Model-Based Programming for Multi-Processor Platforms with TTool/DIPLODOCUS and OMC

Andrea Enrici[1], Julien Lallet[1], Renaud Pacalet[2], Ludovic Apvrille[2], Karol Desnos[3], and Imran Latif[1]

[1] Nokia Bell Labs, Centre de Villarceaux, 91620 Nozay, France,
firstname.secondname@nokia-bell-labs.com,
[2] LTCI, Télécom ParisTech, Université Paris-Saclay, 75013 Paris, France
firstname.secondname@telecom-paristech.fr
[3] INSA Rennes, IETR, UBL, CNRS UMR 6164, 35708 Rennes, France,
firstname.secondname.@insa-rennes.fr

Abstract. The complexity of today's multi-processor architectures raises the need to increase the level of abstraction of software development paradigms above third-generation programming languages (e.g., C/C++). Code generation from model-based specifications is considered to be more efficient with respect to traditional paradigms where software is mainly developed from code. However, existing model-based approaches typically generate application software in SoC-programming languages (e.g., C/C++, OpenCL, Verilog/VHDL) without considering the optimization of non-functional properties (e.g., memory footprint, scheduling). This paper proposes a novel approach and tools where system-level models are compiled into standard C code while optimizing the system's memory footprint. We show the effectiveness of our approach with the model-based programming of UML/SysML diagrams for a 5G decoder. From the compiled C code, we generate both a software implementation for a Digital Signal Processor platform and a hardware-software implementation for a platform based on hardware Intellectual Property (IP) blocks. Overall, our optimizations achieve a memory footprint reduction of 80.07% in the first case and 88.93% in the second case.

Keywords: Model-Based Engineering, MPSoC Programming, UML/SysML

1 INTRODUCTION

In order to support the data need of Internet of things and cloud computing, 5G networks are expected to provide a 10x higher data-rate with regards to 4G networks. Future architectures supporting 5G networks will probably be based on both dedicated circuits in base stations and on more flexible computing solutions such as cloud systems equipped with both programmable and configurable components (CPUs, Digital Signal Processors - DSPs, Field Programmable Gate Arrays - FPGAs). Because of the high complexity and heterogeneity of these architectures, developing signal-processing application will become a nightmare for engineers. Thus, a new programming paradigm is needed in order to increase the quality (e.g., scheduling, memory footprint) and productivity

(e.g., correct-by-construction code generation) of application software development.
Model-Driven Engineering (MDE) [Schmidt, 2006] is widely accepted as a promising software development paradigm to answer the above-mentioned issues. MDE combines domain-specific modeling languages to abstract the structure, behavior and requirements of a system under design, with transformation engines and generators. The latter analyze models and produce artifacts such as source code, simulation, verification inputs or alternative model representations. In the context of MDE, model-based code generation is incorrectly seen as a replacement for programming. On the contrary, it actually *is* an alternative programming paradigm, where the languages aim at abstracting software systems rather than hardware computing mechanisms [**?**]. Similarly to programming languages, also modeling languages have features (e.g., UML relationships) that are well suited to described certain things (e.g., relations among functions or classes) and are not suited to describe some other things (e.g., access and indexing of arrays).
The process of creating models from existing software systems is well understood. However, the reverse process of compiling model-based specifications into executable implementations still remain an open issue. Here, multiple challenges arise from the desire to generate code for different architecture topologies (e.g., IP-based platforms, Multi-Processor Systems-on-Chip - MPSoC), implementations (i.e., software, hardware and mixed hardware-software) and execution units (e.g., DSPs, CPUs, Hardware Accelerators).
In this paper we propose tools and a methodology to enhance the application software development for MPSoC platforms from system-level models. Our approach for model-based programming is centered around a model development environment, TTool/DIPLODOCUS [TTool, 2017a, Apvrille et al., 2006] and the Optimizing Model Compiler [Enrici et al., 2018] (OMC). TTool/DIPLODOCUS allows the creation, editing and debugging of UML/SysML diagrams, while OMC is a model-to-code compiler that takes as input system-level models (i.e., application, communications and architecture). It converts them into Intermediate Representations (IRs) and attempts to optimize the system's memory footprint. OMC produces as output standard C code for the memory allocation and scheduling of signal-processing operations, regardless of the system's final realization technology (e.g., FPGA, Application Specific Integrated Circuit - ASIC).
As a practical case study we propose the model-based design of a 5G datalink-layer decoder. The C program compiled from the decoder's models is transformed into executable implementations for (i) a DSP-based platform (software executable file further compiled with GNU/gcc) and (ii) a hardware IP-based platform (FPGA bitstream) by a traditional software compiler and a SoC design tool (Xilinx SDx).
The rest of the paper is organized as follows. Section 2 positions our work with respect to related contributions. The overall methodology for model-based programming is presented in Section 3. This is followed by the structure of a generic model compiler in Section 4. The model development environment that we selected, TTool/DIPLODOCUS, is briefly described in Section 5. The Optimizing Model Compiler (OMC) is presented in Section 6. Section 7 shows how TTool/DIPLODOCUS and OMC are applied to program the 5G decoder. Section 8 concludes this paper.

2 RELATED WORK

In the context of UML-based MDE, code generation for SoCs is based on a direct *translation* of UML modeling assets into constructs of a target language (e.g., a UML block becomes a C function), according to precise translation rules [Vanderperren et al., 2012]. Many works propose one-to-one translation rules for SoC languages such as C [Nicolas et al., 2014], C++ [Ciccozzi et al., 2012], Verilog [Bazydlo et al., 2014], VHDL [Moreira et al., 2010] and SystemC [Mischkalla et al., 2010, Xi et al., 2005, Tan et al., 2004]. A representative work that uses one-to-many translation rules is Gaspard2 (Graphical Array Specification for Parallel and Distributed Computing) [Gamatie et al., 2008, DaRTteam, 2009], a MDE SoC co-design framework based on MARTE [OMG, 2017]. Thanks to the notion of *Deployment*, in Gaspard2, an Elementary Component (a resource or a functionality in a MARTE model) is related to implementation code that specifies low-level behavioral or structural details in a usual programming language (e.g., C/C++) for formal verification, simulation, software execution and hardware synthesis.

Executable UML (xUML) or *executable and translatable UML* (xtUML) [Mellor and Balcer, 2003, Mellor and Balcer, 2002] defines both a software development methodology and a highly abstract software language that combines a subset of UML's graphical notation with executable semantics and timing rules. When programming in xUML, a system's application is captured in the metamodel. The model compiler comprises some library code and a set of rules that are interpreted against the metamodel to produce text for a target SoC (e.g., C++ classes, C structs; VHDL specifications for hardware registers). However, the overall architecture of the generated SoC is defined by the model compiler itself (i.e., its translation rules). As opposed to our approach, xUML considers a platform-*independent* model as the only input. To the best of our knowledge, no work exists that attempts to optimize the performance of code generated from the xUML subset.

The Foundational Subset for Executable UML Models (fUML) [fUML, 2016] and the Action Language for fUML (Alf) [Alf, 2017] standard were created to make xUML models detailed enough and well specified for detailed programming and machine execution. The goal of fUML is to go beyond xUML in specifying a reasonable subset of UML with a precise semantics, in order not to be specific to any executable modeling methodology. The syntax of Alf is borrowed from Java, C, C++ and C# to specify the behavior and computation (concurrent data-flow semantics) of graphical fUML models. xUML, fUML and Alf are essentially focused on specifying a semantics suitable to generate executable code from UML graphical models. With respect to this, our work goes one step further. Our model compiler demonstrates that non-functional properties of a system denoted with UML/SysML diagrams can be improved before code generation, with a significant impact on the performance of the final executable (e.g., memory footprint reduction).

In the 2011 edition of MODELS, the work in [Floch et al., 2011] illustrated how MDE techniques (e.g., meta-metamodels, meta-tools, Domain Specific Languages) can be applied to help in solving or simplifying issues such as code maintainability and sustainability, interfacing with external tools, semantics preserving of the Intermediate Representation transformations and code generation. While [Floch et al., 2011] tries to

bridge the gap between model-based optimizations and abstract representations of pure software systems, our work transforms system-level models that also include hardware components (e.g., on-chip RAM memories).
The landscape of industrial tools that generate SoC implementations of signal-processing applications from models is also very rich, e.g., National Instruments LabVIEW Communications System Design [Labview, 2017], MATLAB (MAtrix LABoratory) [Mathworks, 2017], GNU Radio [GNURadio, 2017]. While our compilation approach targets multi-processor architectures, these tools translate models that describe the functionality of a system to be executed onto single-processor architectures where data are processed onto a single unit.
The MPSoC Application Programming Studio (MAPS) [Leupers et al., 2017, Sheng et al., 2013] is a work that shares many commonalities with our approach. MAPS is a compilation framework for heterogeneous MPSoCs that targets streaming applications. The most important difference between our contributions and MAPS is the input formalism. MAPS takes as input specifications written in the CPN (C for Process Networks) programming language. The latter is an extension of the C language that adds features to describe the modeling constructs (processes and channels) of Kahn Process Networks [Kahn, 1974]. MAPS' core component is the CPN-to-C compiler. It scans and parses CPN input files and first builds an Abstract Syntax Tree (AST) for each processor in the target platform. The AST is transformed to produce plain C code, where the CPN extensions are transformed into platform-specific API calls (e.g., FIFO primitives). In our case, the data-flow semantics of DIPLODOCUS UML/SysML diagrams is equivalent to that of KPNs while it allows to handle more complex communication schemes (e.g., DMA transfers).

3 MODEL-BASED PROGRAMMING FLOW

In the embedded systems community, nearly 90% of the software developers nowadays use C and C++ [**?**] that have an underlying sequential programming model. Nevertheless, multi-processor platforms cannot be efficiently programmed with the constructs proposed by classical sequential programming languages (e.g., functions in C, methods in C++, arrays in C/C++). The reason being that, such constructs have a low level of abstraction that is not sufficient to tackle the complexity of the services and hardware computing paradigm offered by multi-processor architectures. These constructs cannot be translated by compilers into executable instructions in a way that improves the quality and productivity of software, from the user perspective. For these reasons, we advocate that modeling languages (e.g., UML/SysML/MARTE, SDF, KPN, AADL, Simulink) that provide higher-level abstractions (e.g., channels, actors) are more suitable to the task of programming structural aspects (e.g., memory footprint, scheduling) of multi-processor platforms.
Fig. 1 illustrates the methodology that we propose to develop executable implementations from system-level specifications expressed in a modeling language. These specifications are written in a MDE development environment, step (1) in Fig. 1, such as those described in [Gerstlauer et al., 2009]. Here, the system under design is described in a modeling language where models are used to capture the system's functionality

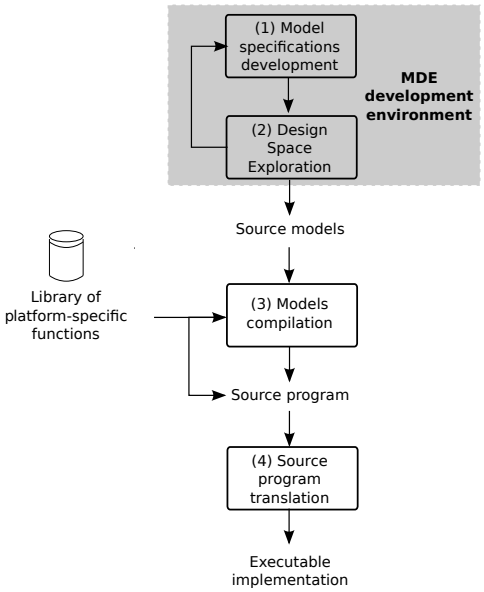

Fig. 1. The software development flow of executable implementations from system-level models.

(i.e., behavior) and the target platform (i.e., the behavior and structure of the available resources). In this phase, models are used as the primary artifact for software development. Models are created, edited and debugged (e.g., formal verification, simulation, profiling) until legal specifications are obtained. Such specifications respect the syntax and semantics of the modeling language and design constraints. This is similar to the way code is created, edited and debugged in traditional Integrated Development Environments (IDE) such as Eclipse [The Eclipse Foundation, 2017]. Subsequently, step (2) in Fig. 1, a Design Space Exploration (DSE) phase takes place, where alternative partitions (i.e., mappings) of the system's functionality onto the available resources are explored until a solution that satisfies some desired requirements (e.g., power consumption, latency, throughput) is found to be realized.

Next, model-based specifications for the best mapping are compiled into a source program, step (3) in Fig. 1, expressed in a programming language (e.g., C/C++). From this point, code becomes the primary artifact for software development as in traditional software engineering methodologies. Further development may take place where, for instance, the program produced by the model compiler is manually completed with code from an external library (e.g., I/O specific code, platform-specific code for OS or middleware).

A final implementation is produced by means of a further translation step, (4) in Fig. 1. This implementation can be realized entirely in software (e.g., an application that runs on top of an Operating System onto a general-purpose control processor) or in hardware (e.g., a hardware IP-based design) or both (e.g., some functionalities are executed by a general-purpose control processor and some are accelerated in hardware). In the case of

implementations that require some functionality to be realized in hardware, the translation is performed by a Computer Aided Design (CAD) toolsuite (e.g., Xilinx Vivado High Level Synthesis). In case of pure software implementations, the translation occurs by means of a traditional programming-language compiler (e.g., GNU/gcc/g++, clang, TurboC). In the next section, we describe the generic structure of a sytem-level model compiler.

4 THE STRUCTURE OF A MODEL COMPILER

As shown in Fig. 2, the structure of the compiler that we propose in this paper is inspired by those of classical compilers [Torczon and Cooper, 2007] for programming languages. The main difference lies in the level of abstraction at which our compiler operates, known as *system-level* [Gerstlauer and Gajski, 2002]. Classical compilers target uni-processor platforms centralized around a single processing element (e.g., a general-purpose control processor, a specialized Digital Signal Processor - DSP) that is connected via a bus to a memory unit. On the contrary, the scale at which a model compiler operates is the one of an entire multi-processor platform that is composed of several computation, communication and storage units. For instance, the back-end in a programming language compiler generates code that allocates array pointers to CPU registers. In evolution with this, code produced by the back-end in a model compiler, allocates data arrays to entire memory areas (e.g., banks in a DRAM memory).
In the following, we present an overview of such a model compiler, regardless of the input modeling language. We conclude this section with a discussion about the impact of modeling languages and programming languages onto the implementation of a multi-processor compiler.

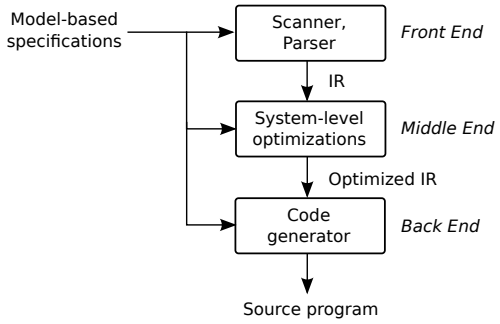

Fig. 2. The structure of a system-level compiler.

4.1 Overview

The **front-end** in Fig. 2 is dedicated to "understanding", with scanning and parsing techniques, input specifications denoted in a specific modeling language. Knowledge

about the structure and behavior of the system under design (e.g., dependencies between functions, mapping constraints) is encoded into Intermediate Representations, IR in Fig. 2 (e.g., a directed graph). By definition, in programming-language compilers, an Intermediate Representation is an abstraction of the given source code. In the context of modeling-language compilers where the input specifications are models themselves, also IRs are models. The IR's structure or the modeling language used to express such an IR thus becomes the IR's metamodel. It follows that the process of producing IRs from input models is a model-to-model transformation.

The purpose of the **middle-end** in Fig. 2 is to attempt to rewrite IRs in a way that is more convenient to optimize the performance of the final implementation in terms of memory management, power consumption, throughput, etc. Such a rewriting results in a second intermediate representation, Optimized IR in Fig. 2. This optimized IR must respect the partitioning of functions onto resources defined by the Design Space Exploration phase (Fig. 1), in order to comply to the DSE's design constraints (e.g., latency, throughput, power consumption). For instance, if this partitioning is static, a function f that has been allocated to unit u_1, f cannot migrate to a different unit u_2 at run-time. Examples of optimizations that can be performed at this stage are: optimizations that reduce the memory occupancy of storage units, scheduling optimizations that minimize the workload of processing and communication units. To perform these optimizations, as shown in Fig. 2, the middle-end also takes as input models of the system's platform and mapping. This is not the case of middle-ends for uni-processor compilers, where only the back-end takes as input information about the target platform.

In terms of the granularity level of optimizations, working at system-level of abstraction allows to focus on parallelism patterns (e.g., task-level) that are more coarse-grained than those in classical programming-languager compilers (e.g., basic blocks). As a consequence, the compiler benefits from managing smaller IRs with respect to equivalent IRs produced from input specifications based on programming languages. In turn, this enables the compiler to implement more agressive performance evaluation, data-flow analysis (e.g., global analysis) and program transformations.

In Fig. 2, the **back-end** is a code generator that translates a mapping configuration into a program (Source progam in Fig. 2) expressed in a high-level programming language (e.g., C/C++). It takes as input both a library of platform-specific functions and models for the platform, mapping and functionality of the system under programming, from the MDE development tool. The souce program must be behaviorally equivalent to the intermediate representations and the input models. The back-end schedules the execution of computation and communication operations, manages the allocation of physical memory regions and selects constructs of the source program's language that correspond to those in the IRs.

In multi-processor platforms, processing units are typically provided with dedicated tool-chains (e.g., C compilers, linkers and assemblers). The model compiler's back-end generates code that is further translated by these processor-specific tool-chains (Source program translation in Fig. 1). In this sense, a model compiler can be seen as a *meta-compiler* that coordinates standard uni-processor compilers.

4.2 Discussion

Generally speaking, the input specifications' formalism impacts a compiler's intermediate representations and the functionality implemented by each building block (front-end, middle-end and back-end). Three types of formalisms can be used to program multi-processor platforms: modeling languages (e.g., UML/SysML/MARTE, SDF, KPN, AADL, Simulink), parallel programming languages (e.g., POSIX threads [**?**], MPI [**?**], OpenMP [**?**]) and sequential programming languages (e.g., C/C++). In this subsection we address two aspects that have the most profound impact: (i) how parallelism is expressed and (ii) the granularity level of a formalism's constructs. For a more in-depth discussion we refer the reader to [**?**]

How parallelism is expressed influences the functionality implemented in the compiler's front-end. In this context a marking line can be drawn between sequential and parallel formalisms. When sequential programming languages are used, the compiler's front-end must implement dedicated techniques to extract parallelism from sequential code (i.e., static analysis, dynamic analysis, speculative execution). Conversely, this added complexity is absent with formalisms that naturally express parallelism.

The second most important criteria that impacts a compiler's implementation is the granularity level offered by a formalism's constructs to manipulate data and control instructions (e.g., data structures and operations on these data structures). This impacts the size and expressiveness of the compiler's intermediate representations, the optimizations that can be performed on these IRs as well as the source code that can be generated by the compiler's back-end. In this context, a marking line can be drawn between modeling and programming languages. Modeling languages typically abstracts away the low-level constructs used by programming languages to manipulate data and control instructions (e.g., pointers, variable initialization). As such, intermediate representations that are derived from modeling languages are lighter (e.g., graphs of smaller size) and can be used to perform more aggressive optimizations (e.g., global data-flow analysis). The implementations of these optimizations in the compiler's middle-end is significantly facilitated by the absence of pointers in modeling languages. However, the abstractions offered by modeling languages often prevent the compiler's back-end to generate complete software implementations. These implementations must be either manually completed by the user, or require the user to accompany the input models with specifications written in programming languages that are then used by the compiler's back-end to produce a complete implementation (e.g., Alf for fUML). In either case, the additional specifications correspond to low-level operations that manipulate and access data structures (e.g., arrays, structs, lists).

5 MODEL-BASED DEVELOPMENT IN TTOOL/DIPLODOCUS

In the context of our research on the efficient programming of signal-processing systems for MPSoC platforms, we selected UML/SysML as modeling language and TTool/DIPLODOCUS as a development environment for model-based specifications.

The choice of using UML/SysML as modeling language is motivated by the following reasons. First, it is widely used in the software development community where it was initially created as a specification language to abstract large software systems. Here,

UML/SysML Activity diagrams have become an established notation to capture control and data-flow at various level of abstractions. Their semantics makes them particularly suited to describe the behavior of the systems we target. With respect to sequential programming languages, UML/SysML diagrams express parallelism explicitly and offer richer contructs than other concurrent modeling languages such as SDF [Lee and Parks, 1995] and KPN [Kahn, 1974]. The latter do not express the internal behavior of processing and communication operations that are modeled in terms of black-boxes interconnected by data dependencies. Instead, such an internal behavior can be expressed by means of UML/SysML Activity diagrams that a compiler's middle-end can analyze in the attempt to optimize the system's behavior.
TTool/DIPLODOCUS [TTool, 2017c] is a framework for the hardware/software co-design of data-flow systems from UML/SysML diagrams. It was selected as it is open source, lightweight and offers system-level modeling and debugging features that are unavailable in concurrent tools for UML design of embedded systems. More precisely, TTool [TTool, 2017a] is the name of the toolkit that allows to create, edit and validate UML/SysML diagrams for different profiles (e.g., Avatar [TTool, 2017b], SysMLSec [TTo, 2017]). DIPLODOCUS [Apvrille et al., 2006, Apvrille, 2008] is the profile dedicated to the hardware/software co-design of data-flow systems.

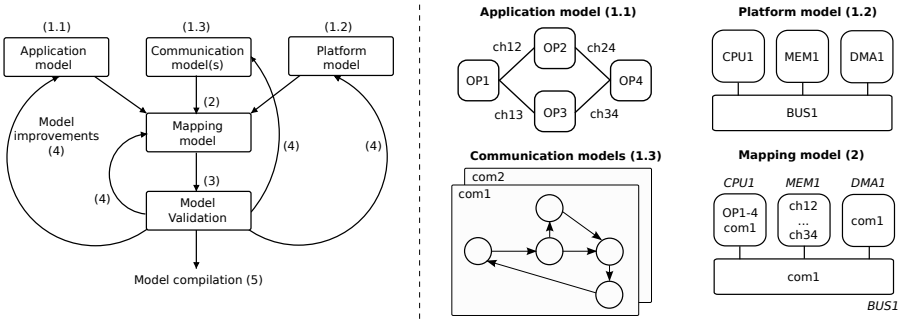

Fig. 3. The Ψ-chart development methodology (left side) and a graphical visualization of its constituent models (right side) that are supported by TTool/DIPLODOCUS.

Fig. 3 shows the model development methodology (called Ψ-chart [Enrici et al., 2017]) that is supported by TTool/DIPLODOCUS to design source models for the compiler OMC, Section 6. In TTool/DIPLODOCUS, the functionality of the system under design (Application model, 1.1 in Fig. 3) 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 model is based on 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 choices based on the value of random variables).
- Algorithmic abstraction: algorithms are described using abstract cost operators that express the complexity of processing items in terms of the number of operations required to transform data (e.g., number of integer operations).

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 resources of the system under design (Platform model, 1.2 in Fig. 3) are denoted with 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 validation and debugging purposes (e.g., simulation, formal verification).
The communication protocols (e.g., DMA transfers) are captured with dedicated diagrams (Communication models, 1.3 in Fig. 3) separately from the application and platform [Enrici et al., 2014]. UML Activity Diagrams capture the steps that compose a communication protocols (e.g., configuration, acknowledgement) while UML Sequence diagrams denote message exchanges among master and slave components.
A partitioning of the system's functionality onto the available resources (Mapping model, 2 in Fig. 3) is created from a 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.
Designs in TTool/DIPLODOCUS can be validated (step 3 in Fig. 3) 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). Validation can be performed both manually via the tool's GUI or automatically via a set of scripts that configure the simulation and formal verification engines to evaluate different mapping alternatives. The simulator's GUI also provides a rich set of debugging features that, for instance, allow the user to animate and execute UML/SysML diagrams step-by-step, inspect variables' values, save and load execution traces.
Models are iteratively debugged, validated and improved (steps 3-4 in Fig. 3), until legal specifications can be compiled into implementations (step 5 in Fig. 3).

6 OMC: THE OPTIMIZING MODEL COMPILER

In this section, we describe an implementation of the model compiler in Fig. 2 that results into OMC (Optimizing Model Compiler), Fig. 4. The current implementation is the result of a continuous research work that first started with the code generation engine in [Enrici et al., 2017]. With respect to the latter, OMC's middle-end has

been extended with memory allocation optimizations and the back-end has been extended with the capability to target any platform whose architecture can be designed in TTool/DIPLODOCUS. Similarly to Integrated Development Environments for programming languages (e.g., Eclipse), TTool/DIPLODOCUS allows to select OMC as model compiler by means of a plugin.

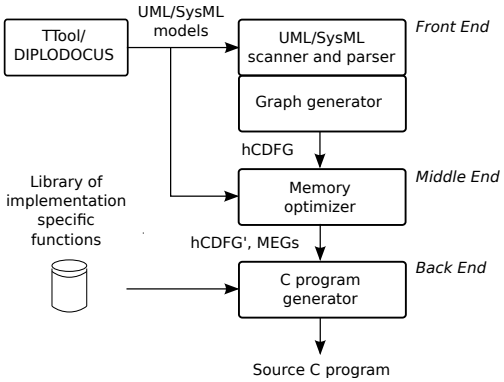

Fig. 4. The structure of OMC, the Optimizing Model Compiler

6.1 Front-end

The front-end in Fig. 4 converts UML/SysML models[4] that describe a system's functionality into a first intermediate representation: hCDFG, a hierarchical Control and Data Flow Graph. The latter is defined as a directed multigraph $G =< N, E >$. N is a set of nodes derived from a DIPLODOCUS Block Definition diagram that capture both data and control operations; each node $n \in N$ is itself a graph that describes the internal behavior of an operation (derived from DIPLODOCUS Activity diagrams). E is a set of edges that capture the data and control dependencies among operations (relations among DIPLODOCUS Blocks). This intermediate representation is derived from UML/SysML diagrams which have a data-flow execution semantics. As a consequence, the hCDFG's has the execution semantics of a Synchronous Data-Flow (SDF) graph [Lee and Parks, 1995]. In a SDF graph, nodes (actors) represent processing entities interconnected by a set of First-In First-Out (FIFO) data queues. An actor starts execution (firing) when its incoming FIFO(s) contains enough tokens, it cannot be preempted and produces tokens onto its outgoing FIFO(s). The number of tokens consumed/produced by each firing is a fixed scalar that is annotated with the graph edges. As actors have no state in the SDF Model of Computation (MoC), if enough tokens are available, an actor can start several executions in parallel. For this reason, SDF graphs naturally express the parallelism of signal-processing applications and can be statically

[4] These models are specified in .xml format by TTool/DIPLODOCUS.

analyzed during compilation for memory allocation optimizations.

6.2 Middle-end

In this version of the compiler's middle-end, Fig. 5, we propose a system-level memory optimizer that minimizes the footprint of the *logical* buffers associated to edges in the hCDFG. We differentiate between *logical* and *physical* buffers. A physical buffer defines a range of memory addresses of a physical memory device (e.g., a Random Access Memory - RAM). A logical buffer, instead, is a virtual address space that can be mapped onto one or multiple physical buffers.

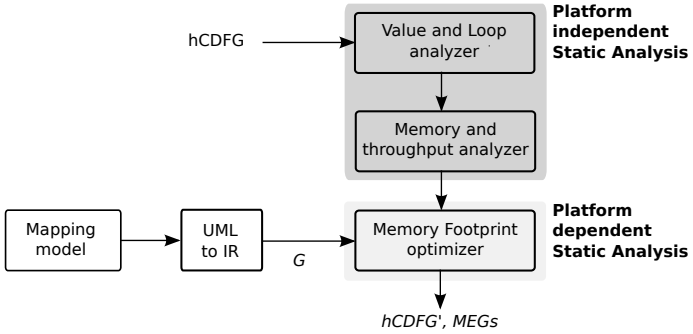

Fig. 5. The Optimizing Model Compiler's middle-end

Value and Loop analyzer Value analysis in traditional program analysis attempts to determine the values in a processor's registers for every program point and execution context [Torczon and Cooper, 2007]. Similarly, this step aims to assign a value to the control variables that determine the size and direction of the data-flows among signal-processing operations, for every execution scenario. The analyzer starts from the hCDFG source vertex' graph. The latter is traversed and a value is assigned for all its variables. This determines a value to the consumption and production rates of hCDFG edges as well as to the number of initial tokens that the source vertex exchanges with its neighbors. The latter are iteratively examined and the variables' values are propagated to their neighbors until the hCDFG sink vertex is reached. In case these values cannot be determined exactly (e.g., presence of random operators in the input models), the analyzer attempts to determine a lower and an upper bound.

Memory and throughput analyzer The results of the Value and Loop analyzer are used to determine the memory footprint and the throughput of each hCDFG vertex. The throughput is given by the amount of data that is exchanged on the edges of

a hCDFG node. As edges in the hCDFG are associated to logical First-In-First-Out (FIFO) buffers, the throughput of all edges of a hCDFG node determines its memory footprint. The throughput and the memory footprint is computed by multiplying the tokens exchanged on input/output edges by the size of the tokens' data type that is specified in the input DIPLODOCUS models (e.g., int16, cpx32).
The values determined by the Value and Loop analyzer and by the Memory and Throughput analyzer are annotated to the hCDFG graph that results into hCDFG'.

Memory footprint optimizer Our optimizer implements a variant of the allocation techniques presented in [Desnos et al., 2014] that we adapted to allow the sharing of input and output buffers of actors, similar to one of the memory reuse techniques presented in [Desnos et al., 2016b]. Essentially, the optimizer performs a series of graph transformations to deduce a set of graphs that specify relations among logical buffers that can or cannot share physical memory.
The hCDFG graph in Fig. 4 is transformed first into a single-rate SDF, where the production and consumption rates on each FIFO are made equal. The single-rate SDF is transformed into a Direct Acyclic Graph (DAG) by isolating one iteration of the single-rate SDF and by ignoring FIFOs with initial tokens. The DAG graph contains two types of memory objects:

- Communication buffers that are used to transfer tokens between consecutive actors.
- Feedback/pipeline buffers that store feedback FIFOs, i.e., buffers corresponding to (feedback) edges whose input and output port are associated with the same actor.

Our work differs from [Desnos et al., 2014] as, in the latter, a DAG also expresses an estimation of an actor's internal memory (e.g., the stack space of a task allocated by an Operating System). This is because we target platforms composed of IP blocks (Section 7) for which OMC does not need to allocate extra buffers for the IPs' internal working memory.
From the mapping model, scheduling information is added to the DAG. Subsequently, a Memory Exclusion Graph (MEG) is derived. Nodes in the MEG represent logical memory objects: FIFO buffers whose size is equal to the number of tokens in the single-rate SDF. Edges in the MEG link logical FIFO buffers that cannot be allocated to overlapping physical buffers. The MEG is then updated with mapping information from the input models that specifies the execution constraints (scheduling) for each signal-processing operation. This allows to remove edges (exclusion relations) between nodes in the MEG. The purpose of this operation is to merge logical buffers so that physical buffers in the executable code can share common memory regions, thus reducing the total footprint of the software application produced by OMC.
At this point, the heuristics proposed in [Desnos et al., 2014] is applied to compute a lower bound for the memory of the physical buffers. This bound is defined in [Fabri, 1979] as the weight of a Maximum Weight Clique (MWC). A *clique* is a subgraph of MEG vertices within which each pair of vertices is linked with an edge. As the memory objects of a clique cannot share memory space because they mutually exclude each other, the weight of a clique gives a lower bound to the amount of memory that must be allocated for all of the clique's buffers. This amount is equal to the sum of the sizes of

all clique's buffers. The pseudo-code of the heuristics proposed in [Desnos et al., 2014] is shown in Algorithm 1.

Algorithm 1: The MWC heuristics

```
   /* C = the clique                                                    */
   /* nb_edges = number of edges in C                                   */
   /* cost(·) = cost function of C                                      */
   /* v = generic vertex in C                                           */
   /* w(v) = weight of vertex v                                         */
   /* N(v) = neighbor vertices of v                                     */
   /* |N(v)| = lowest number of v's neighbors                           */
1  C ← V
2  nb_edges ← |E|
3  foreach v ∈ C do
4  |  cost(v) ← w(x) + Σ_{v'∈N(v)} w(v')
5  end
6  while |C| > 1 and 2·nb_edges / (|C|·(|C|−1)) < 1.0 do
7  |  Select v* from V that minimizes cost(·)
8  |  C ← C\{v*}
9  |  nb_edges ← nb_edges − |N(v*) ∩ C|
10 |  foreach v ∈ {N(v*) ∩ C} do
11 |  |  cost(v) ← cost(v) − w(v*)
12 |  end
13 end
14 Select a vertex v_random ∈ C
15 foreach v ∈ {N(v_random)\C} do
16 |  if C ⊂ N(v) then
17 |  |  C ← C ∪ {v}
18 |  end
19 end
```

In each iteration of the main loop (lines 6-13) in Algorithm 1, minimum cost vertices v^* are removed from C (line 8). If multiple vertices have the same cost, the vertex v with the lowest number of neighbors $|N(v)|$ is removed. If the number of neighbors is equal, then the vertex v with the smallest weight $w(v)$ is removed. If there are still multiple vertices with equal properties, a random vertex v_{random} is selected. The loop iterates until the vertices in C form a clique. This condition is verified, line 6, by comparing the edge density of a clique with the edge density of the MEG subgraph formed by the remaining vertices in C. The edge density of a clique is defined as the ratio between existing exclusions and all possible exclusions. Such density is equal to 1.0 in the case of the complete MEG. The number of edges, nb_{edges}, is decremented at line 9 by the number of edges in L that link the removed vertex v^* to vertices in C. Lines 10-12 update the costs of the remaining vertices for the next iteration. The complexity of the heuristic algorithm is of the order of magnitude of $O(|V|^2)$, where $|V|$ is the number of

vertices of the MEG subgraph.
This MEG is further split into separates MEGs, one for each memory unit in the target platform onto which physical buffers must be allocated. This split is perfomed according to the algorihm described in [Desnos et al., 2016b], where FIFO buffers whose producer and consumer are mapped onto different processors are duplicated in different MEGs.

6.3 Back-end

OMC's back-end, Fig. 6, translates hCDFG' and the MEGs into a C program that can run as a user-space application on top of a general-purpose Operating System. The back-end is composed of three code generators. The Memory Manager code generator generates a static memory allocation for the buffers in the Memory Exclusion Graph. Buffers whose lifetimes overlap are assigned to dedicated memory areas, whereas buffers whose lifetimes do not overlap are assignd to shared memory areas. The Scheduler code generator produces a static scheduling of operations that corresponds to the scheduling used by the middle-end to produce the MEG graphs. The Operation code generator translates each operation in hCDFG' with 3 C routines for initialization, execution and clean-up purposes (Fig. 4). Initialization and clean-up routines are called once, when the program starts and terminates, respectively. These routines manipulate the software data structures that are needed by processing units in the target platform to prepare and clean up the execution of a node in hCDFG'. Execution routines (implementation-specific functions in Fig. 4) are added to each actor from an external library. They trigger the execution of an operation on the hardware.

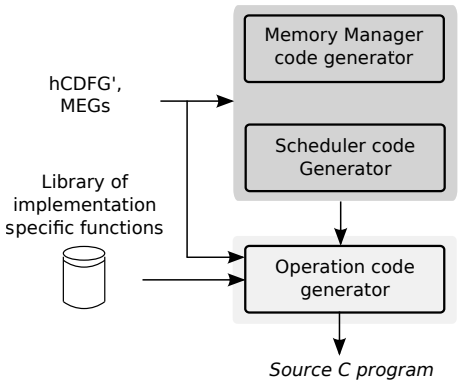

Fig. 6. The Optimizing Model Compiler's back-end

In the current implementation of the compiler, the body of execution routines must be manually written by a user and included in the final source code via dedicated C header files. These execution routines specify, at a lower level of abstraction, the implementation details of the signal-processing algorithms that are described, at system-level, by

the input UML/SysML diagrams. Given the real-time nature of the systems that we aim to program, it is mandatory to specify these algorithms with a less abstract language (i.e., C) that offers constructs which match more closely the characteristics of the underlying hardware execution platform (e.g., memory alignment of struct fields to accelerate DMA transfers). In analogy with traditional C programs that embed assembly code for functionalities that are time-critical, our UML/SysML programs embed C code for functionalities that are time-critical at system-level of abstraction.

6.4 Discussion

In this version of the model compiler, we did not include any environment for the analysis of the IRs' models and their transformations as this goes out of the scope of our current research interests. As described in [Floch et al., 2011], techniques such as generative approaches, model mapping, Domain Specific Languages and metamodel instrumentation exist to guarantee the correctness and maintainability of IR transformations. However, due to scalability reasons, their use is difficult to apply to research compilers. It is, however, a practically surmountable problem that can be solved by developing additional features to the model compiler. In the context of the case study of Section 7, we manually verified the equivalence between (i) the relations in graphs hCDFG and hCDFG' (ii) the scheduling of operations in the output C program and (iii) the inter-operation dependencies in the input models.

Portability This implementation of the model compiler addresses platforms where the scheduling of operations is centrally executed by a single general-purpose control processor. The latter configures and dispatches the execution of operations to a set of physically distributed units (e.g., DSPs, DMAs, IPs), according to events generated upon the consumption/production of data by computation and communication operations. For each platform, a dedicated library of implementation-specific functions must be provided by re-using those from other projects as templates. To target designs where the control code of an application is fragmented into separate executables that each run onto different CPUs, OMC must be extended to produce multiple executables, include synchronization primitives among multiple units, etc..
In order to use this OMC's implementation with a MDE tool other than TTool/DIPLODOCUS, the user needs to write a new plug-in for the front-end's scanner and parser. The existing plug-in can be used as a template to reduce development efforts.

Debugging In our model-based programming approach, debugging is done at different locations: in TTool/DIPLODOCUS, in the output C program (OMC's output, Fig. 6) and the library's implementation-specific functions (e.g., Valgrind, gdb). Transformations of the Intermediate Representations can be manually debugged by comparing the data-flow relations among nodes in hCDFG, hCDFG' and those between SysML blocks in the input models. Also, simulation and formal verification techniques in TTool/DIPLODOCUS can be used to guarantee the correctness of the input UML/SysML models with respect to design requirements.

7 CASE STUDY

According to the methodology in Fig. 1, we used TTool/DIPLODOCUS and OMC to develop software from the UML/SysML models of a 5G decoder for the uplink (SC-FDMA), single antenna case, Physical Uplink Shared channel (xPUSCH), based on the specifications in [Verizon, 2015].

The algorithm of the signal-processing operations (**application**) that compose the 5G decoder is shown in Fig. 7. Data coming from the air are received and converted into digital samples in the `RF/ADC` block (not considered in our implementation). Digital samples are converted to the frequency domain (`DFT`) information carriers are demapped (`Sub-carrier Demapping`) and samples converted back to the time domain (`IDFT`). The flow of received information is demodulated (`64QAM Demodulation`), decoded (`LDPC Decoder`) and assembled (`Code Block Concatenation`, `Remove CRC blocks`) into a transport block to be further processed by higher network layers.

We captured this application with a SysML Block Definition diagram containing 11 SysML Composite Block Components (1 for each signal-processing operation in Fig. 7 as well as one source and one sink components). Each Composite Block Component contains 2 SysML Primitive Block Components that are, in turn, each associated to a UML Activity diagram. By way of example, Fig. 8a shows the SysML block components of operation `64QAM Demodulation`. Here, the primitive block `F_QAMDemod` denotes the ensemble of the control operations (e.g., determining the amount of samples to process) that govern the primitive block `X_QAMDemod`. The latter captures the processing operations that transform input samples into demodulated output samples. Fig. 8b shows the the UML Activity diagrams of the primitive block `X_Demod`. This UML Activity diagram is representative of the behavioral models of all the 5G decoder operations. Here, the amount of input and output samples is received from `F_QAMDemod`, operator (1) in Fig. 8b. Subsequently, a for-loop (2 in Fig. 8b) iteratively reads `numBitsPerSampleIN` samples from an input channel (3 in Fig. 8), processes them (4 in Fig. 8) and writes `numBitsPerSampleOUT` samples to an output channel (5 in Fig. 8). In Fig. 8, variables `numBitsPerSampleIN` and `numBitsSampleOUT` model 32-bits samples. The algorithm of the demodulation operations in Fig. 8 is abstracted by means of the cost operator `EXECC`(min, max). The latter captures the complexity of an algorithm in terms of its minimum and maximum number of operations onto complex data.

Table 1 lists the data produced and consumed by operations in Fig. 7, for an input subframe composed of 14 OFDM symbols and 41 LDPC code blocks.

In this case study we programmed two **target platforms**. One is **Embb** [Embb, 2017], a generic baseband architecture dedicated to signal-processing applications. Embb is composed of a Digital Signal Processing (DSP) part and a general purpose control part. The DSP part is composed of a set of Digital Signal Processing Units (DSPUs) interconnected by a crossbar. Each DSP unit is equipped with a Processing Sub-System (PSS) as computational unit, a Direct Memory Access controller (DMA) and a local memory called the Memory Sub-System (MSS). These DSPUs can be seen as programmable IPs that are more flexible than traditional fully hard-wired accelerators. The general purpose control part is composed of a RAM memory and of a CPU that configures and controls the processing operations performed by the DSPUs and the data transfers.

The architecture of the second target platform, a hardware **IP-based platform** is com-

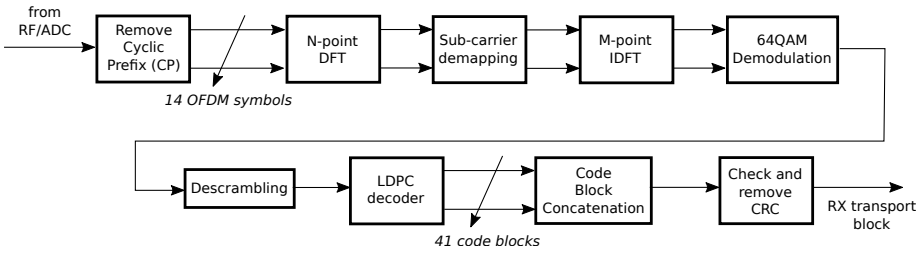

Fig. 7. The block diagram of the 5G decoder that is designed and programmed in this case study.

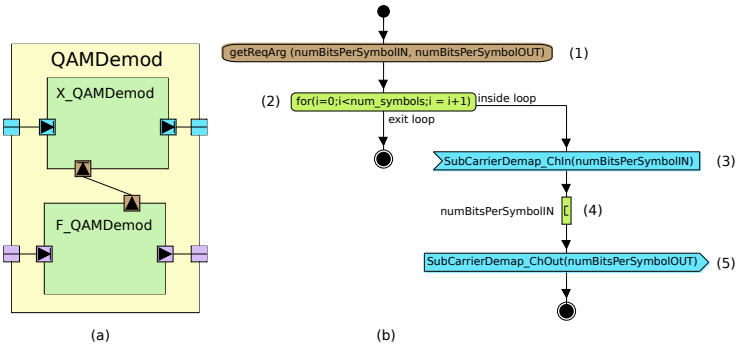

Fig. 8. The SysML block diagram (a) and UML Activity (b) diagram for operation `64QAM Demodulation`.

Table 1. Input/Output data of the decoder operations

Operation	Input	Output
Remove CP	30720 samples (14 OFDM symbols)	2048 samples (1 OFDM symbol)
DFT	2048 samples	2048 samples
Sub-carrier demapping	2048 samples	1200 samples
IDFT	1200 samples	1200 samples
Demodulation	1200 resource elements	7200 soft bits
Descrambling	7200 soft bits	7200 soft bits
LDPC decoder	1944 soft bits	1620 hard bits
Code Block Concatenation	1620 hard bits	66416 hard bits
Remove CRC	66416 hard bits	66392 hard bits

posed of a programmable and of a configurable subsystem. The programmable subsystem executes control functions as well as signal-processing operations whose performance are not time critical. It is composed of a CPU and a RAM memory. The configurable subsystem accelerates performance-critical operations onto a dedicated hardware IP block that can be selected by Xilinx SDx [Xilinx, 2017] from the source C program produced by OMC. An IP block includes a processing core, a local memory and a DMA engine, similarly to a DSPU in Embb.
Thanks to the similarities in the structure of the two target platforms, we captured their architecture in the UML Deployment diagram of TTool/DIPLODOCUS in Fig. 9. In

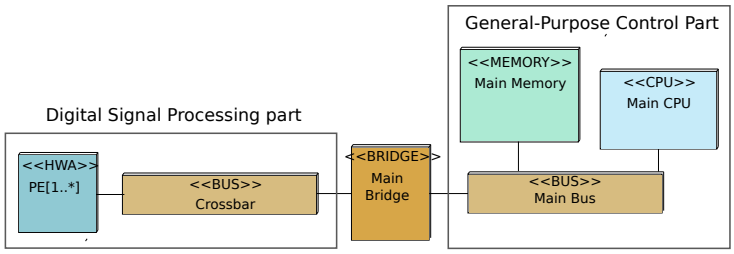

Fig. 9. The UML Deployment Diagram of a generic instance of Embb and of the hardware IP-based platform.

Fig. 9, the left-hand part describes the subsystem where the processing of data is accelerated. Here, a PE (Processing Element) block models the architecture of a DSPU in Embb or a hardware IP block. The TTool/DIPLODOCUS model of a PE's internal architecture is depicted in Fig. 10. The right-hand side of Fig. 9 captures the control part of our two target platforms: a CPU and a memory units interconnected by a bus unit.
To program the two platforms, we instantiated a model such as the one in Fig. 9 that contains two Processing Elements for Embb and one Processing Element for the IP-

based platform. The mapping information that results from the Design Space Exploration phase (Fig. 1) for each platform is listed in Table. 7.

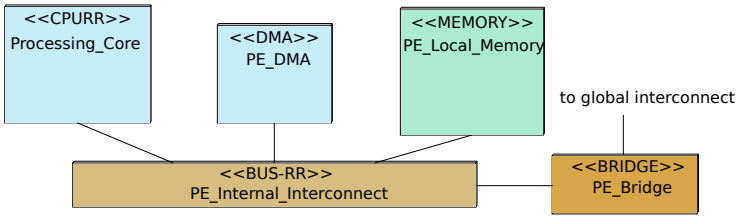

Fig. 10. The UML Deployment Diagram for the generic architecture of a Processing Element (PE) in Fig. 9.

Table 2. Memory footprint of the 5G decoder logical buffers and mapping configuration

Operation	**Memory footprint [bytes]**		**Mapping**	
	Input	**Output**	**Embb**	**IP-based platform**
Remove CP	122880	14×8192	Main CPU (sw)	Main CPU (sw)
DFT	8192	8192	FEP DSPU (hw)	Main CPU (sw)
Demapping	//	//	Data transfer	Main CPU (sw)
IDFT	4800	4800	FEP DSPU (hw)	Main CPU (sw)
Demodulation	4800	7200	FEP DSPU (hw)	Main CPU (sw)
Descrambling	7200	7200	Main CPU (sw)	Main CPU (sw)
LDPDC Decoder	1944	202.5	LDPC DSPU (hw)	IP block (hw)
Code Block Concatenation	202.5	8302	Main CPU (sw)	Main CPU (sw)
Remove CRC	8302	8299	Main CPU (sw)	Main CPU (sw)

7.1 The model compilation

The optimization techniques used by our model compiler reduce the memory footprint by sharing the physical buffers among operations that are mapped to a given execution unit. To understand this optimization, Table 7 also shows the memory footprint of the logical buffers for the 5G decoder operations.
In the case of Embb, in Table 7, three sets of logical buffers can be identified, $\mathscr{B}_0$, $\mathscr{B}_1$, $\mathscr{B}_2$, that are associated to operations mapped onto the `Main CPU`, the `FEP DSPU` and the `LDPC DSPU`, respectively. For the `Main CPU` the logical buffers are $\mathscr{B}_0 =$ `{RemoveCP--DFT, Demodulation-Descrambling, Descrambling-LDPCDecoder, LDPCDecoder--CodeBlockConcatenation, CodeBlockConcatenation-RemoveCRC}`. For unit `FEP DSPU`, the buffers are $\mathscr{B}_1 =$ `{RemoveCP-DFT, DFT-IDFT, IDFT-Demodulation, Demodulation--Descrambling}`. For unit `LDPC DSPU` the buffers are $\mathscr{B}_2 =$ `{Descrambling-LDPCDecoder,`

LDPCDecoder-Descrambling}.
For the IP-based platform, based on the mapping in Table 7, we identify 2 sets of logical buffers, $\mathscr{B}_3$, $\mathscr{B}_4$, that are associated to operations mapped onto the Main CPU and the IP block, respectively. For the Main CPU the logical buffers are $\mathscr{B}_3 =$ {RemoveCP--DFT, DFT-Demapping, Demapping-IDFT, IDFT-Demodulation, Demodulation--Descrambling, Descrambling-LDPCDecoder, CodeBlockConcatenation-RemoveCRC}. For the IP core, the logical buffers are $\mathscr{B}_4 =$ {Descrambling-LDPCDecoder, LDPCDecoder-CodeBlockConcatenation}.
For each set of buffers above, the middle-end produces a Memory Exclusion Graph. As mentioned in Section 6, the middle-end duplicates, in the Memory Exclusion Graphs, buffers whose producer and consumer operations are mapped onto different execution units. For instance, buffer Descrambling-LDPCDecoder is present in both $\mathscr{B}_0$ and $\mathscr{B}_2$, when compiling for Embb, and in both $\mathscr{B}_3$ and $\mathscr{B}_4$, when compiling for the IP-based platform.

When programming Embb, the back-end allocates 8192 bytes for $\mathscr{B}_1$ to the local memories of the FEP DSPU, which is equal to the size of RemoveCP-DFT. It allocates 1944 bytes for $\mathscr{B}_2$ to the local memory of the LDPC DSPU, which is equal to the size of buffer Descrambling-LDPCDecoder, and 8302 bytes for $\mathscr{B}_0$ to the Main CPU memory, which is equal to the size of buffer CodeBlockConcatenation-RemoveCRC. Assigning separate physical buffers to each of the logical buffers would have allocated 50976 bytes to the FEP local memory (the size of all logical buffers in $\mathscr{B}_1$), 2147 bytes to the LDPC processor's local memory (the size of all logical buffers in $\mathscr{B}_2$) and 39399 bytes to the main CPU memory (the size of $\mathscr{B}_0$). Compilation reduces the memory footprint of 83.88%, 9.46% and 78.93% for each of these three units, respectively. Overall, it reduces by 80.07% the memory used by the final executable code, with respect to pure translation-based approaches.
For the IP-based platform, the back-end allocates 8302 bytes for $\mathscr{B}_3$ to the main CPU memory (the programmable system), which is the size of buffer CodeBlockConcatenation--LDPCDecoder, and 1944 bytes for $\mathscr{B}_4$ to the hardware IP-core memory (configurable system), which is the size of buffer DescramblingLDPCDecoder. A pure translation-based approach would have reserved 90375 bytes (the size of $\mathscr{B}_3$) and 2147 bytes (the size of $\mathscr{B}_4$) to the main CPU and the hardware IP-core memories, respectively. Our compilation achieves a memory footprint reduction equal to 90.81% and 9.46%, respectively, for these two units. Overall, this reduces by 88.93% the memory used in the mixed hardware-software implementation.

These advanced memory optimizations were possible thanks to two types of analysis performed by the compiler's middle-end. First, thanks to the lifetime analysis of buffers, that is derived from the data dependencies and the scheduling information as described in Section 6. Secondly, thanks to the static analysis performed on the UML Activity diagrams that describe the internal behavior of each signal-processing operation (Fig. 5). This static analysis allows to retrieve information about the data dependencies that are *internal* to operations. More in details, if the input and output buffers of an operation have disjoint lifetimes (i.e., the input buffer is read before an output

buffer of equal size is first written), these buffers can be allocated in the same physical memory space. For instance, the behavior described by the UML Activity diagram in Fig. 8 for the demodulation operation is representative for the models of all the 5G decoder operations. Here, `numBitsPerSymbolIN` data from the input channel (operator 3 in Fig. 8b) is always read before `numBitsPerSymbolOUT` data are written to the output channel (operator 5 in Fig. 8b). Hence, a subrange in the input logical buffer of size `numBitsPerSymbolIN`/4 bytes can be shared with the output logical buffer, when allocated to physical memory[5]. Similar techniques that take advantage of internal data dependencies are presented in [de Greef et al., 1997, Desnos et al., 2016a].

The middle-end in OMC optimizes an application's memory footprint by accounting for the mapping information of SDF actors onto a platform's execution units. This scheduling update does not impact the overall **timing properties** of the final executable. Specifically to this 5G decoder, its real-time properties are limited by two factors. First, by the lack of parallelism between operations that is inherent to the application in Fig. 7. Secondly, by the absence in the target platforms of multiple units capable to process different OFDM symbols in parallel. Because of the limited size of the FPGAs onto which we prototyped our platforms, it was only possible to instantiate one Front-End Processor unit and one LDPC processor in Embb as well as one hardware IP-block in the second platform (due to Design Space Exploration constraints). For instance, in Embb, the availability of only one FEP unit does not allow to pipeline the execution of operations `DFT`, `Demapping`, `IDFT` and `Demodulation` for consecutive OFDM symbols.

7.2 The target program translation

In the case of Embb, the target C program is translated into an executable with GNU/gcc v.5.4.0 cross-compiled onto Ubuntu v.16.04.4. This executable (a pure software implementation of the input models) runs on the main CPU in Fig. 9 as a user-space application for Linux v.4.4.0-xilinx. In terms of the library of implementation specific functions that are necessary to produce a complete source C program, 371 functions were included in the source C program produced by OMC.
In the case of the IP-based platform, we translate the target C program with Xilinx SDx [Xilinx, 2017] into a mixed hardware-software implementation. The output of the Xilinx SDx translation process are a Linux image and an .elf file for the software part of the implementation, to be executed by the CPU of the programmable subsystem. The executable for the hardware part of the implementation is a FPGA bitstream. The latter is loaded into the target FPGA's configurable fabric by a Linux image that runs onto the FPGA's control processor (not represented in our models). The number of functions included to the source C program produced by OMC amounts to 33.

[5] We remind to the reader than in our DIPLODOCUS model, each operation reads a certain amount of input data, `num_samples`, that are expressed on 32 bits. Thus the size of `num_samples` samples corresponds to `num_samples`/4 bytes.

7.3 Discussion

With respect to programming approaches based on un-optimized model translations (Section 2), we showed that optimizing non-functional properties (i.e., memory footprint) of models can result in significant performance improvement. These improvements do not alter the semantics of the source models. In this case study, models do not require to be accompanied by additional specifications for their semantics to be correctly translated by the compiler. The reason for this is that an invertible mapping relation exists between the constructs of a SysML Block Definition diagram (i.e., blocks and relationships) and structural constructs in C (i.e., functions and arrays in C). This mapping relation, also called *interpretation* [Seidewitz, 2003], gives the models meaning with respect to the system under design that models abstract away. The latter is, ultimately, the software (e.g., C code) and not the signal-processing operations (e.g., a Fast Fourier Transform) of a system under design [Selic, 2003].
However, when this mapping relation is not invertible, code cannot be generated from models without the support of an external formalism that precisely specifies the models' semantics. This is the case for the Action Language Alf [Alf, 2017] and the Foundational Subset for Executable UML, fUML [fUML, 2016]. In our case study, this can be seen if attempting to produce code that implements the internal behavior of signal-processing operations from DIPLODOCUS Activity diagrams (Fig. 8). The DIPLODOCUS profile abstracts away operations that manipulate data and control instructions with a cost operator *EXEC(·)* [Apvrille et al., 2006]. The latter is a natural number that expresses the amount of operations on integer, *EXECI(·)*, or custom *EXECC(·)* numbers. For instance, given two arrays of integers $\bar{A}$ and $\bar{B}$ of size n and m, respectively, the element-wise multiplication of $\bar{A} \times \bar{B}$ would be abstracted as *EXECI*$(n \times m)$. However, the same cost operator may as well represent the element-wise sum $\bar{A} + \bar{B}$, or the element-wise difference $\bar{A} - \bar{B}$. This formally explains the reason why source code produced by OMC from DIPLODOCUS models, must be manually completed by the user with calls to operation-specific functions that contain the C code modeled by DIPLODOCUS Activity diagrams.

8 CONCLUSION

This paper proposes a software development flow to program parallel platforms (e.g., Multi-Processor Systems-on-Chip) from system-level models. The flow we proposed produces application software specified in a thid-generation language (i.e., C). It combines the UML/SysML toolkit TTool/DIPLODOCUS [TTool, 2017a], as model development environment, with OMC (the Optimizing Model Compiler [Enrici et al., 2018]) as a source-to-source compiler that translates UML/SysML diagrams into C code. Our contributions were applied to program a 5G decoder onto a multi-processor platform and onto an IP-based platform. In future work, we will extend our case study with the complete design of an encoder chain and we will extend the middle-end with other optimizations (e.g., power consumption).
Based on the discussion in sub-section 7.3, we conclude that model-based code generation does not always replace software development based on programming languages.

More in details, model-based code generation is an advantageous paradigm when the mapping relation between constructs in the modeling and programming languages is invertible. This occurs between constructs that describe the *structure* of a software system (e.g., blocks in a SysML Block Definition diagram map to C functions and vice-versa, classes in a UML Class diagram map to Java classes and vice-versa). These constructs can be efficiently translated into compiler's IRs and analyzed to optimize structural properties of the system under design. This is exemplified by the memory footprint optimizations that we presented in this paper. Another structural property that can be optimized while increasing the software quality and productivity is the scheduling of the operations executed by the system under design.
Conversely, modeling constructs that capture the *behavioral* aspects of a system under design (e.g., UML Activity diagrams) cannot be invertibly mapped to equivalent constructs in programming languages. Additional languages (e.g., Alf for fUML) must then be used to fill the semantic gap between an input model and its output program.
For these reasons, it is our belief that the quality and productivity of software development for multi-processor embedded systems cannot be improved by relying only on the high-level abstractions (e.g., UML/SysML blocks and channels) offered by modeling languages. The latter must be efficiently integrated by tools and methodologies to the constructs that already exist in sequential programming languages to abstract the manipulation of data and control instructions (e.g., operations on arrays and structs in the C language). We believe that the tools and approach proposed in this paper are a good starting point in this direction, rather than an arrival point.

Bibliography

[TTo, 2017](2017). TTool/SysMLSec. http://sysml-sec.telecom-paristech.fr.

[Alf, 2017]Alf (2017). Action language for foundational uml (alf). http://www.omg.org/spec/ALF/.

[Apvrille, 2008]Apvrille, L. (2008). Ttool for diplodocus: An environment for design space exploration. In *NOTERE*, pages 28:1–28:4.

[Apvrille et al., 2006]Apvrille, L., Muhammad, W., Ameur-Boulifa, R., Coudert, S., and Pacalet, R. (2006). A uml-based environment for system design space exploration. In *ICECS*, pages 1272–1275.

[Bazydlo et al., 2014]Bazydlo, G., Adamski, M., and Stefanowicz, L. (2014). Translation uml diagrams into verilog. In *HSI*, pages 267–271.

[Ciccozzi et al., 2012]Ciccozzi, F., Cicchetti, A., and Sjodin, M. (2012). Full code generation from uml models for complex embedded systems. In *STEW*.

[DaRTteam, 2009]DaRTteam (2009). Graphical array specification for parallel and distributed computing (gaspard2). http://www.gaspard2.org/.

[de Greef et al., 1997]de Greef, E., Catthoor, F., and de Man, H. (1997). Array placement for storage size reduction in embedded multimedia systems. In *ASAP*, pages 66–75.

[Desnos et al., 2014]Desnos, K., Pelcat, M., Nezan, J., and Aridhi, S. (2014). Memory analysis and optimized allocation of dataflow applications on shared-memory mpsocs. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, pages 1–19.

[Desnos et al., 2016a]Desnos, K., Pelcat, M., Nezan, J., and Aridhi, S. (2016a). On memory reuse between inputs and outputs of dataflow actors. *ACM Transactions on Embedded Computing Systems*, pages 30:1–30:25.

[Desnos et al., 2016b]Desnos, K., Pelcat, M., Nezan, J.-F., and Aridhi, S. (2016b). Distributed Memory Allocation Technique for Synchronous Dataflow Graphs. In *SiPS 2016*.

[Embb, 2017]Embb (2017). http://embb.telecom-paristech.fr/.

[Enrici et al., 2014]Enrici, A., Apvrille, L., and Pacalet, R. (2014). A uml model-driven approach to efficiently allocate complex communication schemes. In *MODELS*, pages 370–385.

[Enrici et al., 2017]Enrici, A., Apvrille, L., and Pacalet, R. (2017). A model-driven engineering methodology to design parallel and distributed embedded systems. *ACM TODAES*, 22(2):34:1–34:25.

[Enrici et al., 2018]Enrici, A., Lallet, J., Latif, I., Apvrille, L., Pacalet, R., and Canuel, A. (2018). A Model Compilation Approach for Optimized Implementations of Signal-Processing Systems. In *Modelsward*, pages 25–35.

[Fabri, 1979]Fabri, J. (1979). *Automatic storage optimization*. Courant Institute of Mathematical Sciences, New York University.

[Floch et al., 2011]Floch, A., Yuki, T., Guy, C., Derrien, S., Combemale, B., Rajopadhye, S., and France, R. B. (2011). Model-driven engineering and optimizing compilers: A bridge too far? In *MODELS*, pages 608–622.

[fUML, 2016]fUML (2016). http://www.omg.org/spec/FUML/1.2.1/.

[Gamatie et al., 2008]Gamatie, A., Beux, S. L., Piel, E., Etien, A., Atitallah, R. B., Marquet, P., and Dekeyser, J. L. (2008). A model driven design framework for high performance embedded systems. http://hal.inria.fr/inria-00311115/en.

[Gerstlauer and Gajski, 2002]Gerstlauer, A. and Gajski, D. D. (2002). System-level abstraction semantics. In *15th International Symposium on System Synthesis*, pages 231–236.

[Gerstlauer et al., 2009]Gerstlauer, A., Haubelt, C., Pimentel, A. D., Stefanov, T. P., Gajski, D. D., and Teich, J. (2009). Electronic system-level synthesis methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530.

[GNURadio, 2017]GNURadio (2017). Gnu radio. http://gnuradio.org/.

[Kahn, 1974]Kahn, G. (1974). The Semantics of a Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475.

[Knorreck, 2011]Knorreck, D. (2011). *UML-Based Design Space Exploration, Fast Simulation and Static Analysis*. PhD thesis, Telecom ParisTech.

[Labview, 2017]Labview (2017). Labview communications system design. http://www.ni.com/labview-communications/.

[Lee and Parks, 1995]Lee, E. A. and Parks, T. M. (1995). Dataflow process network. *Proceedings of the IEEE*, 83(5):1235–1245.

[Leupers et al., 2017]Leupers, R., Aguilar, M. A., Eusse, J. F., Castrillon, J., and Sheng, W. (2017). *MAPS: A Software Development Environment for Embedded Multicore Applications*, pages 917–949. Springer Netherlands.

[Mathworks, 2017]Mathworks, T. (2017). https://www.mathworks.com/solutions/model-based-design.html.

[Mellor and Balcer, 2002]Mellor, S. J. and Balcer, L. (2002). *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley.

[Mellor and Balcer, 2003]Mellor, S. J. and Balcer, M. J. (2003). Executable and translatable uml. http://www.omg.org/news/meetings/workshops/\\UML_2003_Manual/Tutorial4-Balcer.

[Mischkalla et al., 2010]Mischkalla, F., He, D., and Mueller, W. (2010). Closing the gap between uml-based modeling, simulation and synthesis of combined hw/sw designs. In *DATE*, pages 1201–1206.

[Moreira et al., 2010]Moreira, T. G., Wehrmeister, M. A., Pereira, C. E., Petin, G. F., and Levrat, E. (2010). Automatic code generation for embedded systems: From uml specifications to vhdl code. In *International Conference on Industrial Informatics*, pages 1085–1090.

[Nicolas et al., 2014]Nicolas, A., Penil, P., Posadas, H., and Villar, E. (2014). Automatic synthesis over multiple apis from uml/marte models for easy platform mapping and reuse. In *Euromicro DSD*, pages 443–450.

[OMG, 2017]OMG (2017). Uml profile for marte: Modeling and analysis of real-time embedded systems. http://www.omg.org/omgmarte/.

[Oracle, 2017]Oracle (2017). Netbeans ide. https://netbeans.org.

[Schmidt, 2006]Schmidt, D. C. (2006). Model-driven engineering. *Computer*, 39(2):25–31.

[Seidewitz, 2003]Seidewitz, E. (2003). What Models Mean. *IEEE Software*, 20(5):26–32.

[Selic, 2003]Selic, B. (2003). The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25.

[Sheng et al., 2013]Sheng, W., Schürmans, S., Odendahl, M., Bertsch, M., Volevach, V., Leupers, R., and Ascheid, G. (2013). A Compiler Infrastructure for Embedded Heterogeneous MPSoCs. In *PMAM*, pages 1–10.

[Tan et al., 2004]Tan, W. H., Thiagarajan, P. S., Wong, W. F., Zhu, Y., and Pilakkat, S. K. (2004). Synthesizable systemc code from uml models.

[The Eclipse Foundation, 2017]The Eclipse Foundation (2017). Eclipse. http://www.eclipse.org.

[Torczon and Cooper, 2007]Torczon, L. and Cooper, K. (2007). *Engineering a Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition.

[TTool, 2017a]TTool (2017a). TTool. http://ttool.telecom-paristech.fr.

[TTool, 2017b]TTool (2017b). TTool/Avatar. http://ttool.telecom-paristech.fr/avatar.html.

[TTool, 2017c]TTool (2017c). TTool/DIPLODOCUS. http://ttool.telecom-paristech.fr/diplodocus.html.

[Vanderperren et al., 2012]Vanderperren, Y., Mueller, W., He, D., Mischkalla, F., and Dehaene, W. (2012). Extending uml for electronic systems design: A code generation perspective. In *Design Technology for Heterogeneous Embedded Systems*, pages 13–39.

[Verizon, 2015]Verizon (2015). 5g specifications. http://www.5gtf.org/.

[Xi et al., 2005]Xi, C., JianHua, L., Zucheng, Z., and Yaohui, S. (2005). Modeling systemc design in uml and automatic code generation. In *ASP-DAC*, pages 932–935.

[Xilinx, 2017]Xilinx (2017). Sdx development environment. https://www.xilinx.com/products/design-tools/\\all-programmable-abstractions.html.