the datalink layer of the ISO/OSI reference model [Zimmermann 1980]. CPs are deployed to model communication protocols at Electronic System-Level of abstraction. Therefore, CPs do not consider the effects of caching on communications (e.g., the transfers between a cache and main memory due to a cache miss) that occur at a micro-architecture level of abstraction. Caching effects are abstracted in the timing attributes of a generic CPU block in the platform model. An attribute, called *cache-miss ratio*, is used by the DSE engine of TTool/DIPLODOCUS as a penalty that is associated to each read/write operation between generic CPUs and memory blocks of the platform model.
A Communication Pattern describes the *behavior* of a communication protocol, intended as a set of rules for the exchange of data between *components* of an embedded system. A component is intended as a generic architecture unit, regardless its implementation, i.e., hardware, software or both.
The following tuple provides a formal description of the UML/SysML diagrams and operators that compose a Communication Pattern.

$$\mathcal{CP} = (\mathcal{M}_{CP}, \mathcal{AD}_{CP}, \mathcal{SD}_{CP})$$

— $\mathcal{M}_{CP}$ is the main Activity Diagram (interface) of a Communication Pattern $\mathcal{CP}$
— $\mathcal{AD}_{CP}$ is the set of Activity Diagrams that are referenced in the entire Communication Pattern $\mathcal{CP}$
— $\mathcal{SD}_{CP}$ is the set of Sequence Diagrams that are referenced in the entire Communication Pattern $\mathcal{CP}$

An Activity Diagram $\mathcal{AD}$ is defined as the following tuple:

$$\mathcal{AD} = (\mathcal{R}_{SD}, \mathcal{R}_{AD}, \mathcal{C}_{op}, N, L)$$

— $\mathcal{R}_{SD}$ is the set of Sequence Diagrams referenced by $\mathcal{AD}$. A reference to a Sequence Diagram $r \in \mathcal{R}_{SD}$ is considered as a node $n \in N$ that has one incoming and one outgoing edges.
— $\mathcal{R}_{AD}$ is the set of Activity Diagrams referenced by $\mathcal{AD}$. A reference to an Activity Diagram $r \in \mathcal{R}_{SD}$ is considered as a node $n \in N$ that has one incoming and one outgoing edges.
— $\mathcal{C}_{op}$ is the set of control operators that are used to compose the references to other diagrams. A control operator $c \in \mathcal{C}_{AD}$ can be of type $c \in \{parallelism, sequence, choice, iteration, start, final\}$. $start$ is the start node (symbol) of the Activity Diagram. $final$ is the end node (symbol) of a *path* (defined in Property 5) within an Activity Diagram.
— N is the set of nodes that compose the Activity Diagram. A node $n \in N$ is either a reference to a diagram $r \in \{\mathcal{R}_{SD}, \mathcal{R}_{AD}\}$ or a control operator $c \in \mathcal{C}_{op}$.
— L is a set of links (edges). Each link $l \in L$ interconnects a pair of nodes $n_1, n_2 \in N$ with the following notation:

$$l_{n_1,n_2} = n_1 \rightarrow n_2$$

A Sequence Diagram $\mathcal{SD}$ is defined as the following tuple:

$$\mathcal{SD} = (\mathcal{I}_{SD}, E, \mathcal{M}_{SD}, \mathcal{A}_{SD}, \prec, \mathcal{V}_{I_{SD}})$$

— $\mathcal{I}_{SD}$ is the set of instances that are used to describe the components of a communication protocol. An instance $i \in \mathcal{I}_{SD}$ can be of type $i \in \{controller, transfer, storage\}$.
— E_i is a set of events that compose the lifeline of an instance $i \in \mathcal{I}_{SD}$. Each event $e \in E$ can be one of type $e \in \{SND_m, RCV_m, ACT_a\}$, where:
  — SND_m is the dispatch (send) of a message $m \in \mathcal{M}_{SD}$
  — RCV_m is the reception of a message $m \in \mathcal{M}_{SD}$
  — ACT_a is the occurrence of an action $a \in \mathcal{A}_{SD}$
— $\mathcal{M}_{SD}$ is the set of parameterized messages that are exchanged by instances $\mathcal{I}_{SD}$. A message $m \in \mathcal{M}_{SD}$ is part of a library that is composed of messages $m \in \{Read(), Write(), TransferRequest(), TransferTerminated()\}$. To ease the transformation of CPs (Section 5), we currently consider only synchronous messages in $\mathcal{M}_{SD}$. We envisage to extend the semantics of messages to the asynchronous case in our future work.
— $\mathcal{A}_{SD}$ are the actions performed by instances $\mathcal{I}_{SD}$ of type $controller$ on variables $\mathcal{V}_{I_{SD}}$ of a Sequence Diagram $s \in \mathcal{SD}$. As part of these actions, a parameterized timing function called $wait()$ is also available.
— $\prec$ is a total order relation of events $e \in E_i$.
— $\mathcal{V}_{I_{SD}}$ is a set of user attributes (variables) of an instance $i \in \mathcal{I}_{SD}$. A variable $v \in \mathcal{V}_{I_{CP}}$ can be of type $v \in \{integer, boolean, address\}$. It can be assigned a value in Sequence Diagrams $s \in \mathcal{SD}$. This value is of type read-only in the guards of the $choice$ control operator, within an Activity Diagram.

We used two types of diagrams to enforce a separation of concerns between the general algorithm of a communication protocol (Activity Diagrams) as well as its data and event exchanges (Sequence Diagrams). This allows CPs to be more modular, easy to (re-)use and to port thus reducing the number of iterations that occur when models are modified (labels 4 in Fig. 2) as opposed to the case where only one type of diagram is used. In a design based on the Ψ-chart, Communication Patterns are instantiated from a library. This improves the scalability of a design as it limits the number of CPs that must be instantiated for each data dependency of the application model (this number can be large in applications such as 5G signal-processing algorithms). Each

time that data are transferred with a CP that has already been deployed, that CP must only be re-mapped and must not be re-instantiated.
To the best of our knowledge, Message Sequence Charts (MSCs) [Reniers 1999] are the closest models to CPs that are currently in use to capture communication protocols. A MSC is an interaction diagram that provides a language to specify the communication behavior between a set of system components and their surrounding environment via the exchange of messages. UML 2.0 Sequence Diagrams are inspired by MSCs.

4.1. Semantics properties for well-formed CPs

To transform a Communication Pattern for DSE and prototyping (Section 5), we specified the following semantic properties that define a *well-formed* CP:

(1) **Property 1. Non-modeling of returned data:** data that is returned upon the reception of a message is not modeled as we assume it to be implicit. For instance, data that is returned upon the issue of a $Read()$ message is not modeled.
(2) **Property 2. Access to attributes:** variables in $\mathcal{V}_{I_{SD}}$ are read-only in Activity Diagrams. They are initialized in the Sequence Diagrams $\mathcal{SD}$ and their value can be changed by actions $\mathcal{A}_{SD}$ only. Only variables of type int and $boolean$ can be used to govern the execution of control operators $\mathcal{C}_{AD}$. Variables of type $address$ are used for automatic generation of the executable system control code.
(3) **Property 3. Active instances:** instances of type $controller$ are the only type of active instances. Both $controller$ and $transfer$ instances are allowed to both send and receive messages. However, $transfer$ instances are only allowed to forward incoming messages. Instances of type $storage$ are only allowed to receive messages.
(4) **Property 4. Starting diagram:** in any Activity Diagram of a Communication Pattern, the starting symbol must always be followed by a reference to a non-empty Sequence Diagram.
(5) **Property 5. Path:** we define a (generic) path as a set of interconnected nodes that terminate with the $final$ node, as follows:

$$path = (n_1,\ l_{n_1,n_2}),\ \{(n_i,\ l_{n_i,n_{i+1}})^{+}_{i:2\rightarrow m-1}\},\ (n_m,\ l_{m,final},\ final),\ with\ n \in N, l \in L$$

(6) **Property 6. Complete path:** in any Activity Diagram of a Communication Pattern, a continuous path must interconnect the $start$ and the $final$ nodes via a set of control operators and references to diagrams in N and links $l \in L$. It is defined as follows:

$$complete\_path = (start,\ l_{start,n_1}),\ \{(n_i,\ l_{n_i,n_{i+1}})^{+}_{i:1\rightarrow m-1}\},\ (n_m,\ l_{m,final},\ final),\ with\ n \in N, l \in L$$

where the repetition operator $(...)^{+}$ defines one or multiple occurrences of the content enclosed by the parenthesis. The above path can be developed as:

$$complete\_path = \{start,\ l_{start,n_1},\ n_1,\ l_{n_1,n_2},\ n_2,\ l_{n_2,n_3},\ ...,\ l_{n_m,final},\ final\}$$

where m defines the length of the path in terms of interconnected nodes.
(7) **Property 7. Parallelism control operator:** the parallelism operator is considered as a single node $n \in N$. It is composed of a fork and a join bars that are used as delimiters to, respectively, fork and join the execution of k branches of sequentially interconnected diagrams.

$$parallelism = fork\ \{(l_{fork,n_1},\ n_1)^t\ (l_{n_i,n_{i+1}},\ n_{i+1})^t_{i:1\rightarrow m}\ (n_m,\ l_{n_m,join})^t\}^{t:1\rightarrow k}\ join$$

Above, index t is used to label each parallel interconnected branch.
(8) **Property 8. Choice control operator:** the choice control operator is composed of one incoming and k outgoing edges, one for each outgoing branch. Each outgoing

branch is labeled by a $guard_k$ that specifies a boolean condition for the corresponding branch to be executed.

$$choice = \{guard_i,\ path_i\}_{i:1 \to k}$$

The empty boolean condition is a valid guard that is always evaluated as true.

(9) **Property 9. Sequence control operator:** the sequence control operator is given by the simple interconnection of two or more references to diagrams in $\mathcal{R}_{SD}$ and $\mathcal{R}_{AD}$. We formally define a sequence path as follows:

$$path = (n_1,\ l_{n_1,n_2}),\ \{(n_i,\ l_{n_i,n_{i+1}})^{+}_{i:2 \to m-2}\},\ (n_{m-1},\ l_{n_{m-1},n_m},\ n_m)$$
$$l \in L, n \in \{R_{SD}, R_{AD}\}$$

(10) **Property 10. Iteration control operator:** the iteration control operator (i.e., a for-loop) has one incoming and two outgoing edges. These two outgoing edges are connected to the body of the loop and to the exit branch. These two branches both constitute a path. A condition is composed of a set of 3 clauses, as in standard for-loops: an initialization, a stop condition and an increment.

$$iteration = condition,\ \{branch_{exit},\ branch_{body}\}$$
$$branch_{exit},\ branch_{body} = path$$
$$condition = initialization;\ stop\_condition;\ increment$$

(11) **Property 11. No recursion:** recursion is not allowed for any outgoing branch of any control operator $c \in \mathcal{C}_{op}$.

(12) **Property 12. Start and final nodes:** an Activity Diagram is allowed to contain one and only one start node. It must contain at least one final node. Moreover, the following links are not valid connections between nodes:

$$\{l_{n,start}, l_{final,n}\},\ with\ l \in L, n \in N$$

(13) **Property 13. Mapping of instances of type storage:** An instance $i \in \mathcal{I}_{SD}$ of type $i \in controller$ cannot be mapped onto a memory block that describes a cache memory in a platform model. In DIPLODOCUS, the effects of caching are modeled by the *cache-miss ratio* parameter of a generic CPU block [Apvrille et al. 2006; Apvrille 2008].

4.2. Modeling a DMA data transfer with UML/SysML Communication Patterns

In this subsection we present the model of a generic DMA transfer with CPs. The main Activity Diagram of the Communication Pattern is illustrated in Fig. 4a. In this diagram we decomposed the communication protocol in three Sequence Diagrams: first the data transfer is configured (ConfigureTransfer in Fig. 4a), then data are transferred (TransferCycle in Fig. 4a) and the data transfer is terminated (TerminateTransfer in Fig. 4a). Data are transferred iteratively, as expressed by the for-loop operator, based on the value assigned to the control variable `counter` in diagram ConfigureTransfer.

The Sequence Diagram ConfigureTransfer is depicted in Fig. 4b. Here, we model how a generic CPU unit configures the DMA controller unit. These two units are represented as two instances of type controller, interconnected by an instance of type transfer. The CPU instance sends the source and destination addresses as well as the amount of data to transfer as parameters of the message `TransferRequest()` to the DMA controller, via the transfer instance. The DMA controller, upon reception of the message, assigns variable `dataToTransfer` to `counter`. The value of these variables is not known at modeling phase as CPs are independent of the data dependencies in the application model. A value will be assigned at mapping phase, step L1 in Fig. 3.

In the Sequence Diagram of Fig. 5, TransferCycle, we model one DMA transfer cycle. For this purpose we instantiate the DMA controller of Fig. 4b, a source and destination

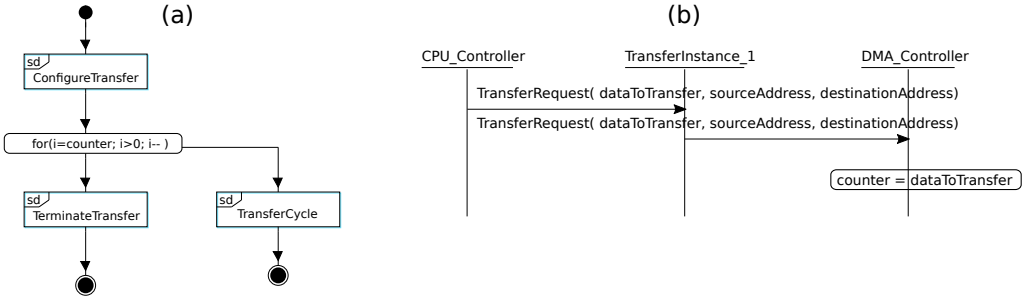

Fig. 4. The main Activity Diagram of a DMA data transfer (a), Sequence Diagram ConfigureTransfer (b).

storage instances interconnected by two transfer instances. In this diagram, the DMA controller reads samples out of the source storage instance via a parameterized `Read()` message. Subsequently, it writes data to the destination storage instance via a `Write()` message. As the values of parameters `size`, `sourceAddress` and `destinationAddress` depend on the architecture units, they will be assigned a value when mapping the instances onto DMA and memory units as described in subsection 2.1. Parameter `size` defines the amount of data that the DMA channel transfers each transfer cycle.

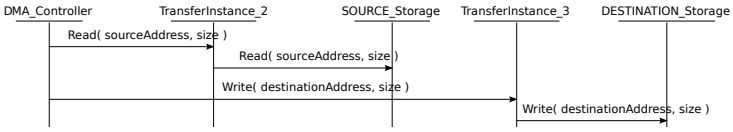

Fig. 5. The Sequence Diagram TransferCycle of Fig. 4a.

In the Sequence Diagram TerminateTransfer of Fig. 4a, the DMA controller informs the CPU instance that the transfer is terminated via an acknowledgment message.

5. THE TRANSFORMATION OF COMMUNICATION PATTERNS IN TTOOL/DIPLODOCUS

The introduction in the design flow of dedicated communication models adds modularity and allows to capture more complex transfers than those occurring on simple paths such as processor-bus-memory. It allows to evaluate performance in a single design step, thus adding better chances to avoid local optima. However, these additional models also increase the complexity of the design space, as the impact of both computations and communications must be evaluated within the same design phase.
In this section, we discuss the model transformations that we implemented to simplify the design space and to allow for a one-step evaluation of the impact of both communications and computations. Subsection 5.2 describes the transformation of Communication Patterns for the simulation and formal verification engine of TTool/DIPLODOCUS [Knorreck 2011]. Subsection 5.3 describes the transformation of a mapping model into executable control code. By executable control code, we mean the code that configures and triggers the data-processing units (e.g., CPU, DSP) onto which the computations of the application model have been mapped. This code also configures and triggers the data-transfer operations described by the CPs.

5.1. The software architecture of TTool/DIPLODOCUS

Fig. 6 illustrates the software architecture of TTool/DIPLODOCUS that is relevant to the model-to-code transformations presented in this section. At its topmost level, the Diagram Editor is an in-house Java Graphical User Interface that permits designers

to draw UML/SysML diagrams. Graphical models are first converted into an Intermediate Format (IF) Java data structure (Models-to-data-structure Transformation, Intermediate Format in Fig. 6). This software data structure constitutes a common layer from which models are transformed for DSE via formal verification, simulation and for implementation (prototyping) via the synthesis of the executable control code. TTool/DIPLODOCUS is also capable to produce the source code for Design Space Exploration from a textual representation of a design, specified in a dedicated language called Task Modeling Language (TML) [Waseem et al. 2006]. This TML representation can be automatically generated by the tool from the IF data structure, as depicted in Fig. 6, or manually input by the user, as an alternative to UML/SysML diagrams.

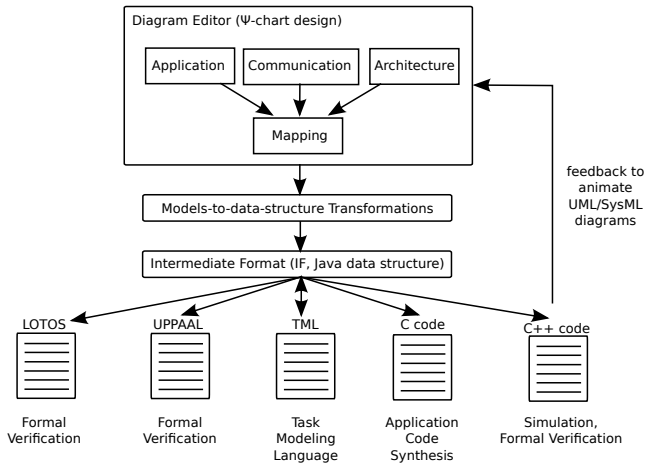

Fig. 6. The software architecture of TTool for the UML/SysML profile DIPLODOCUS

5.2. From UML/SysML models to the simulation and formal verification source code

The challenge of translating Communication Patterns for simulation and formal verification lays in providing transformation rules and a transformation algorithm that integrate the existing transformation process [Knorreck 2011] without modifying the internal functioning of the simulation and formal verification engine. This engine evaluates the functionality of a system described as a network of tasks interconnected by data and control dependencies. The internal behavior of each task is specified as a state machine that references operators for reading/writing to data channels, sending/receiving control information and governing the execution of activities (e.g., for-loop, choice, random sequence). These tasks and operators are proper to the above mentioned Task Modeling Language (TML). Therefore, in the rest of this subsection, we describe the transformation of CPs in terms of TML concepts rather than in terms of the elements that constitute the IF data structure.

5.2.1. The transformation process. The overall transformation process is depicted in Fig. 7. Our contribution is the translation of a mapped Communication Pattern into a set of equivalent TML tasks, CPs-to-TML transformation in Fig. 7. These TML tasks represent the functional behavior of a CP's controller instances (the only active instances of a CP, as defined in Section 4). Additionally, we integrated the CPs-to-TML transformation with the existing TML representation of an application model, Merge TML_{CP} and TML_{APP} in Fig. 7.

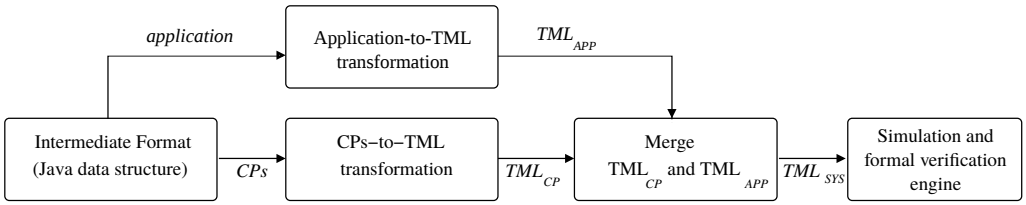

Fig. 7. An overview of the code generation process for simulation and formal verification

For each instance of type controller $i^{controller} \in \mathcal{I}_{SD}$ in a Communication Pattern, a TML task $TML^{task}_{i^{controller}}$ is produced. The internal behavior of $TML^{task}_{i^{controller}}$ is defined by the events $e_{i^{controller}} \in E_{i^{controller}}$. Each $e_{i^{controller}}$ is translated into its equivalent TML instruction, e.g., messages are translated into the TML instructions to read/write data from/to channels and receive/dispatch control information. The resulting set of equivalent TML tasks, TML_{CP} in Fig. 7, is then interconnected to the TML tasks of the data-processing operations from the application model, TML_{APP} in Fig. 7. These interconnections are created according to the mapping information that associates a Communication Pattern to a data-dependency between a computation producing data and a computation consuming these data (step L1 in Fig. 3). This results into a holistic representation of the system functionality, TML_{SYS}, that constitutes the input to the the existing simulation and formal verification environment [Knorreck 2011].

5.3. From UML/SysML models to the executable system control code

Code generation from system-level models is challenging as target platforms are composed of a set of heterogeneous units (e.g., DSPs, CPUs, DMAs, Hardware Accelerators) with different characteristics such as Instruction Set Architecture, Application Programming Interface and memory organization. For a given functionality, it may be desirable to generate executable code for different target platforms. In this context, the key issue is how to efficiently add implementation details that are platform-specific, to system-level models that make use of high level abstractions in order to be platform-independent.

To address this issue, the approach that we propose, Fig. 8, is based on two separate compilation steps. First, an input mapping model is translated by a model-to-executable-code compiler into C code that is compliant with a target platform's Application Programming Interface (API) and data structures. In this first compilation step, implementation details (e.g., data structures, register files) are added by a library of platform-specific entities called Model Extension Constructs (MECs). By linking to the compiler a library of MECs for a different platform (or for a different platform configuration), the code generation process achieves the desired cross-platform portability. Secondly, this C code is given as input to a commercially available compiler (e.g., gcc, Turbo C) to produce an executable file.

In the rest of this sub-section we detail our implementation of the code generation process in Fig. 8 for TTool/DIPLODOCUS. In this context, the model-to-executable-code compiler, Fig. 8, has been developed in Java in order to be easily plugged to the existing software architecture (Fig. 6). As this work is a first contribution that lays the ground for future developments, we specify here that our implementation is focused on signal-processing platforms. We also precise that, specifically to our implementation, the executable output file is a monolithic *application* that runs as a single process on top of the software stack (e.g., Board Support Package, Operating System) of a control processor in the target platform.

5.3.1. The compilation process. The compilation process of Fig. 8 is an extension of the the code-generation engine first proposed in [Gonzalez Pina 2013]. Compilation step I in Fig. 8 takes as input the equivalent representation of a mapping model from the Intermediate Format Java data structure of TTool/DIPLODOCUS (Fig. 6). It outputs a set of C files and a Makefile to automate the second compilation step. In the output C files, processing and communication operations from the initial mapping model are transformed into three routines that contain initialization, execution and clean-up code. Additionally, a fourth routine, called *fire-rule*, is assigned to an operation to specify the logical dependencies that must be satisfied for its execution.
The front-end of our model compiler in Fig. 8 is a parser that checks the correctness and coherency of a mapping (e.g., the mapping of instances of a Communication Pattern must respect the topology specified in the platform model) and converts the IF Java data structure into a directed graph representation, $G = (O, E)$. In this graph, processing and communication operations constitute the vertexes $o \in O$. The edges $e \in E$ in G represent dependencies between operations that are created based on the information entered by a user when mapping the models in the Ψ-chart.
Subsequently, the compiler's middle-end takes as input the operation graph G, analyzes its schedulability and produces an annotated version G', where edges and nodes are enriched with scheduling information. G' is then processed in order to allocate memory regions for input/output data of each processing and communication operation. This produces a second annotated graph, G'', that is transformed in C code by the compiler's back-end. The latter is a C code generator that also takes as input a library of data structures and code snippets that are compliant to the target platform's API. To cope with the heterogeneity of units in a target platform, the back-end relies on dedicated Model Extension Constructs (MECs). A MEC is associated to each annotated operation $o \in O''$, where $O'' \in G''$. It maps o to the code snippets and the data structures offered by the platform unit to which o had been bound in the initial mapping model.

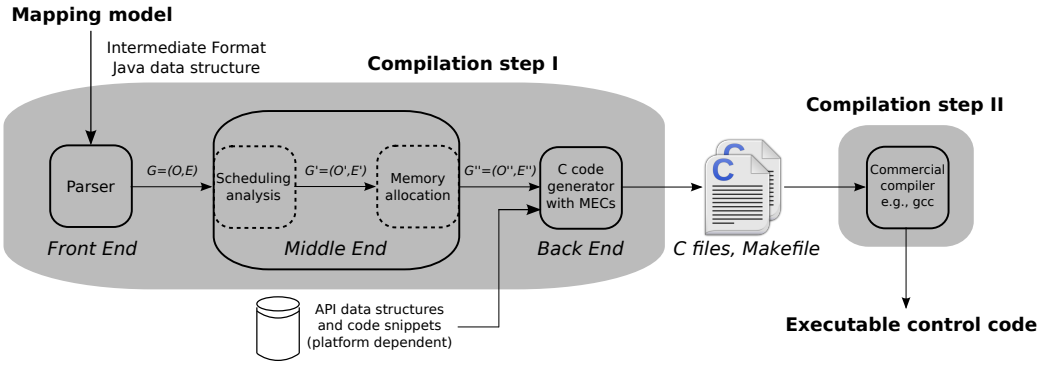

Fig. 8. An overview of the two-step compilation process to generate the executable system control code

5.3.2. Scheduling of operations. Scheduling information is annotated in $G' = (O', E')$ based on the *events* generated by the availability of data produced/consumed by operations $o \in O,\ O \in G$, according to the Synchronous Data Flow Model of Computation. This MoC has been selected as our code generation engine currently targets radio signal-processing applications. As part of our future work, we will extend the scheduling of G to other MoCs. Our implementation of the scheduling analyzer favors

an event-driven programming model rather than threads, as using threads and synchronization mechanisms would lead to rigid descriptions that are difficult to be scaled according to the different scenarios that can occur in data-dominated systems [Ousterhout 1996; Dabek et al. 2002]. For instance, in a signal-processing system composed of multiple applications, in case one or more of these applications stops execution, it would be more difficult to re-synchronize its execution using threads [Lee 2006].

5.3.3. Memory allocation. The compiler's middle-end in Fig. 8 allocates memory regions for operations according to the mapping information introduced by a user at step L1 (mapping of storage resources) in Fig. 3). This results into a static allocation policy that we propose to extend to a more dynamic solution (i.e., the memory regions are selected by a memory manager at run-time), as part of our future work, Section 8.

5.3.4. Portability of the code-generation approach. Our framework addresses platforms where the scheduling of operations is centralized by a general-purpose control processor. The latter configures and dispatches the execution of operations to a set of physically distributed units (e.g., DSPs, DMAs), according to the events generated by the consumption/production of data. For a design project that includes multiple platforms with a centralized controller and distributed execution units, a library of MECs must be provided to compile mapping models to sets of code snippets and data structures that are compliant to different APIs. For each platform, dedicated MECs must be provided by re-using those from other projects as templates. To target architectures where both control and execution of operations are physically distributed onto different units, the C code generator that must be adapted to produce multiple *applications* that will each be executed by a different control processor. Therefore, synchronization primitives must also be added to coordinate the parallel execution of these applications.

6. CASE STUDY

In this section, we deploy our implementation of the Ψ-chart in TTool/DIPLODOCUS to design the physical (PHY) layer of a ZigBee (IEEE 802.15.4 standard) [IEEE 802.15.4 2003] transmitter. Subsequently, we demonstrate, for a set of different mapping alternatives, how the Ψ-chart approach improves the portability of a design and reduces the number of design iterations, with respect to the Y-chart.

The IEEE 802.15.4 standard specifies both the MAC and the PHY layers of the IEEE 802.15.4 protocol. It is a standard for low-rate Wireless Personal Area Networks (WPANs) [Cooklev 2004], which are used to convey information over relatively short distances. ZigBee has been deployed for several applications including Wireless Sensor Networks (WSN) for building automation, remote control, health care, smart energy, telecommunication services. Among the different schemes that can be derived from the IEEE 802.15.4 standard for a ZigBee transmitter, we selected the one proposed by [Koteng 2006], shown in Fig. 9, because of its simplicity in terms of implementation.

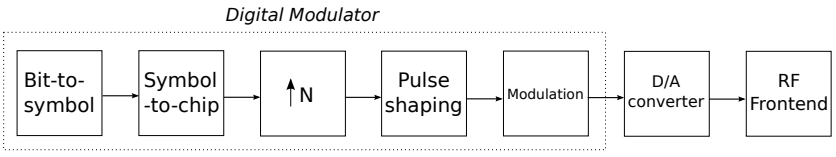

Fig. 9. The functional block diagram of the ZigBee transmitter as proposed by [Koteng 2006].

6.1. The platform

The target hardware architecture for our case study is Embb [Muhammad et al. 2008], a generic baseband architecture dedicated to signal processing applications.
Fig. 10a shows the UML Deployment Diagram of the Embb architecture, as modeled in TTool/DIPLODOCUS. Embb is composed of a Digital Signal Processing part (DSP part) and a general purpose control processor (the main CPU). In the DSP part, left-hand side of Fig. 10a, samples coming from the air are processed in parallel by a distributed set of Digital Signal Processing Units (DSPU1 through DSPUn) interconnected by a crossbar (Crossbar). Fig. 10b illustrates the internal architecture of a DSPU: each unit is equipped with a local micro-controller (μC) that allows to reduce the intervention of the main CPU, a Processing Sub-System (PSS), a computational unit, and a Direct Memory Access controller (DMA) to transfer data in and out of the DSPU's local memory (the Memory Sub-System, MSS). The latter is mapped on the global address map of the main CPU and is accessible by the DMAs, the μCs and the system interconnect. The system interconnect permits exchanges of control and data items: it is composed of a crossbar (Crossbar), a bridge (Main Bridge) and a main bus (Main Bus). The system interconnect is shared between the DSP part and the main CPU, where the control operations of an application are executed. The main CPU is in charge of configuring and controlling the processing operations performed by the DSPUs and the data transfers. The main CPU has direct access to a memory unit (MAINmemory) and a bus interconnect (MAINbus) that communicates with the DSP part via the Main Bridge.
According to our design experience with the Embb platform, the best configuration for

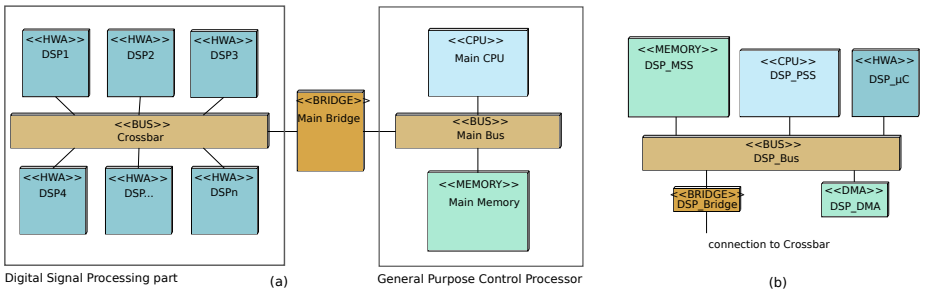

Fig. 10. The UML Deployment Diagrams of an instance of Embb, part (a), with its Digital Signal Processing part (left side) and main CPU (right side). Part (b) shows the internal architecture of each DSP unit.

signal-processing applications, in terms of performance is the one with the following four DSP units:

— Front End Processor (FEP): it implements Discrete Fourier Transform and vector processing operations.
— Interleaver (INTL): it implements permutations (i.e., interleaving and de-interleaving) of sequences of data samples.
— Mapper (MAPPER): it transforms a frame of input symbols into a frame of complex numbers representing the points of a 2D constellation diagram, via Look-Up-Tables.
— Analog to Digital-Digital to Analog Interface (ADAIF): a dispatcher that is capable of receiving up to 4 input streams from 4 A/D converters and of transmitting up to 4 output streams to 4 D/A converters.

As Embb is a reconfigurable platform where each DSP unit is programmable via a unique set of registers with specific memory access policies, we linked the library of

Model Extension Constructs corresponding to the above configuration to the model-to-executable-code compiler of Fig. 8.

6.2. The application

Fig. 11 shows the TTool/DIPLODOCUS diagram corresponding to data-flow model of Fig. 9 implemented for Embb. Here, the block labeled Source produces the data to be transmitted in the form of a flow of bits. These data are then converted to symbols by the Symbol2ChipSeq block. In this block, we model the mapping of each incoming 4-bits symbol to one of the 16 sequences of 32 chips as defined by the IEEE standard 802.15.4. The Chip_to_Octet block, then transforms each incoming chip (bit) of a chip sequence into an unsigned 8-bits integer as expressed in equation 1:

$$\{0; 1\} \rightarrow \{0x00; 0x01\} \tag{1}$$

Chip_to_Octet also separates the even-indexed chips that are used to modulate the in-phase (I branch) carrier component from the odd-indexed chips that are used to modulate the quadrature (Q branch) carrier component. The output is then transformed by means of a Component Wise Lookup (CWL block) that maps unsigned 8-bits integers to signed 16 bits integers as expressed by equation 2:

$$\{0x00; 0x01\} \rightarrow \{0xffff; 0x0001\} \tag{2}$$

At this point, given the separation of the I and Q branches, their pulse shaping can be executed independently. The application graph exposes this parallelism by forking the output data of block CWL to two distinct Component Wise Product (CWP) blocks, CWP_I for the I branch and CWP_Q for the Q branch. These blocks multiply the input samples with a half-sine wave to realize the O-QPSK modulation. The quadrature shift between the I and Q branches is implemented by means of an offset between the memory addresses of the output samples. This results into a frame of complex samples (16 bits for the real part and 16 bits for the imaginary part) that is then collected by block Sink and transmitted over the air.
Each block of the model in Fig. 11 is composed of two tasks: one modeling the data-processing and one modeling the related control operations. By convention we name the data-processing tasks with a heading X that stands for eXecution and the control tasks with a heading F that stands for Firing.

6.3. Communications

6.3.1. The communication mismatch. In TTool/DIPLODOCUS, communications are described as in the application model of Fig. 11 as point-to-point data channels between tasks. In the platform model of Fig. 10, communications are represented as read/write operations performed by CPU and DSP units to/from memory units. Therefore, communication mismatches arise when data are transferred via paths, in the platform model, that encompass a sequence of more than one pair bus-bridge between a source CPU/DSP and its destination memory. For instance, when data are transferred (i) from `MainMemory` to any of the DSP local memories and vice-versa, and (ii) from a DSP local memory to any other DSP local memory. The only case in which there is a match between the application and the architecture MoC is the path `MainCPU-MainBus-MainMemory` or by the path that links a DSP PSS to its local memory.

6.3.2. The platform's communication protocols and mechanisms. Data in Embb are transferred in one of the two following ways: (i) via a DMA transfer (to upload data to process in MSS and to download processing results) and (ii) via load/store instructions issued by the main CPU (i.e., General Purpose Control Processor) to read/write data from/to the main memory.

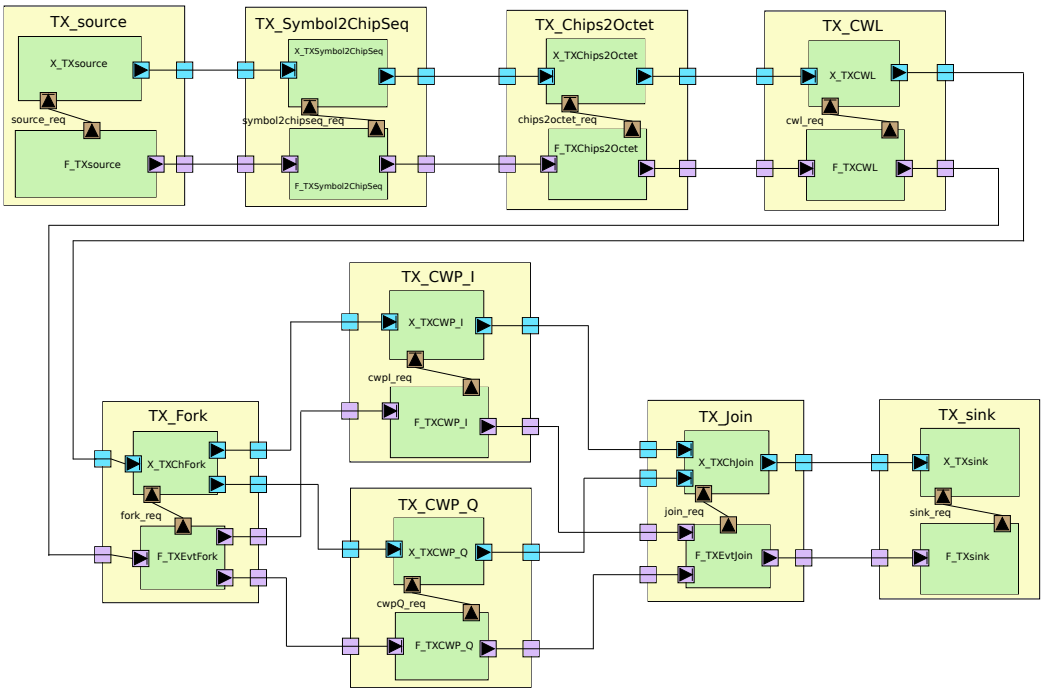

Fig. 11. The TTool/DIPLODOCUS model of the ZigBee transmitter

6.3.3. Communication Patterns. As described above, the communication protocols and patterns that we need to model are DMA transfers with interrupt mechanisms. The CP for a DMA transfer has already been illustrated in Section 2 and will not be repeated. In this Communication Pattern, the interrupt signal that is sent by a DMA controller to a CPU controller is represented by the message `TranferTerminated()`, Fig. 4a.
A novel Communication Pattern that we need in this case study is the one shown in Fig. 12a. Here, the main Activity Diagram captures a memory copy transfer that is used in Embb to move data from the `MainMemory` to the local memory of any DSP unit, via a store operation issued by the `MainCPU`. The Sequence Diagram `TransferCycle` in Fig. 12a, models the message exchanges in the same way as the diagram in Fig. 5, except for the decrement of attribute `counter`.
Additionally, we also need the CP illustrated in Fig. 12b. This model captures a pair of sequential DMA transfers and can be used to describe a copy operation from one source storage to two different destination storages. The main Activity Diagram of this CP is composed of two references to Activity Diagrams, that each describe a DMA transfer as the one illustrated in Fig. 12c for DMATransfer1.

6.4. The mapping

According to the mapping methodology of Section 2.1, we first map the computations of the application model. Such a mapping results in each control task (e.g., F_Symbol2ChipSeq) being executed by the Main CPU unit and the data-processing tasks (e.g., X_Symbol2ChipSeq) being executed by the DSPUs PSS. Secondly, the memories where to store input/output data are chosen. This results into a mapping where the local memory of each DSPU is used to store the input/output data for the computations that have been mapped onto the DSP's Processing SubSystem, e.g., task

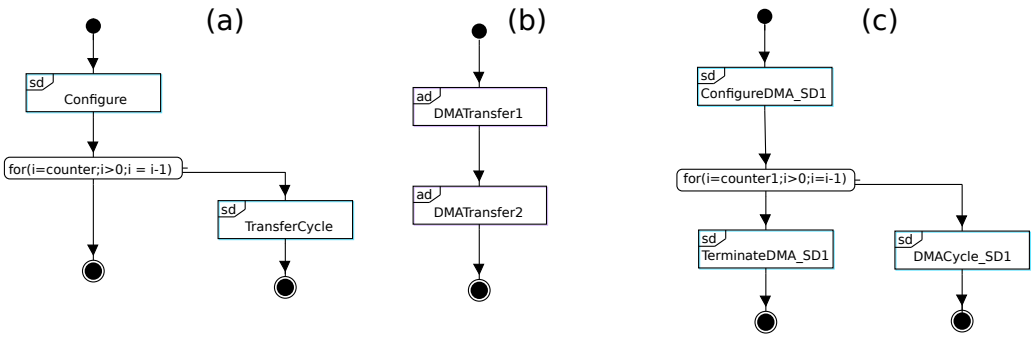

Fig. 12. The main AD for a CP modeling a CPU memory copy (a). The main AD for a CP modeling the sequence of two DMA transfers (b). Part (c) shows the AD referenced by DMATransfer1 in (b).

X_Symbol2ChipSeq is mapped to the Mapper PSS, the input/output data are mapped to the Mapper local Memory SubSystem (MSS).
From our library we instantiate and map 4 Communication Patterns:

— *CP01*: a memory copy CP that transfers the output data of task X_Source. It is composed of: 1 controller instance (CPU_Controller), 2 storage instances (Src_Storage, Dst_Storage) and 2 transfer instances.
— *CP02*: a DMA CP that transfers the output data of X_Symbol2ChipSeq. It is composed of: 2 controller instances (CPU_Controller, DMA_Controller), 2 storage instances (Src_Storage, Dst_Storage) and 4 transfer instances.
— *CP03*: a DMA CP that transfers the output data of X_Chip2Octet. It is composed of: 2 controller instances (CPU_Controller, DMA_Controller), 2 storage instances (Src_Storage, Dst_Storage) and 4 transfer instances.
— *CP04*: the sequence of two DMA CPs that transfer the output data of X_CWP_I and X_CWP_Q. It is composed of: 4 controller instances (2 CPU_Controllers, 2 DMA_Controllers), 4 storage instances (2 Src_Storage, 2 Dst_Storage) and 8 transfer instances.

The above CPs are mapped onto the units listed in Table I. Due to lack of space, we do not show the mapping at routing level (L3 in Fig. 3) of the transfer instances.

Table I. The mapping of the CPs' controller and storage instances

Identifier	Instance	Architecture unit
CP01	Src_Storage, Dst_Storage	Main Memory $\text{MAPPER}_{\text{MSS}}$
CP02	DMA_Controller, Src_Storage, Dst_Storage	$\text{MAPPER}_{\text{DMA}}$ $\text{MAPPER}_{\text{MSS}}$ INTL_{MSS}
CP03	DMA_Controller, Src_Storage, Dst_Storage	INTL_{DMA} INTL_{MSS} FEP_{MSS}
CP04	DMA_Controllers, Src_Storages, Dst_Storages	FEP_{DMA} FEP_{MSS} $\text{ADAIF}_{\text{MSS}}$

Table II shows a set of mapping alternatives for the DMA controllers of Communication Patterns CP01-CP04. In subsections 6.5 and 6.6, we discuss the Model Improvements phase (step 4 in Fig.1 and Fig.2) of these alternatives and present the related

costs in the frame of the Ψ-chart and Y-chart versions of TTool/DIPLODOCUS, respectively. We then compare the costs for the two approaches in subsection 6.7.

Table II. A summary of the mapping alternatives for the DMA controllers of the CPs

Mapping configuration	CP01	CP02	CP03	CP04
Configuration I	CPU load-store	MAPPER_DMA	INTL_DMA	FEP_DMA
Configuration II	CPU load-store	ADAIF_DMA	INTL_DMA	FEP_DMA
Configuration III	MAPPER_DMA	MAPPER_DMA	INTL_DMA	FEP_DMA
Configuration IV	ADAIF_DMA, ADAIF_DMA	MAPPER_DMA	INTL_DMA	FEP_DMA

6.5. Model Improvements in the frame of the Ψ-chart

Exploring the alternatives in Table II means to (re-)bind CPs to different platform units. As CPs are part of a pre-mapping library of models, there are no costs associated to their instantiation. When mapping alternative I, the 4 CPs must be bound to the platform model from scratch, while alternatives II-IV only require to change the binding of CP02 and CP01. Generally speaking, in case a novel CP must be created from the composition of "primitive" communication protocols from the library, additional costs to improve models only involve the creation of a novel main Activity Diagram.

6.6. Model Improvements in the frame of the Y-chart approach

Because of communication mismatches, the mapping phase in the Y-chart of Fig. 1 is altered as shown in Fig. 13. As a consequence, the design process involves more iterations, i.e., re-modeling and re-mapping steps (4). In Fig. 13, steps (1.1), (1.2) and (3) are equivalent to those in Fig. 1, while steps (2.1)-(2.3) are described as follows:

2.1 Partial mapping model: it captures the mapping of computation operations only.
2.2 Hybrid application model: it is an instance of the Pure application model, where additional tasks are added between processing operations to capture data-transfers due to one or more communication mismatches that prevent the creation of a single mapping model.
2.3 Complete mapping model: it is an instance of the Partial mapping model that includes the mapping of communications described in the Hybrid application model.

By way of example, Fig. 14 shows the Hybrid application model for mapping configuration I in Table II.

6.7. A comparison between the Model Improvements phase in the Y-chart and in the Ψ-chart

To quantitatively evaluate the benefits of the Ψ-chart over the Y-chart when improving a design in terms of communications, we measured the time taken by a user to bind a CP (Ψ-chart) versus the time required to model and map the tasks that capture communication protocols in the Hybrid application model (Y-chart in Fig.13). In the Y-chart version of TTool/DIPLODOCUS, for the DMA transfer and the CPU memory-copy operations, the average time required for modeling is 30 seconds. The average time required to map a task or a channel is 5 seconds. In the Ψ-char version of TTool/DIPLODOCUS, the average time required to map a CP (regardless of the specific protocol being captured) is 60 seconds.
Table IV summarizes the costs associated to the mapping configurations of Table II. In configuration I, the costs associated to the Y-chart design are given by the creation of 6 tasks and 12 channels (blocks `DMAmapper`, `DMA_INTL`, `DMA_FEP` in Fig.14) that amounts to 180 s as well as by their mapping that amounts to 90 s. Instead the costs associated to the Ψ-chart-based design amount to 240 s, due to the mapping of the 4 CPs. In

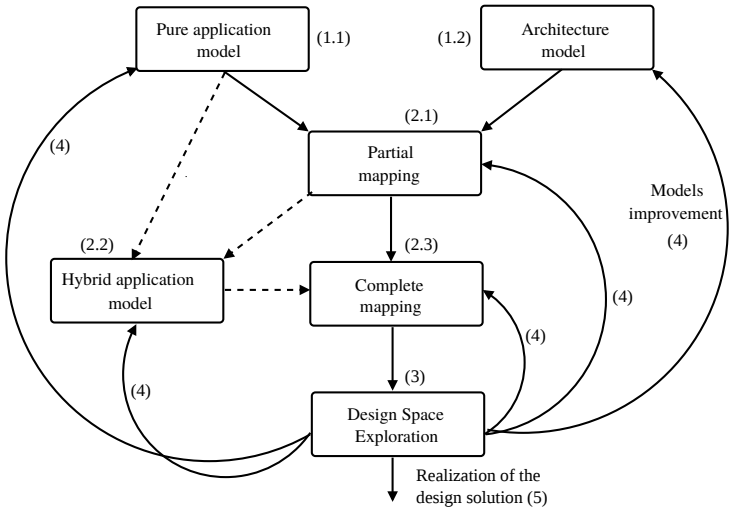

Fig. 13. The Y-chart in TTool/DIPLODOCUS as it results from being altered by communication mismatches.

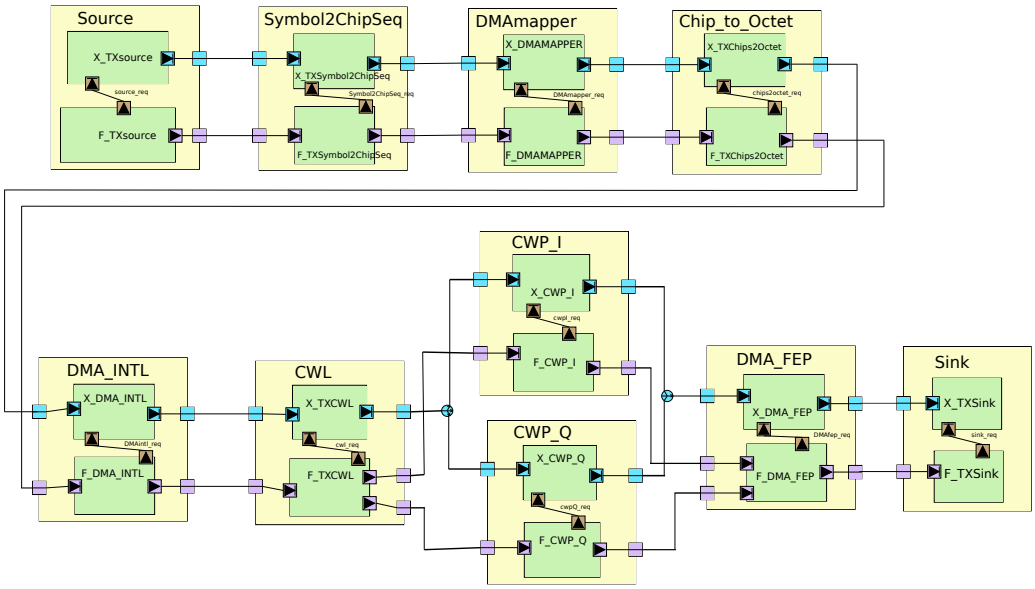

Fig. 14. The TTool/DIPLODOCUS Hybrid application model for mapping configuration I in Table II.

configuration II, the transfer of data from task `TX_Symbol2ChipSeq` to `TX_Chips2Octet` in Fig.11 is assigned to a different DMA engine. In the Ψ-chart-based design, this requires to change the binding of CP02 and costs 60 s. In the Y-chart-based design, this requires to change the mapping of block `DMAmapper` in Fig.14 and costs 30 s (remapping of 2 tasks and 4 channels). In configuration III, a DMA transfer is used instead of the CPU memory-copy operation to move data from the source to `TX_Symbol2ChipSeq`. In the Ψ-chart-based design, this requires to instantiate a CP for a DMA from the library and to bind it to the platform model, thus it requires 60 s. In the Y-chart-based design, this requires the modeling and mapping of 2 tasks and 4 channels that results in 90

s. In configuration IV, a double DMA transfer is used instead to move data from the source to `TX_Symbol2ChipSeq`. In the Ψ-chart-based design, this requires to instantiate a CP for a double DMA from the library and to bind it to the platform model, thus it requires 60 s. In the Y-chart-based design, this requires the modeling and mapping of 4 tasks and 8 channels that costs 180 s.

Table III. A summary of the costs required to improve model alternatives in the Y and in the Ψ charts

Mapping configuration	Model improvement costs Y-chart	Model improvement costs Ψ-chart
Configuration I	modeling: 180 s, mapping: 90 s	mapping: 240 s
Configuration II	modeling: 0 s, mapping: 30 s	mapping: 60 s
Configuration III	modeling: 60 s, mapping: 30 s	mapping: 60 s
Configuration IV	modeling: 120 s, mapping: 60 s	mapping: 60 s

We specify, that these costs are only related to the improvements of the models for communication protocols and patterns. They are not representative of the total design costs as they do not include the time taken to create the application and platform models, nor the time required to explore the design space of each configuration, etc.
From the above discussion, the total costs for exploring the 4 mapping alternatives amounts to 570 s for the Y-chart and to 420 s for the Ψ-chart. Thus, the latter reduces the design costs associated to the model improvements phase by a factor of 26%.

7. RELATED WORK

Many design approaches at Electronic System-Level (ESL) have been proposed since [Balarin et al. 1997] and [Kienhuis et al. 1997] first discussed the importance to separate application and platform design and its methodological consequences. Existing tools and environments are the implementation of two strategies: (i) the *platform-based* approach and (ii) the *layered* approach, [Lukasiewycz et al. 2009].
In the *platform-based* approach, resources are allocated from a platform template that is instantiated from a library of generic components. Subsequently, an application model is mapped according to a set of mapping constraints. These approaches follow the Y-chart paradigm. We discuss here a few relevant implementations.
The work in [Lukasiewycz et al. 2009] introduces a DSE approach to enable concurrent process binding and communication routing. Design in [Lukasiewycz et al. 2009] is based on the Y-chart. This work differentiates from other implementations of the Y-chart, as the allocation and binding of communications occurs in a single mapping step, together with the allocation and binding of computations. Similarly to our contributions, this avoids expensive design iterations (i.e., re-modeling and re-mapping of communications). To cope with the increase in complexity of the design space, the authors in [Lukasiewycz et al. 2009] propose the transformation of the initial specifications into graph-based models and symbolic representations.
The authors in [Pimentel and Erbas 2003] propose a transformation method for communication refinement. In this work, communication events (e.g., check data, load data, store data) are refined from the abstract read and write operations generated from the simulation of an application modeled as a Kahn Process Network (KPN). The refinement of communication events aims to provide a match between the abstract semantics of the communications in the application model and those in the architecture units. Nevertheless, this design flow does not provide a solution to explicitly describe the behavior of the platform units in terms of communication protocols and services (e.g., network behavior). While this refinement methodology lays the ground for the application scheduling analysis, only read and write accesses from a processor to a local or shared memory can be described. In terms of model improvements, the whole

refinement process must be re-done from scratch if a given mapping results not to be compliant with the desired performance requirements. This is opposed to the Ψ-chart, where only the mapping of the communication models must be repeated.

Independently of our work, [van Stralen and Pimentel 2012; van Stralen 2014] extend the Y-chart approach with a third input to explicitly model fault-tolerance patterns. The authors remark that the resulting methodology appears to be a Ψ-chart but claim that there is no difference with respect to the principles of the Y-chart (i.e., separation between the design of the system's functionality and that of the resources). With respect to our contributions, the work in [van Stralen and Pimentel 2012; van Stralen 2014] has a common denominator: to include an important design aspect (i.e., fault-tolerant versus communication patterns) early, as a third input to the design flow. However, in [van Stralen and Pimentel 2012; van Stralen 2014] communication schemes and protocols may be accounted in a fault-tolerant pattern only if that specific pattern considers faults that affect the interconnect architecture. Additionally, the model for a fault-tolerant pattern is not independent of the application and platform models. More in detail, a hardware fault-tolerant pattern requires to modify the platform model (e.g., to consider faults in processors where computations are mapped). A software fault-tolerant pattern may require to also modify the application model (e.g., a fault tolerant technique called Double Modular Redundancy requires each computation to be performed twice and then to compare the outcome). Indeed, to represent these dependencies among models, the Ψ-chart of [van Stralen and Pimentel 2012; van Stralen 2014] graphically collocates the fault-tolerant patterns at a level of abstraction that is lower than the application and platform model.

On the other hand, in the *layered* approach the partitioning of computations is done before the design and exploration of communications. Typically, this approach does not follow the Y-chart: the design flow starts with a single specification of the system functionality. This is subsequently transformed into an implementation model through a set of refinement steps. These steps progressively disclose implementation details and account for different mapping constraints (e.g., computation, communication, scheduling). Relevant examples of this design paradigm are the System-on-Chip Environment (SCE) [Dömer et al. 2008], SystemCoDesigner [Keinert et al. 2009] and Koski [Kangas et al. 2006].

The System-on-Chip Environment (SCE) [Dömer et al. 2008] is a top-down methodology that takes as input a single specification of the whole system functionality. This model is refined down to an implementation through four hierarchical steps: architecture, scheduling, network and communication. At each refinement step, resources can be allocated and partitioned or a given configuration is explored and evaluated (functionality, performance). In terms of DSE, SCE favors a progressive pruning of the design space by first exploring computations and subsequently communications. Conversely, in the Ψ-chart, both computations and communications are explored in a single phase that privileges the study of their interactions. In terms of the expressive power of communication protocols and patterns, SCE is equivalent to the Ψ-chart as complex architectures (e.g. hierarchical bus-based, explicit DMA transfers) can be represented, explored and realized. The framework that supports SCE operates on descriptions specified in the standard SpecC language [Zhao et al. 2000]. The latter is a System-Level Design Language based on ANSI-C that provides additional constructs for modeling hardware (hierarchy, concurrency, synchronization, exception handling and timing). A system specified in SpecC can be fully realized in terms of both synthesizable hardware and compilable software. On the other hand, the higher level of abstraction of the DIPLODOCUS profile allows software realizations only, but faster verification thanks to the functional and data abstractions (Section 3) that reduce the combinatorial state-explosion problem typical of complex design spaces.

SystemCoDesigner [Keinert et al. 2009] is a SystemC-based tool for the Design Space Exploration and behavioral synthesis of streaming applications. The input of the design flow is a SystemC specification of the application behavior. In this model, communication between actors is based on FIFO semantics and the behavior of the communication mechanism is encoded as a Finite State Machine (FSM). DSE automatically searches for an optimal allocation of resources and an optimal binding of processes onto resources. For each solution, the performance is evaluated via simulation. Before DSE, a library provides generic communication resources that are mapped onto four different FIFO primitives for hardware/hardware, hardware/software, software/software and software/hardware communication.

Koski [Kangas et al. 2006] is a framework for the design of Multiprocessor Systems-on-Chip, where the design flow starts with a single UML 2.0 model that captures the system's requirements, application and architecture. Design Space Exploration in Koski is carried out in two phases: first, coarse-grain exploration is performed by statically analyzing the application model. Secondly, alternative architectures are iteratively explored via simulations and refinement of the system's models. The optimization objective of the DSE phase is to minimize the cost functions provided by a designer in the initial requirements. In terms of communications, the communication protocols are part of an architecture model and the deployment details are provided as part of the platform library. In the application model, UML Composite diagrams are used to described the connections between parts of the UML Class diagram. These parts communicate with each other via signals and ports. At mapping level, in order to perform DSE and later the system implementation, a design automation library provides interprocessor communication routines that are associated to the signals and ports of the UML Composite diagram for an application.

In the above works of [Dömer et al. 2008; Keinert et al. 2009; Kangas et al. 2006], layering the design and DSE in separate phases may lead to sub-optimal designs as opposed to the single-step evaluation of the Ψ-chart. At each refinement step (layer), locally optimal solutions, once explored, may be integrated into the overall design. This may result into a globally non-optimal solution.

8. CONCLUSION

Traditional design approaches for embedded systems (i.e., the Y-chart) start from specifications of the system's application and resources. In this paper, we presented a novel approach called the Ψ-chart that adds a third input to the design flow: dedicated models to capture communication protocols and patterns, independently of the system's functionality (application) and resources (platform), before mapping. We also presented the implementation of this approach in TTool/DIPLODOCUS, a UML/SysML framework for the early design and rapid prototyping of data-dominated embedded systems. Overall, our solution results in better design quality, portability and less design time, due to a reduction in the number of iterations that aim at improving models when DSE evaluates that a mapping configuration does not satisfy the desired requirements.

As part of our future work, we are improving the implementation of the Ψ-chart in TTool/DIPLODOCUS in terms of: (i) modeling (i.e., automate the mapping of CPs and capture communication protocols beyond the ISO/OSI datalink layer) and (ii) code generation (i.e., target platforms other than those dedicated to signal-processing).

REFERENCES

L. Apvrille. 2008. TTool for DIPLODOCUS: An Environment for Design Space Exploration. In *NOTERE*. 28:1–28:4.

L. Apvrille, W. Muhammad, R. Ameur-Boulifa, S. Coudert, and R. Pacalet. 2006. A UML-based Environment for System Design Space Exploration. In *Electronics, Circuits and Systems (ICECS)*. 1272–1275.

F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A.L. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. 1997. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers.

T. Cooklev. 2004. *Standards for Wireless Personal Area Networking*. Wiley-IEEE Standards Association.

F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. 2002. Event-driven Programming for Robust Software. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. 186–189.

Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel D. Gajski. 2008. System-on-chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design. *EURASIP Journal on Embedded Systems* 2008, 5 (2008), 1–13.

A. Enrici, L. Apvrille, and R. Pacalet. 2014. A UML Model-Driven Approach to Efficiently Allocate Complex Communication Schemes. In *Model Driven Engineering Languages and Systems (MODELS)*. 370–385.

J.M. Gonzalez Pina. 2013. *Application Modeling and Software Architectures for the Software Defined Radio*. Ph.D. Dissertation. Telecom ParisTech.

IEEE 802.15.4. 2003. IEEE 802.15 Wireless Personal Area Networks (WPAN) Task Group 4 (TG4). http://www.ieee802.org/15/pub/TG4.html. (2003).

C. Jaber. 2011. *High-Level SoC modeling and performance estimation applied to a multi-core implementation of a LTE enodeb physical layer*. Ph.D. Dissertation. Telecom ParisTech.

T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, J. Riihimäki, and K. Kusilinna. 2006. UML-Based Multiprocessor SoC Design Framework. *ACM Transactions on Embedded Computing Systems* 5, 2 (2006), 281–320.

J. Keinert, M. Streubühorbar, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith. 2009. SystemCoDesigner - an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM TODAES* 14, 1 (2009).

B. Kienhuis, E.F. Deprettere, P. van der Wolf, and K. Vissers. 2002. A Methodology to Design Programmable Embedded Systems - The Y-chart Approach. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS)*. 18–37.

B. Kienhuis, E.F. Deprettere, K. Vissers, and P. van der Wolf. 1997. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. In *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. 338–349.

D. Knorreck. 2011. *UML-Based Design Space Exploration, Fast Simulation and Static Analysis*. Ph.D. Dissertation. Telecom ParisTech.

R.M. Koteng. 2006. *Evaluation of SDR-implementation of IEEE 802.15.4 Physical Layer*. Master's thesis. Norwegian University of Science and Technology (NTNU).

E.A. Lee. 2006. *The Problem with Threads*. Technical Report. EECS Department, UC Berkeley.

M. Lukasiewycz, M. Streubuhr, M. Glass, C. Haubelt, and J. Teich. 2009. Combined system synthesis and communication architecture exploration for MPSoCs. In *DATE*. 472–477.

N.-ul.-I. Muhammad, R. Rasheed, R. Pacalet, R. Knopp, and K. Khalfallah. 2008. Flexible Baseband Architectures for Future Wireless Systems. In *EUROMICRO DSD*. 39–46.

J. Ousterhout. 1996. Why threads are a bad idea (for most purposes). https://web.stanford.edu/ ouster/cgi-bin/papers/threads.pdf. (1996).

A.D. Pimentel and C. Erbas. 2003. An IDF-based Trace Transformation Method for Communication Refinement. In *Design Automation Conference*. 402–407.

M.A. Reniers. 1999. *Message Sequence Chart: Syntax and Semantics*. Ph.D. Dissertation. Eindhoven University of Technology.

P. van Stralen. 2014. *Applications of scenarios in early embedded system design space exploration*. Ph.D. Dissertation. University of Amsterdam.

P. van Stralen and A.D. Pimentel. 2012. A SAFE Approach towards Early Design Design Space Exploration of Fault-tolerant Multimedia MPSoCs. In *CODES+ISSS*. 393–402.

M. Waseem, L. Apvrille, R. Ameur-Boulifa, S. Coudert, and R. Pacalet. 2006. Abstract Application Modeling for System Design Space Exploration. In *Euromicro DSD*. 331–337.

S. Zhao, A. Gerstlauer, J. Zhu, D.D. Gajski, and R. Domer. 2000. *SpecC: Specification Language and Methodology*. Springer.

H. Zimmermann. 1980. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications* 28, 4 (1980), 425–423.