Models of Architecture for DSP Systems

Maxime Pelcat

Models of Architecture for DSP Systems

Maxime Pelcat

Over the last decades, the practice of representing digital signal processing applications with formal Models of Computation (MoCs) has developed. Formal MoCs are used to study application properties (liveness, schedulability, parallelism...) at a high level, often before implementation details are known. Formal MoCs also serve as an input for Design Space Exploration (DSE) that evaluates the consequences of software and hardware decisions on the final system. The development of formal MoCs is fostered by the design of increasingly complex applications requiring early estimates on a system's functional behavior.

On the architectural side of digital signal processing system development, heterogeneous systems are becoming ever more complex. Languages and models exist to formalize performance-related information of a hardware system. They most of the time represent the topology of the system in terms of interconnected components and focus on time performance. However, the body of work on what we will call Models of Architecture (MoAs) in this chapter is much more limited and less neatly delineated than the one on MoCs. This chapter proposes and argues a definition for the concept of an MoA and gives an overview of architecture models and languages that draw near the MoA concept.

1 Introduction

In computer science, system performance is often used as a synonym for real-time performance, i.e. adequate processing speed. However, most Digital Signal Processing (DSP) systems must, to fit their market, be efficient in many of their aspects and meet at the same time several efficiency constraints, including high performance, low cost, and low power consumption. These systems are referred to as high perfor-

Maxime Pelcat
Institut Pascal, Aubière, France, IETR/INSA, Rennes, France, e-mail: mpelcat@insa-rennes.fr

mance embedded systems [38] and include for instance medical image processing systems [34], wireless transceivers [32], and video compression systems [6].

The holistic optimisation of a system in its different aspects is called Design Space Exploration (DSE) [31]. Exploring the design space consists in creating a Pareto chart such as the one in Figure 1 and choosing solutions on the Pareto front, i.e. solutions that represent the best alternative in at least one dimension and respect constraints in the other dimensions. As an example, p_1 on Figure 1 can be energy consumption and p_2 can be response time. Figure 1 illustrates in 2 dimensions a problem that, in general, has many more dimensions. In order to make system-level design efficient, separation of concerns is desirable [16]. Separation of concerns refers to forcing decisions on different design concerns to be (nearly) independent. The separation of concerns between application and architecture design makes it possible to generate many points for the Pareto by varying separately application and architecture parameters and observing their effects on system efficiency.

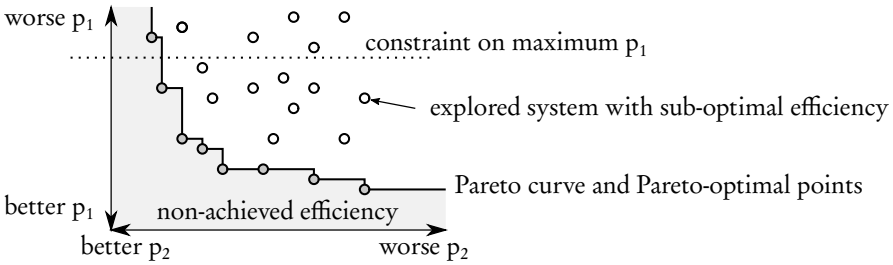

Fig. 1 The problem of Design Space Exploration (DSE) illustrated on a 2-D Pareto chart with efficiency metrics p_1 and p_2.

For example, the designer can build an application, test its efficiency on different platform architectures and, if constraints are not met by any point on the Pareto, iterate the process until reaching a satisfactory efficiency. This process is illustrated on Figure 2 and leads to Pareto points in Figure 1. Taking the hypothesis that a unique contraint is set on the maximum m_{p1} of property p_1, the first six generated systems in Figure 2 led to $p_1 > m_{p_1}$ (corresponding to points over the dotted line in Figure 1) and different values of p_2, p_3, etc. The seventh generated system has $p_1 \leq m_{p1}$ and thus respects the constraint. Further system generations can be performed to optimize p_2, p_3, etc. and generate the points under the dotted line in Figure 1. Such a design effort is feasible only if application and architecture can be played with efficiently. On the application side, this is possible using Models of Computation (MoCs) that represent the high-level aspects (e.g. parallelism, exchanged data, triggering events...) of an application while hiding its detailed implementation. Equivalently on the architectural side, Models of Architecture (MoAs) can be used to extract the fundamental elements affecting efficiency while ignoring the details of circuitry. This chapter aims at reviewing languages and tools for modeling architectures and precisely defining the scope and capabilities of MoAs.

The chapter is organised as follows. The context of MoAs is first explained in Section 2. Then, definitions of an MoA and a quasi-MoA are argued in Section 3. Sections 4 and 5 give examples of state of the art quasi-MoAs. Finally, Section 6 concludes this chapter.

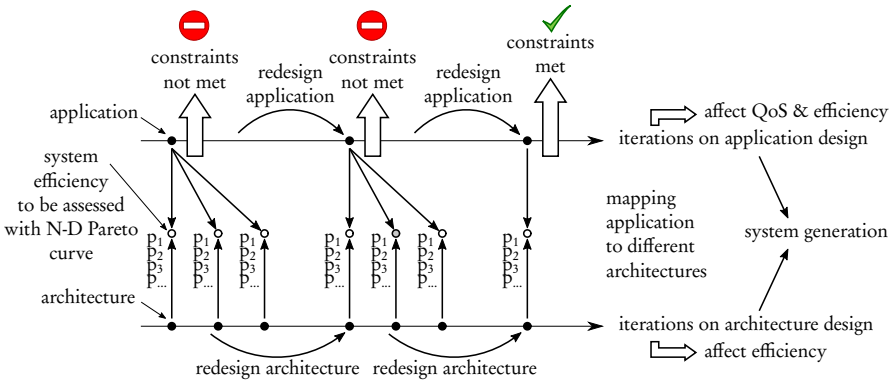

Fig. 2 Example of an iterative design process where application is refined and, for each refinement step, tested with a set of architectures to generate new points for the Pareto chart.

2 The Context of Models of Architecture

2.1 Models of Architecture in the Y-Chart Approach

The main motivation for developing Models of Architecture is for them to formalize the specification of an architecture in a Y-chart approach of system design. The Y-chart approach, introduced in [18] and detailed in [2], consists in separating in two independent models the application-related and architecture-related concerns of a system's design.

This concept is refined in Figure 3 where a set of applications is mapped to a set of architectures to obtain a set of efficiency metrics. In Figure 3, the application model is required to conform to a specified MoC and the architecture model is required to conform to a specified MoA. This approach aims at separating *What* is implemented from *How* it is implemented. In this context, the application is qualified by a *Quality of Service (QoS)* and the architecture, offering resources to this application, is characterized by a given *efficiency* when supporting the application. For the discussion not to remain abstract, next section illustrates the problem on an example.

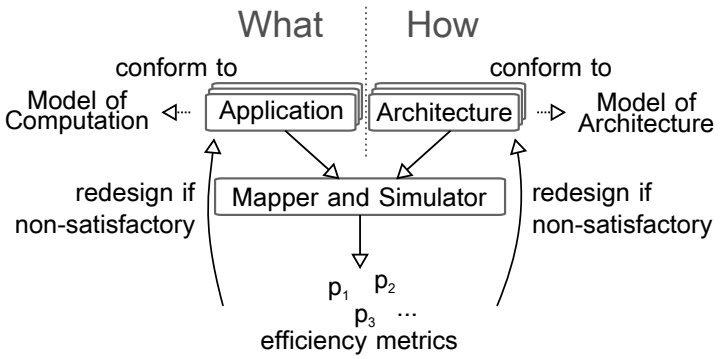

Fig. 3 MoC and MoA in the Y-chart [18].

2.2 Illustrating Iterative Design Process and Y-Chart on an Example System

QoS and efficiency metrics are multi-dimensional and can take many forms. For a signal processing application, QoS may be the Signal-to-Noise Ratio (SNR) or the Bit Error Rate (BER) of a transmission system, the compression rate of an encoding application, the detection precision of a radar, etc. In terms of architectural decisions, the obtained set of efficiency metrics is composed of some of the following Non-Functional Properties (NFPs):

- over time:
  - *latency* (also called response time) corresponds to the time duration between the arrival time of data to process and the production time of processed data,
  - *throughput* is the amount of processed data per time iterval,
  - *jitter* is the difference between maximal and minimal latency over time,
- over energy consumption:
  - *energy* corresponds to the energy consumed to process an amount of data,
  - *peak power* is the maximal instantaneous power required on alimentation to process data,
  - *temperature* is the effect of dissipated heat from processing,
- over memory:
  - *Random Access Memory (RAM) requirements* corresponds to the amount of necessary read-write memory to support processing,
  - *Read-Only Memory (ROM) requirements* is the amount of necessary read-only memory to support processing,
- over security:
  - *reliability* is $1 - p_f$ with p_f the probability of system failure over time,

  - *electromagnetic interference* corresponds to the amount of non-desired emitted radiations,

- over space:
  - *area* is the total surface of semiconductor required for a given processing,
  - *volume* corresponds to the total volume of the built system.
  - *weight* corresponds to the total weight of the built system.
- and *cost* corresponds to the monetary cost of building one system unit under the assumption of a number of produced units.

The high complexity of automating system design with a Y-chart approach comes from the extensive freedom (and imagination) of engineers in redesigning both application and architecture to fit the efficiency metrics, among this list, falling into their applicative constraints. Figure 4 is an illustrating example of this freedom on the application side. Let us consider a video compression system, borrowed from Chapter [6], to be ported on a platform. As shown in Figure 4 a), the application initially has only pipeline parallelism. Assuming that all four tasks are equivallent in complexity and that they receive and send at once a full image as a message, pipelining can be used to map the application to a multicore processor with 4 cores, with the objective to rise throughput (in frames per second) when compared to a monocore execution. However, latency will not be reduced because data will have to traverse all tasks before being output. In Figure 4 b), the image has been split into two halves and each half is processed independently. The application QoS in this second case will be lower, as the redundancy between image halves is not used for compression. The compression rate or image quality will thus be degraded. However, by accepting QoS reduction, the designer has created data parallelism that offers new opportunities for latency reduction, as processing an image half will be faster than processing a whole image.

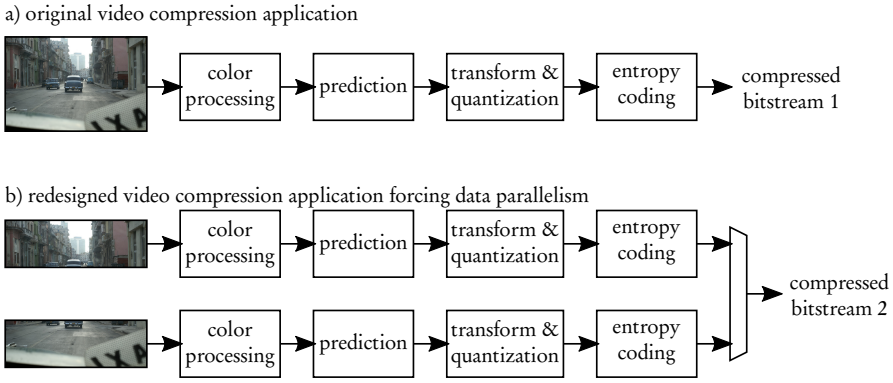

Fig. 4 Illustrating designer's freedom on the application side with a video compression example.

In terms of architecture, and depending on money and design time resources, the designer may choose to run some tasks in hardware and some in software over processors. He can also choose between different hardware interconnects to connect these architecture components. For illustrative purpose, Figure 5 shows different configurations of processors that could run the applications of Figure 4. rounded rectangles represent Processing Elements (PEs) performing computation while ovals represent Communication Nodes (CNs) performing inter-PE communication. Different combinations of processors are displayed, leveraging on high-performance out-of-order ARM Cortex-A15 cores, on high-efficiency in-order ARM Cortex-A7 cores, on the Multi-Format Codec (MFC) hardware accelerator for video encoding and decoding, or on Texas Instruments C66x Digital Signal Processing cores. Figure 5 g) corresponds to a 66AK2L06 Multicore DSP+ARM KeyStone II processor from Texas Instruments where ARM Cortex-A15 cores are combined with C66x cores connected with a Multicore Shared Memory Controller (MSMC) [36]. In these examples, all PEs of a given type communicate via shared memory with either hardware cache coherency (*Shared L2*) or software cache coherency (*MSMC*), and with each other using either the Texas Instruments *TeraNet* switch fabric or the ARM AXI Coherency Extensions (ACE) with hardware cache coherency [35].

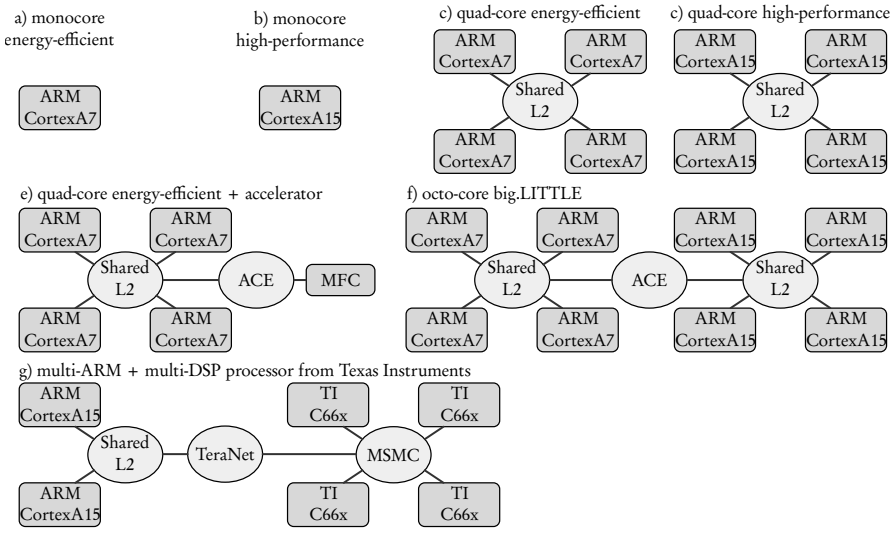

Fig. 5 Illustrating designer's freedom on the architecture side with some current ARM-based and Digital Signal Processor-based multi-core architectures.

Each architecture configuration and each *mapping* and *scheduling* of the application onto the architecture leads to different efficiencies in all the previously listed NFPs. Considering only one mapping per application-architecture couple, models from Figures 4 and 5 already define $2 \times 7 = 14$ systems. Adding mapping choices

of tasks to PEs, and considering that they all can execute any of the tasks and ignoring the order of task executions, the number of possible system efficiency points in the Pareto Chart is already roughly 19.000.000. This example shows how, by modeling application and architecture independently, a large number of potential systems is generated which makes automated multi-dimensional DSE necessary to fully explore the design space.

2.3 On the separation between application and architecture concerns

Separation between application and architectural concerns should not be confused with software (SW)/hardware (HW) separation of concerns. The software/hardware separation of concerns is often put forward in the term *HW/SW co-design*. Software and its languages are not necessarily architecture-agnostic representations of an application and may integrate architecture-oriented features if the performance is at stake. This is shown for instance by the differences existing between the C++ and CUDA languages. While C++ builds an imperative, object-oriented code for a processor with a rather centralized instruction decoding and execution, CUDA is tailored to GPGPUs with a large set of cores. As a rule of thumb, software qualifies what may be reconfigured in a system while hardware qualifies the static part of the system.

The separation between application and architecture is very different in the sense that the application may be transformed into software processes and threads, as well as into hardware Intellectual Property cores (IPs). Software and Hardware application parts may collaborate for a common applicative goal. In the context of DSP, this goal is to transform, record, detect or synthetize a signal with a given QoS. MoCs follow the objective of making an application model agnostic of the architectural choices and of the HW/SW separation. The architecture concern relates to the set of hardware and software support features that are not specific to the DSP process, but create the resources handling the application.

On the application side, many MoCs have been designed to represent the behavior of a system. The Ptolemy II project [7] has a considerable influence in promoting MoCs with precise semantics. Different families of MoCs exist such as finite state machines, process networks, Petri nets, synchronous MoCs and functional MoCs. This chapter defines MoAs as the architectural counterparts of MoCs and presents a state-of-the-art on architecture modeling for DSP systems.

2.4 Scope of this chapter

In this chapter, we focus on architecture modeling for the performance estimation of a DSP application over a complex distributed execution platform. We keep func-

tional testing of a system out of the scope of the chapter and rather discuss the early evaluation of system non-functional properties. As a consequence, virtual platforms such as QEMU [3], gem5 [4] or Open Virtual Platforms simulator (OVPsim), that have been created as functional emulators to validate software when silicon is not available, will not be discussed. MoAs work at a higher level of abstraction where functional simulation is not central.

The considered systems being dedicated to digital signal processing, the study concentrates on signal-dominated systems where control is limited and provided together with data. Such systems are called *transformational*, as opposed to *reactive* systems that can, at any time, react to non-data-carrying events by executing tasks.

Finally, the focus is put on system-level models and design rather than on detailed hardware design, already addressed by large sets of existing literature. Next section introduces the concept of an MoA, as well as an MoA example named Linear System-Level Architecture Model (LSLA).

3 The Model of Architecture Concept

The concept of MoA is evoked in 2002 in [19] where it is defined as "a formal representation of the operational semantics of networks of functional blocks describing architectures". This definition is broad, and allows the concepts of MoC and MoA to overlap. As an example, a Synchronous Dataflow (SDF) graph [24] [14] representing a system fully specialized to an application may be considered as a MoC, because it formalizes the application. It may also be considered as an MoA because it fully complies with the definition from [19]. The Definition 4 of this chapter, adapted from [30], is a new definition of an MoA that does not overlap with the concept of MoC. The LSLA model is then presented to clarify the concept by an example.

3.1 Definition of an MoA

Prior to defining MoA, the notion of application activity is introduced that ensures the separation of MoC and MoA. Figure 6 illustrates how application activity provides intermediation between application and architecture. Application activity models the computational load handled by the architecture when executing the application.

Definition 1. Application activity $\mathscr{A}$ corresponds to the amount of processing and communication necessary for accomplishing the requirements of the considered application during the considered time slot. Application activity is composed of processing and communication *tokens*, themselves composed of *quanta*.

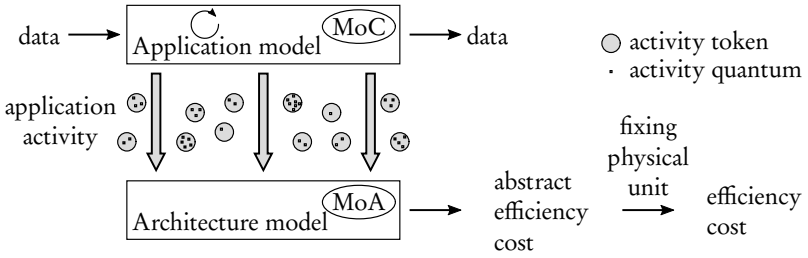

Fig. 6 Application activity as an intermediate model between application and architecture.

Definition 2. A quantum q is the smallest unit of application activity. There are two types of quanta: processing quantum q_P and communication quantum q_C.

Two distinct processing quanta are equivalent, thus represent the same amount of activity. Processing and communication quanta do not share the same unit of measurement. As an example, in a system with a unique clock and byte-addressable memory, 1 cycle of processing can be chosen as the processing quantum and 1 byte as the communication quantum.

Definition 3. A token $\tau \in T_P \cup T_C$ is a non-divisible unit of application activity, composed of a number of quanta. The function $size : T_P \cup T_C \rightarrow \mathbb{N}$ associates to each token the number of quanta composing the token. There are two types of tokens: processing tokens $\tau_P \in T_P$ and communication tokens $\tau_C \in T_C$.

The activity $\mathscr{A}$ of an application is composed of the set:

$$\mathscr{A} = \{T_P, T_C\} \tag{1}$$

where $T_P = \{\tau_P^1, \tau_P^2, \tau_P^3 ...\}$ is the set of processing tokens composing the application processing and $T_C = \{\tau_C^1, \tau_C^2, \tau_C^3 ...\}$ is the set of communication tokens composing the application communication.

An example of a processing token is a run-to-completion task with always identical computation. All tokens representing the execution of this task enclose the same number N of processing quanta (e.g. N cycles). An example of a communication token is a message in a message-passing system. The token is then composed of M communication quanta (e.g. M Bytes). Using the two levels of granularity of a token and a quantum, an MoA can reflect the cost of managing a quantum, and the additional cost of managing a token composed of several quanta.

Definition 4. A Model of Architecture (MoA) is an abstract efficiency model of a system architecture that provides a unique, reproducible cost computation, unequivocally assessing an architecture efficiency cost when supporting the *activity* of an application described with a specified MoC.

This definition makes three aspects fundamental for an MoA:

- *reproducibility*: using twice the same MoC and activity computation with a given MoA, system simulation should return the exact same efficiency cost,
- *application independence*: the MoC alone carries application information and the MoA should not comprise application-related information such as the exchanged data formats, the task representations, the input data or the considered time slot for application observation. *Application activity* is an intermediate model between a MoC and an MoA that prevents both models to intertwine. An *application activity* model reflects the computational load to be handled by architecture and should be versatile enough to support a large set of MoCs and MoAs, as demonstrated in [30].
- *abstraction*: a system efficiency cost, as returned by an MoA, is not bound to a physical unit. The physical unit is associated to an efficiency cost outside the scope of the MoA. This is necessary not to redefine the same model again and again for energy, area, weight, etc.

Definition 4 does not compel an MoA to match the internal structure of the hardware architecture, as long as the generated cost is of interest. An MoA for energy modeling can for instance be a set of algebraic equations relating application activity to the energy consumption of a platform. To keep a reasonably large scope, this chapter concentrates on graphical MoAs defined hereafter:

Definition 5. A graphical MoA is an MoA that represents an architecture with a graph $\Lambda = \langle M, L, t, p \rangle$ where M is a set of "black-box" components and $L \subseteq M \times M$ is a set of links between these components.

The graph Λ is associated with two functions t and p. The *type* function $t : M \times L \mapsto T$ associates a type $t \in T$ to each component and to each link. The type dedicates a component for a given service. The *properties* function $p : M \times L \times \Lambda \mapsto \mathcal{P}(P)$, where $\mathcal{P}$ represents powerset, gives a set of properties $p_i \in P$ to each component, link, and to the graph Λ itself. Properties are features that relate application activity to implementation efficiency.

When the concept of MoA is evoked throughout this chapter, a graphical MoA is supposed, respecting Definition 5. When a model of a system architecture is evoked that only partially compels with this definition, the term *quasi-MoA* is used, equivalent to *quasi-moa* in [30] and defined hereafter:

Definition 6. A quasi-MoA is a model respecting some of the aspects of Definition 4 of an MoA but violating at least one of the three fundamental aspects of an MoA, i.e. *reproducibility*, *application independence*, and *abstraction*.

All state-of-the-art languages and models presented in Sections 4 and 5 define quasi-MoAs. As an example of a graphical quasi-MoAs, the graphical representation used in Figure 5 shows graphs $\Lambda = \langle M, L \rangle$ with two types of components (PE

and CN), and one type of undirected link. However, no information is given on how to compute a cost when associating this representation with an application representation. As a consequence, *reproducibility* is violated. Next section illustrates the concept of MoA through the LSLA example.

3.2 Example of an MoA: the Linear System-Level Architecture Model (LSLA)

The LSLA model computes an additive reproducible cost from a minimalistic representation of an architecture [30]. As a consequence, LSLA fully complies with Definition 5 of a graphical MoA. The LSLA composing elements are illustrated in Figure 7. An LSLA model specifies two types of components: Processing Elements and Communication Nodes, and one type of link. LSLA is categorized as linear because the computed cost is a linear combination of the costs of its components.

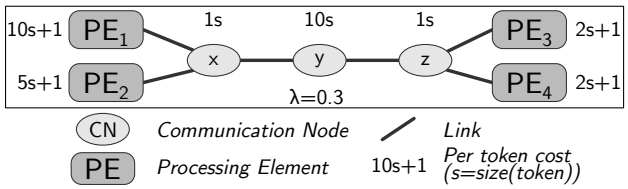

Fig. 7 LSLA MoA semantics elements.

Definition 7. The Linear System-Level Architecture Model (LSLA) is a Model of Architecture (MoA) that consists of an undirected graph $\Lambda = (P, C, L, cost, \lambda)$ where:

- P is a set of Processing Elements (PEs). A PE is an abstract processing facility with no assumption on internal parallelism, Instruction Set Architecture (ISA), or internal memory. A processing token τ_P from application activity must be mapped to a PE $p \in P$ to be executed.
- C is the set of architecture Communication Nodes (CNs). A communication token τ_C must be mapped to a CN $c \in C$ to be executed.
- $L = \{(n_i, n_j) | n_i \in C, n_j \in C \cup P\}$ is a set of undirected links connecting either two CNs or one CN and one PE. A link models the capacity of a CN to communicate tokens to/from a PE or to/from another CN.
- *cost* is a property function associating a cost to different elements in the model. The cost unit is specific to the non-functional property being modeled. It may be in *mJ* for studying energy or in mm^2 for studying area. Formally, the generic unit is denoted ν.

On the example displayed in Figure 7, PE_{1-4} represent Processing Elements (PEs) while x, y and z are Communication Nodes (CNs). As an MoA, LSLA provides reproducible cost computation when the activity $\mathscr{A}$ of an application is *mapped* onto the architecture. The cost related to the management of a token τ by a PE or a CN n is defined by:

$$\begin{array}{rcl} cost : T_P \cup T_C \times P \cup C \rightarrow & & \mathbb{R} \\ \tau, n & \mapsto & \alpha_n.size(\tau) + \beta_n, \\ \alpha_n \in \mathbb{R}, \beta_n \in \mathbb{R} & & \end{array} \tag{2}$$

where α_n is the fixed cost of a quantum when executed on n and β_n is the fixed overhead of a token when executed on n. For example, in an energy modeling use case, α_n and β_n are respectively expressed in $energy/quantum$ and $energy/token$, as the cost unit ν represents energy. A token communicated between two PEs connected with a chain of CNs $\Gamma = \{x, y, z...\}$ is reproduced $card(\Gamma)$ times and each occurrence of the token is mapped to 1 element of Γ. This procedure is illustrated in Figure 8. In figures representing LSLA architectures, the size of a token $size(\tau)$ is abbreviated into s and the affine equations near CNs and PEs (e.g. $10s+1$) represent the cost computation related to Equation 2 with $\alpha_n = 10$ and $\beta_n = 1$.

A token not communicated between two PEs, i.e. internal to one PE, does not cause any cost. The cost of the execution of application activity $\mathscr{A}$ on an LSLA graph Λ is defined as:

$$\begin{array}{rl} cost(\mathscr{A}, \Lambda) = & \sum_{\tau \in T_P} cost(\tau, map(\tau)) + \\ & \lambda \sum_{\tau \in T_C} cost(\tau, map(\tau)) \end{array} \tag{3}$$

where $map : T_P \cup T_C \rightarrow P \cup C$ is a surjective function returning the mapping of each token onto one of the architecture elements.

- $\lambda \in \mathbb{R}$ is a Lagrangian coefficient setting the Computation to Communication Cost Ratio (CCCR), i.e. the cost of a single communication quantum relative to the cost of a single processing quantum.

Similarly to the SDF MoC [24], the LSLA MoA does not specify relations to the outside world. There is no specific PEs type for communicating with non-modeled parts of the system. This is in contrast with Architecture Analysis and Design Language (AADL) `processors` and `devices` that separate I/O components from processing components (Section 4.1). The Definition 1 of activity is sufficient to support LSLA and other types of additive MoAs. Different forms of activities are likely to be necessary to define future MoAs. Activity Definition 1 is generic to several families of MoCs, as demonstrated in [30].

Figure 8 illustrates cost computation for a mapping of the video compression application shown in Figure 4 b), described with the SDF MoC onto the big.LITTLE architecture of Figure 5 f), described with LSLA. The number of tokens, quanta and the cost parameters are not representative of a real execution but set for illustrative purpose. The natural scope for the cost computation of a couple (SDF, LSLA), provided that the SDF graph is consistent, is one SDF graph iteration [30].

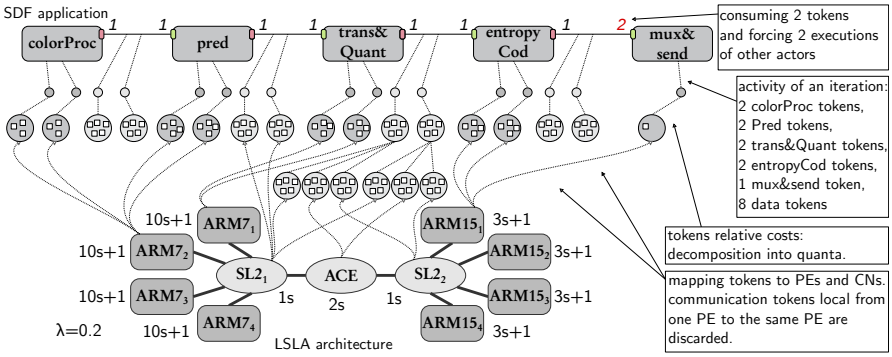

Fig. 8 Computing cost of executing an SDF graph on an LSLA architecture. The cost for 1 iteration is (looking first at processing tokens then at communication tokens from left to right) $31+31+41+41+41+41+13+13+4+0.2\times(5+5+5+10+5+5+10+5)$ = 266 ν (Equation 3).

The SDF application graph has 5 actors *colorProc*, *pred*, *trans&Quant*, *entropyCod*, and *mux&Send* and the 4 first actors will execute twice to produce the 2 image halves required by *mux&Send*. The LSLA architecture model has 8 PEs $ARMj_k$ with $j \in \{7,15\}$ and $k \in \{1,2,3,4\}$, and 3 CNs $SL2_1$, ACE and $SL2_2$. Each actor execution during the studied graph iteration is transformed into one processing token. Each dataflow token transmitted during one iteration is transformed into one communication token. A token is embedding several quanta (white squares), allowing a designer to describe heterogeneous tokens to represent executions and messages of different weight.

In Figure 8, each execution of actors *colorProc* is associated with a cost of 3 quanta and each execution of other actors is associated to a cost of 4 quanta except *mux&Send* requiring 1 quantum. Communication tokens (representing one half image transfer) are given 5 quanta each. These costs are arbitrary here but should represent the relative computational load of the task/communication.

Each processing token is mapped to one PE. Communication tokens are "routed" to the CNs connecting their producer and consumer PEs. For instance, the fifth and sixth communication tokens in Figure 8 are generating 3 tokens each mapped to $SL2_1$, ACE and $SL2_2$ because the data is carried from $ARM7_1$ to $ARM15_1$. It is the responsibility of the mapping process to verify that a link $l \in L$ exists between the elements that constitute a communication route. The resulting cost, computed from Equations 2 and 3, is 266ν. This cost is reproducible and abstract, making LSLA an MoA.

LSLA is one example of an architecture model but many such models exist in literature. Next sections study different languages and models from literature and explain the quasi-MoAs they define.

4 Architecture Design Languages and their Architecture Models

This section studies the architecture models provided by three standard Architecture Design Languages (ADLs) targeting architecture modeling at system-level: AADL, MCA SHIM, and UML MARTE.

While AADL adopts an abstraction/refinement approach where components are first roughly modeled, then refined to lower levels of abstraction, UML MARTE is closer to a Y-Chart approach where the application and the architecture are kept separated and application is mapped to architecture.

For its part, MCA SHIM describes an architecture with "black box" processors and communications and puts focus on inter-PE communication simulation. All these languages have in common the implicit definition of a quasi-MoA (Definition 6). Indeed, while they define parts of graphical MoAs, none of them respect the 3 rules of MoA Definition 4.

4.1 The AADL Quasi-MoA

Architecture Analysis and Design Language (AADL) [9] is a standard language released by SAE International, an organization issuing standards for the aerospace and automotive sectors. The AADL standard is referenced as AS5506 [33] and the last released version is 2.2. Some of the most active tools supporting AADL are Ocarina[1] [21] and OSATE[2] [9].

4.1.1 The Features of the AADL Quasi-MoA

AADL provides semantics to describe a software application, a hardware platform, and their combination to form a system. AADL can be represented graphically, serialized in XML or described in a textual language [10]. The term *architecture* in AADL is used in its broadest sense, i.e. a whole made up of clearly separated elements. A design is constructed by successive refinements, filling "black boxes" within the AADL context. Figure 9 shows two refinement steps for a video compression system in a camera. Blocks of processing are split based on the application decomposition of Figure 4 a). First, the system is abstracted with external data entering a video compression `abstract component`. Then, 4 software `processes` are defined for the processing. Finally, processes are transformed into 4 `threads`, mapped onto 2 processes. The platform is defined with 2 cores and a bus and application threads are allocated onto platform components. The allocation of threads to processors is not displayed. Sensor data is assigned a rate of 30 Hz, correspond-

[1] https://github.com/OpenAADL/ocarina

[2] https://github.com/osate

ing to 30 frames per second. Next sections detail the semantics of the displayed components.

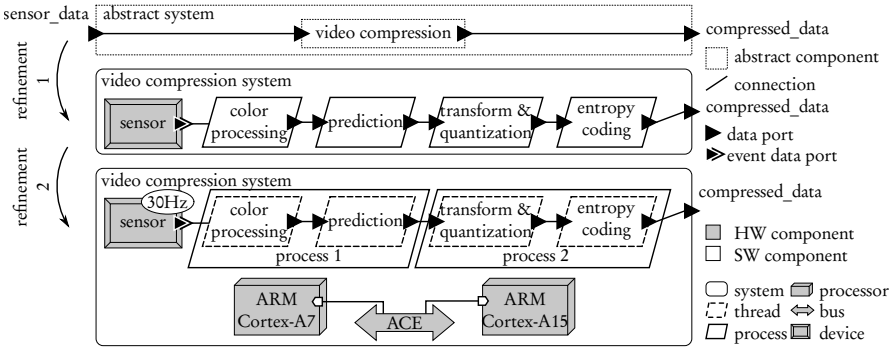

Fig. 9 The AADL successive refinement system design approach.

Software, hardware and systems are described in AADL by a composition of `components`. In this chapter, we focus on the hardware platform modeling capabilities of AADL, composing an implicit graphical quasi-MoA. Partly respecting Definition 5, AADL represents platform with a graph $\Lambda = \langle M, L, t, p \rangle$ where M is a set of components, L is a set of links, t associates a type to each component and link and p gives a set of properties to each component and link. As displayed in Figure 10, AADL defines 6 types of platform components with specific graphical representations. The AADL component type set is such that $t(c \in M) \in \{$`system`, `processor`, `device`, `bus`, `memory`, `abstract`$\}$. There is one type of link $t(l \in L) \in \{$`connection`$\}$. A `connection` can be set between any two components among software, hardware or system. Contrary to the Y-chart approach, AADL does not separate application from architecture but makes them coexist in a single model.

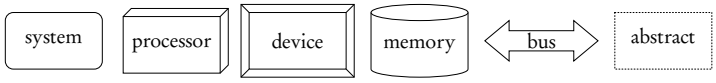

Fig. 10 The basic components for describing a hardware architecture in AADL.

AADL is an extensible language but defines some *standard component properties*. These properties participate to the definition of the quasi-MoA determined by the language and make an AADL model portable to several tools. The AADL standard set of properties targets only the time behavior of components and differs for each kind of component. AADL tools are intended to compute NFP costs such as the total minimum and maximum execution latency of an application, as well as the jitter. An AADL representation can also be used to extract an estimated bus bandwidth or a subsystem latency [20].

Processors are sequential execution facilities that must support thread scheduling, with a protocol fixed as a property. AADL platform components are not merely hardware models but rather model the combination of hardware and low-level software that provides services to the application. In that sense, the architecture model they compose is conform to MoA Definition 4. However, what is mapped on the platform is *software* rather than an *application*. As a consequence, the separation of concerns between application and architecture is not supported (Section 2.3). For instance, converting the service offered by a software thread to a hardware IP necessitates to deeply redesign the model. A `processor` can specify a *Clock_Period*, a *Thread_Swap_Execution_Time* and an *Assign_Time*, quantifying the time to access memory on the processor. Time properties of a processor can thus be precisely set.

A `bus` can specify a fixed *Transmission_Time* interval representing best- and worst-case times for transmitting `data`, as well as a *PerByte Transmission_Time* interval representing throughput. The time model for a message is thus an affine model w.r.t. message size. Three models for transfer cost computation are displayed in Figure 11: linear, affine, and stair. Most models discussed in the next sections use one of these 3 models. The interpretation of AADL time properties is precisely defined in [9] Appendix A, making AADL time computation reproducible.

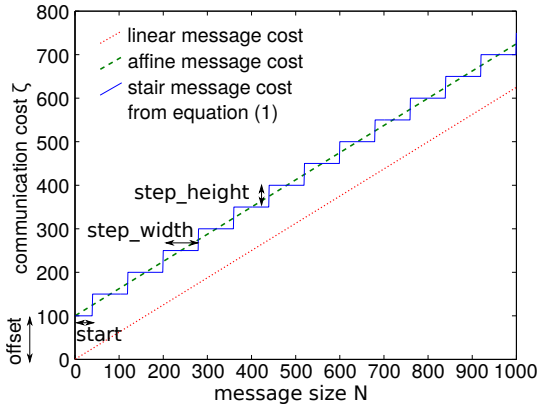

Fig. 11 Examples of different data transfer cost computation functions (in arbitrary units): a linear function (with 1 parameter), an affine function (with 2 parameters) and a step function (with 4 parameters).

A `memory` can be associated to a *Read_Time*, a *Write_Time*, a *Word_Count* and a *Word_Size* to characterize its occupancy rate. A `device` can be associated to a *Period*, and a *Compute_Execution_Time* to study sensors' and actuators' latency and throughput. Platform components are defined to support a software application. The next section studies application and platform interactions in AADL.

4.1.2 Combining Application and Architecture in AADL

AADL aims at analyzing the time performance of a system's architecture, manually exploring the mapping (called `binding` in AADL) of software onto hardware elements. AADL quasi-MoA is influenced by the supported software model. AADL is adapted to the currently dominating software representation of Operating Systems (OS), i.e. the process and thread representation [9]. An application is decomposed into process and thread components, that are purely software concepts. A process defines an address space and a thread comes with scheduling policies and shares the address space of its owner process. A process is not executable by itself; it must contain a least one thread to execute. AADL `Threads` are sequential, preemptive entities [9] and requires scheduling by a `processor`. `Threads` may specify a *Dispatch_Protocol* or a *Period* property to model a periodic behavior or an event-triggered callback or routine.

A values or interval of *Compute_Execution_Time* can be associated to a `thread`. However, in real world, execution time for a thread firing depends on both the code to execute and the platform speed. *Compute_Execution_Time* is not related to the binding of the thread to a `processor` but a *Scaling_Factor* property can be set on the `processor` to specify its relative speed with regards to a reference processor for which `thread` timings have been set. This property is precise when all threads on a processor undergo the same *Scaling_Factor*, but this is not the case in general. For instance, if a thread compiled for the ARMv7 instruction set is first executed on an ARM Cortex-A7 and then on an ARM Cortex-A15 processor, the observed speedup depends much on the executed task. Speedups between $1.3\times$ and $4.9\times$ are reported in this context in [30].

AADL provides constructs for data message passing through *port* features and data memory-mapped communication through *require data access* features. These communications are bound to `busses` to evaluate their timings.

A *flow* is neither a completely software nor a completely hardware construct. It specifies an end-to-end flow of data between sensors and actuators for steady state and transient timing analysis. A *flow* has timing properties such as *Expected_Latency* and *Expected_Throughput* that can be verified through simulation.

4.1.3 Conclusions on the AADL Quasi-MoA

AADL specifies a graphical quasi-MoA, as it does define a graph of platform components. AADL violates the *abstraction* rule because cost properties are explicitly time and memory. It respects the *reproducibility* rule because details of timing simulations are precisely defined in the documentation. Finally, it violates the *application independence* rule because AADL does not conform to the Y-chart approach and does not separate application and architecture concerns.

AADL is a formalization of current best industrial practices in embedded system design. It provides formalization and tools to progressively refine a system from an abstract view to a software and hardware precise composition. AADL targets

all kinds of systems, including transformational DSP systems managing data flows but also reactive system, reacting to sporadic *events*. The thread MoC adopted by AADL is extremely versatile to reactive and transformational systems but has shown its limits for building deterministic systems [23] [37]. By contrast, the quasi-MoAs presented in Section 5 are mostly dedicated to transformational systems. They are thus all used in conjunction with process network MoCs that help building reliable DSP systems. The next section studies another state-of-the-art language: MCA SHIM.

4.2 The MCA SHIM Quasi-MoA

The Software/Hardware Interface for Multicore/Manycore (SHIM) [12] is a hardware description language that aims at providing platform information to multicore software tools, e.g. compilers or runtime systems. SHIM is a standard developed by the Multicore Association (MCA). The most recent released version of SHIM is 1.0 (2015) [27]. SHIM is a more focused language than AADL, modeling the platform properties that influence software performance on multicore processors.

SHIM components provide timing estimates of a multicore software. Contrary to AADL that mostly models hard real-time systems, SHIM primarily targets best-effort multicore processing. Timing properties are expressed in clock cycles, suggesting a fully synchronous system. SHIM is built as a set of UML classes and the considered NFPs in SHIM are time and memory. Timing performances in SHIM are set by a `shim::Performance` class that characterizes three types of software activity: instruction executions for instructions expressed in the LLVM instruction set, memory accesses, and inter-core communications. LLVM [22] is used as a portable assembly code, capable of decomposing a software task into instructions that are portable to different ISAs.

SHIM does not propose a chart representation of its components. However, SHIM defines a quasi-MoA partially respecting Definition 5. A `shim::System-Configuration` object corresponds to a graph $\Lambda = \langle M, L, t, p \rangle$ where M is the set of components, L is the set of links, t associates a type to each component and link and p gives a set of properties to each component and link. A SHIM architecture description is decomposed into three main sets of elements: `Components`, `Address Spaces` and `Communications`. We group and rename the components (referred to as "objects" in the standard) to makes them easier to compare to other approaches. SHIM defines 2 types of platform components. The component types $t(c \in M)$ are chosen among:

- `processor` (*shim::MasterComponent*), representing a core executing software. It internally integrates a number of `cache` memories (*shim::Cache*) and is capable of specific data access types to memory (*shim::AccessType*). A `processor` can also be used to represent a Direct Memory Access (DMA),
- `memory` (*shim::SlaveComponent*) is bound to an address space (*shim::Address-Space*).

Links $t(l \in L)$ are used to set performance costs. They are chosen among:

- `communication` between two `processors`. It has 3 subtypes:
  - `fifo` (*shim::FIFOCommunication*) referring to message passing with buffering,
  - `sharedRegister` (*shim::SharedRegisterCommunication*) referring to a semaphore-protected register,
  - `event` (*shim::EventCommunication* for polling or *shim::InterruptCommunication* for interrupts) referring to inter-core synchronization without data transfer.
- `memoryAccess` between a `processor` and a `memory` (modeled as a couple *shim::MasterSlaveBinding*, *shim::Accessor*) sets timings to each type of data read/write accesses to the memory.
- `sharedMemory` between two `processors` (modeled as a triple *shim::SharedMemoryCommunication*, *shim::MasterSlaveBinding*, and *shim::Accessor*) sets timing performance to exchanging data over a shared memory,
- `InstructionExecution` (modeled as a *shim::Instruction*) between a processor and itself sets performance on instruction execution.

Links are thus carrying all the performance properties in this model. Application activity on a link l is associated to a `shim::Performance` property, decomposed into *latency* and *pitch*. *Latency* corresponds to a duration in cycles while *pitch* is the inverse (in cycles) of the throughput (in cycles^{-1}) at which a SHIM object can be managed. A latency of 4 and a pitch of 3 on a communication link, for instance, mean that the first data will take 4 cycles to pass through a link and then 1 data will be sent per 3 cycles. This choice of time representation is characteristic of the SHIM objective to model the average behavior of a system while AADL targets real-time systems. Instead of specifying time intervals $[min..max]$ like AADL, SHIM defines triplets $[min, mode, max]$ where mode is the statistical mode. As a consequence, a richer communication and execution time model can be set in SHIM. However, no information is given on how to use these performance properties present in the model. In the case of a communication over a shared memory for instance, the decision on whether to use the performance of this link or to use the performance of the shared memory data accesses, also possible to model, is left to the SHIM supporting tool.

4.2.1 Conclusions on MCA SHIM Quasi-MoA

MCA SHIM specifies a graphical quasi-MoA, as it defines a graph of platform components. SHIM violates the *abstraction* rule because cost properties are limited to time. It also violates the *reproducibility* rule because details of timing simulations are left to the interpretation of the SHIM supporting tools. Finally, it violates the *application independence* rule because SHIM supports only software, decomposed into LLVM instructions.

The modeling choices of SHIM are tailored to the precise needs of multicore tooling interoperability. The two types of tools considered as targets for the SHIM standard are Real-Time Operating Systems (RTOSs) and auto-parallelizing compilers for multicore processors. The very different objectives of SHIM and AADL have led to different quasi-MoAs. The set of components is more limited in SHIM and communication with the outside world is not specified. The communication modes between processors are also more abstract and associated to more sophisticated timing properties. The software activity in SHIM is concrete software, modeled as a set of instructions and data accesses while AADL does not go as low in terms of modeling granularity. To complement the study on a third language, the next section studies the different quasi-MoAs defined by the Unified Modeling Language (UML) Modeling And Analysis Of Real-Time Embedded Systems (MARTE) language.

4.3 The UML MARTE Quasi-MoAs

The UML Profile for Modeling And Analysis Of Real-Time Embedded Systems (MARTE) is standardized by the Object Management Group (OMG) group. The last version is 1.1 and was released in 2011 [28]. Among the ADLs presented in this chapter, UML MARTE is the most complex one. It defines hundreds of UML classes and has been shown to support most AADL constructs [8]. MARTE is designed to coordinate the work of different engineers within a team to build a complex real-time embedded system. Several persons, expert in UML MARTE, should be able to collaborate in building the system model, annotate and analyze it, and then build an execution platform from its model. Like AADL, UML MARTE is focused on hard real-time application and architecture modeling. MARTE is divided into four *packages*, themselves divided into *clauses*. 3 of these clauses define 4 different quasi-MoAs. These quasi-MoAs are named $QMoA^{i}_{MARTE} \mid i \in \{1,2,3,4\}$ in this chapter and are located in the structure of UML MARTE clauses illustrated by the following list:

- The *MARTE Foundations* package includes:
  - the *Core Elements* clause that gathers constructs for inheritance and composition of abstract objects, as well as their invocation and communication.
  - the *Non-Functional Property (NFP)* clause that describes ways to specify non-functional constraints or values (Section 2.2), with a concrete type.
  - the *Time* clause, specific to the time NFP.
  - the *Generic Resource Modeling (GRM)* clause that offers constructs to model, at a high level of abstraction, both software and hardware elements. It defines a generic component named `Resource`, with `clocks` and `non-functional properties`. `Resource` is the basic element of UML MARTE models of architecture and application. The quasi-MoA $QMoA^{1}_{MARTE}$ is defined by GRM and based on `Resources`. It will be presented in Section 4.3.1.

  - the *Allocation Modeling* clause that relates higher-level `Resources` to lower-level `Resources`. For instance, it is used to allocate `Schedulable-Resources` (e.g. threads) to `ComputingResources` (e.g. cores).

- The *MARTE Design Model* package includes:
  - the *Generic Component Model (GCM)* clause that defines structured components, connectors and interaction ports to connect core elements.
  - the *Software Resource Modeling (SRM)* clause that details software resources.
  - the *Hardware Resource Modeling (HRM)* clause that details hardware resources and defines $QMoA^2_{MARTE}$ and $QMoA^3_{MARTE}$ (Section 4.3.2).
  - the *High-Level Application Modeling (HLAM)* clause that models real-time services in an OS.

- The *MARTE Analysis Model* package includes:
  - the *Generic Quantitative Analysis Modeling (GQAM)* clause that specifies methods to observe system performance during a time interval. It defines $QMoA^4_{MARTE}$.
  - the *Schedulability Analysis Modeling (UML MARTE) (SAM)* clause that refers to thread and process schedulability analysis. It builds over GQAM and adds scheduling-related properties to $QMoA^4_{MARTE}$.
  - the *Performance Analysis Modeling (PAM)* clause that performs probabilistic or deterministic time performance analysis. It also builds over GQAM.

- *MARTE Annexes* include *Repetitive Structure Modeling (RSM)* to compactly represent component networks, and the *Clock Constraint Specification Language (CCSL)* to relate clocks.

The link between application time and platform time in UML MARTE is established through clock and event relationships expressed in the CCSL language [25]. Time may represent a physical time or a logical time (i.e. a continuous repetition of events). Clocks can have causal relations (an event of clock A causes an event of clock B) or a temporal relations with type *precedence*, *coincidence*, and *exclusion*. Such a precise representation of time makes UML MARTE capable of modeling both asynchronous and synchronous distributed systems [26]. UML MARTE is capable, for instance, of modeling any kind of processor with multiple cores and independent frequency scaling on each core.

The UML MARTE `resource` composition mechanisms give the designer more freedom than AADL by dividing his system into more than 2 layers. For instance, execution platform `resources` can be allocated to operating system `resources`, themselves allocated to application `resources` while AADL offers only a hardware/software separation. Multiple allocations to a single `resource` are either time multiplexed (*timeScheduling*) or distributed in space (*spatialDistribution*). Next sections explain the 4 quasi-MoAs defined by UML MARTE.

4.3.1 The UML MARTE Quasi-MoAs 1 and 4

The UML MARTE GRM clause specifies the $QMoA^{1}_{MARTE}$ quasi-MoA. It corresponds to a graph $\Lambda = \langle M, L, t, p \rangle$ where M is a set of `Resources`, L is a set of UML `Connectors` between these resources, t associates types to `Resources` and p gives sets of properties to `Resources`.

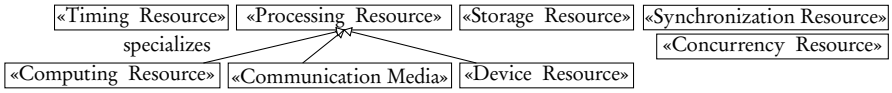

Fig. 12 Elements of the quasi-MoA define in UML MARTE Generic Resource Modeling (GRM).

7 types of resources are defined in GRM. Some inconsistencies between resource relations make the standard ambiguous on resource types. As an example, `CommunicationMedia` specializes `CommunicationResource` on standard p.96 [28] while `CommunicationMedia` specializes `ProcessingResource` on standard p.99. `SynchResource` disappears after definition and is possibly equivalent to the later `SwSynchronizationResource`. Considering the most detailed descriptions as reference, types of resources (illustrated in Figure 12) are:

- a `Processing Resource`, associated to an abstract *speed Factor* property that can help the designer compare different `Processing Resources`. It has 3 subtypes: `Computing Resource` models a real or virtual PE storing and executing program code. It has no property. `Device Resource` communicates with the system environment, equivalently to an AADL device. It also has no property. `Communication Media` can represent a bus or a higher-level protocol over an interconnect. It has several properties: a mode among simplex, half-duplex, or full-duplex specifies whether the media is directed or not and the time multiplexing method for data. `Communication Media` transfers one data of *elementSize* bits per clock cycle. A *packet time* represents the time to transfer a set of elements. A *block time* represents the time before the media can transfer other packets. A *data rate* is also specified.
- a `Timing Resource` representing a clock or a timer, fixing a clock rate.
- a `Storage Resource` representing memory, associated with a unit size and number of units. Memory read and write occur in 1 clock cycle.
- a `Concurrency Resource` representing several concurrent flows of execution. It is a generalization of `SchedulableResources` that model logical concurrency in threads and processes.

The communication time model of $QMoA^{1}_{MARTE}$, set by the `Communication Media`, is the affine model illustrated in Figure 11. Precise time properties are set but the way to correctly compute a timing at system-level from the set of resource timings is not explicitly elucidated.

$QMoA^{1}_{MARTE}$ can be used for more than just time modeling. `ResourceUsage` is a way to associate physical properties to the usage of a resource. When events

occur, amounts of physical resources can be specified as "consumed". A resource consumption amount can be associated to the following types of NFPs values: energy in Joules, message size in bits, allocated memory in bytes, used memory in bytes (representing temporary allocation), and power peak in Watts.

The Generic Quantitative Analysis Modeling (GQAM) package defines another quasi-MoA ($QMoA^4_{MARTE}$) for performing the following set of analysis: counting the repetitions of an event, determining the probability of an execution, determining CPU requirements, determining execution latency, and determining throughput (time interval between two occurrences). New resources named `GaExecHost` (ExecutionHost) and `GaCommHost` (CommunicationHost) are added to the ones of $QMoA^1_{MARTE}$ and specialize the `ProcessingResource` for time performance and schedulability analysis, as well as for the analysis of other NFPs. $QMoA^4_{MARTE}$ is thus close to $QMoA^1_{MARTE}$ in terms of resource semantics but additional properties complement the quasi-MoA. In terms of MoAs, $QMoA^1_{MARTE}$ and $QMoA^4_{MARTE}$ have the same properties and none of them clearly states how to use their properties.

4.3.2 The UML MARTE Quasi-MoAs 2 and 3

The UML MARTE Hardware Resource Modeling (HRM) defines two other, more complex quasi-MoAs than the previously presented ones: $QMoA^2_{MARTE}$ (logical view) and $QMoA^3_{MARTE}$ (physical view).

An introduction of the related software model is necessary before presenting hardware components because the HRM is very linked to the SRM software representation. In terms of software, the UML MARTE standard constantly refers to threads as the basic instance, modeled with a `swSchedulableResource`. The `swSchedulableResources` are thus considered to be managed by an RTOS and, like AADL, UML MARTE builds on industrial best practices of using preemptive threads to model concurrent applications. In order to communicate, a `swSchedulableResource` references specifically defined software communication and synchronization resources.

The `HW_Logical` subclause of HRM refers to 5 subpackages: `HW_Computing`, `HW_Communication`, `HW_Storage`, `HW_Device`, and `HW_Timing`. It composes a complex quasi-MoA referred to as $QMoA^2_{MARTE}$ in this chapter. For brevity and clarity, we will not enter the details of this quasi-MoA but give some information on its semantics.

The UML MARTE $QMoA^2_{MARTE}$ quasi-MoA is, like AADL, based on a HW/SW separation of concerns rather than on an application/architecture separation. In terms of hardware, UML MARTE tends to match very finely the real characteristics of the physical components. UML MARTE HRM is thus torn between the desire to match current hardware best practices and the necessity to abstract away system specificities. A $QMoA^2_{MARTE}$ processing element for instance can be a processor, with an explicit Instruction Set Architecture (ISA), caches, and a Memory Management Unit (MMU), or it can be a Programmable Logic Device (PLD). In the description of a PLD, properties go down to the number of available Lookup Tables (LUTs) on the

PLD. However, modern PLDs such as Field-Programmable Gate Arrays (FPGAs) are far too heterogeneous to be characterized by a number of LUTs. Moreover, each FPGA has its own characteristics and in the space domain, for instance, FPGAs are not based on a RAM configuration memory, as fixed in the MARTE standard, but rather on a FLASH configuration memory. These details show the interest of abstracting an MoA in order to be resilient to the fast evolution of hardware architectures.

`HW_Physical` composes the $QMoA^3_{MARTE}$ quasi-MoA and covers coarser-grain resources than $QMoA^2_{MARTE}$, at the level of a printed circuit board. Properties of resources include shape, size, position, power consumption, heat dissipation, etc.

Interpreting the technological properties of HRM quasi-MoAs $QMoA^2_{MARTE}$ and $QMoA^3_{MARTE}$ is supposed to be done based on designer's experience because the UML MARTE properties mirror the terms used for hardware design. This is however not sufficient to ensure the *reproducibility* of a cost computation.

4.3.3 Conclusions on UML MARTE Quasi-MoAs

When considering as a whole the 4 UML MARTE quasi-MoAs, the standard does not specify how the hundreds of NFP standard resource parameters are to be used during simulation or verification. The use of these parameters is supposed to be transparent, as the defined resources and parameters match current best practices. However, best practices evolve over time and specifying precisely cost computation mechanisms is the only way to ensure tool interoperability in the lon run. UML MARTE quasi-MoAs do not respect the *abstraction* rule of MoAs because, while cost properties target multiple NFPs, each is considered independently without capitalizing on similar behaviors of different NFPs. Finally, $QMoA^1_{MARTE}$ and $QMoA^4_{MARTE}$ respect the *application independence* rule, and even extend it to the construction of more than 2 layers, while $QMoA^2_{MARTE}$ and $QMoA^3_{MARTE}$ rather propose a HW/SW decomposition closer to AADL.

4.4 Conclusions on ADL Languages

AADL and UML MARTE are both complete languages for system-level design that offer rich constructs to model a system. MCA SHIM is a domain-specific language targeted to a more precise purpose. While the 3 languages strongly differ, they all specify quasi-MoAs with the objective of modeling the time behavior of a system, as well as other non-functional properties. None of these 3 languages fully respects the three rules of MoA's Definition 4. In particular, none of them abstracts the studied NFPs to make generic the computation of a model's cost from the cost of its constituents. Abstraction is however an important feature of MoAs to avoid redesigning redundant simulation mechanisms.

To complement this study on MoAs, the next section covers four formal quasi-MoAs from literature.

5 Formal Quasi-MoAs

In this Section, we put the focus on graphical quasi-MoAs that aim at providing system efficiency evaluations when combined with a model of a DSP application. The models and their contribution are presented chronologically.

5.1 The AAA Methodology Quasi-MoA

In 2003, an architecture model is defined for the *Adéquation Algorithm Architecture (AAA)* Y-chart methodology, implemented in the SynDEx tool [13]. The AAA architecture model is tailored to the needs of an application model that splits processing into tasks called *operations* arranged in a Directed Acyclic Graph (DAG) representing data dependencies between them.

The AAA architecture model is a graphical quasi-MoA $\Lambda = \langle M, L, t, p \rangle$, where M is a set of components, L is a set of undirected edges connecting these components, and t and p respectively give a type and a property to components. As illustrated in Figure 13, there are three types $t \in T$ of components, each considered internally as a Finite State Machine (FSM) performing sequentially application management services : *memory*, *sequencer*, and *bus/multiplexer/demultiplexer (B/M/D)*. For their part, edges only model the capacity of components to exchange data.

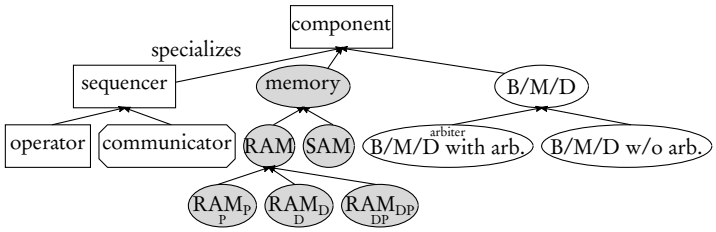

Fig. 13 Typology of the basic components in the AAA architecture model [13]. Leaf components are instantiable.

In this model, a *memory* is a Sequential Access Memory (`SAM`) or a Random Access Memory (`RAM`). A `SAM` models a First In, First Out data queue (FIFO) for message passing between components. A `SAM` can be point-to-point or multipoint and support or not broadcasting. A `SAM` with broadcasting only pops a data when all readers have read the data. A `RAM` may store only data (RAM_D), only programs

(RAM_P) or both (RAM_{DP}). When several sequencers can write to a memory, it has an implicit arbiter managing writing conflicts.

A *sequencer* is of type `operator` or `communicator`. An `operator` is a PE sequentially executing *operations* stored in a RAM_P or RAM_{DP}. An operation reads and writes data from/to a RAM_D or RAM_{DP} connected to the `operator`. A `communicator` models a DMA with a single channel that executes communications, i.e. operations that transfer data from a memory M_1 to a memory M_2. For the transfer to be possible, the communicator must be connected to M_1 and M_2.

A *B/M/D* models a bus together with its multiplexer and demultiplexer that implement time division multiplexing of data. As a consequence, a B/M/D represents a sequential schedule of transfered data. A B/M/D may require an arbiter, solving write conflicts between multiple sources. In the AAA model, the arbiter has a maximum bandwidth *BPMax* that is shared between writers and readers.

Figure 14 shows an example, inspired by [13], of a model conforming the AAA quasi-MoA. It models the 66AK2L06 processor [36] from Texas Instruments illustrated in Figure 5 g). `Operators` must delegate communication to `communicators` that access their data memory. The architecture has hardware cache coherency on ARM side (L2CC for L2 Cache Control) and software cache coherency on c66x side (SL2C for Software L2 Coherency). The communication between ARML2 and MSMC memories is difficult to model with AAA FSM components because it is performed by a Network-on-Chip (NoC) with complex topology and a set of DMAs so it has been represented as a network of B/M/Ds and communicators in Figure 14.

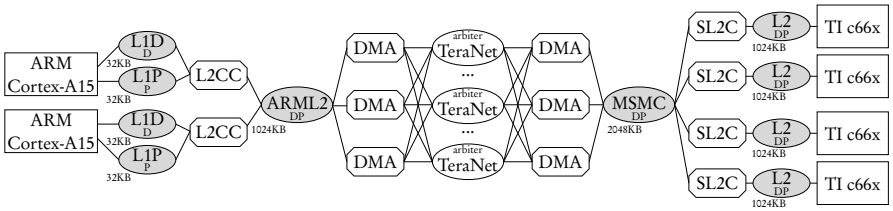

Fig. 14 Example of an architecture description with the AAA quasi-MoA.

Properties p on components and edges define the quasi-MoA. An `operator` Op has an associated function δ_{Op} setting a Worst Case Execution Time (WCET) duration to each operation $\delta_{Op}(o) \in \mathbb{R}_{\geq 0}$ where O is the set of all operations in the application. This property results from the primary objective of the AAA architecture model being the computation of an application WCET. Each edge of the graph has a maximum *bandwidth B* in bits/s. The aim of the AAA quasi-MoA is to feed a multicore scheduling process where application operations are mapped to `operators` and data dependencies are mapped to routes between `operators`, made of `communicators` and `busses`. Each `operator` and `communicator` being an FSM, the execution of operations and communications on a given sequencer is totally ordered. The application graph being a DAG, the critical path

of the application is computed and represents the latency of one execution, i.e. the time distance beween the beginning of the first operation and the end of the last operation. The computation of the latency from AAA application model and quasi-MoA in [13] is implicit. The behavior of the arbiter is not specified in the model so actual communication times are subject to interpretations, especially regarding the time quantum for the update of bandwidth utilization.

The AAA syntax-free quasi-MoA is mimicking the temporal behavior of a processing hardware in order to derive WCET information on a system. Many hardware features can be modeled, such as DMAs; shared memories and hardware FIFO queues. Each element in the model is sequential, making a coarse-grain model of an internally parallel component impossible. There is no cost abstraction but the separation between architecture model and application model is respected. The model is specific to dataflow application latency computation, with some extra features dedicated to memory requirement computation. Some performance figures are subject to interpretation and latency computation for a couple application/architecture is not specified.

The AAA model contribution is to build a system-level architecture model that clearly separates architecture concerns from algorithm concerns. Next section discusses a second quasi-MoA, named CHARMED.

5.2 The CHARMED Quasi-MoA

In 2004, the CHARMED co-synthesis framework [17] is proposed that aims at optimizing multiple system parameters represented in Pareto fronts. Such a multi-parameter optimization is essential for DSE activities, as detailed in [31].

In the CHARMED quasi-MoA $\Lambda = \langle M, L, t, p \rangle$, M is a set of PEs, L is a set of Communication Resources (CR) connecting these components, and t and p respectively give a type and a property to PEs and CRs. There is only one type of component so in this model, $t = PE$. Like in the AAA architecture model, PEs are abstract and may represent programmable microprocessors as well as hardware IPs. The PE vector of properties p is such that $p(PE \in M) = [\alpha, \kappa, \mu_d, \mu_i, \rho_{idle}]^T$ where α denotes the area of the PE, κ denotes the price of the PE, μ_d denotes the size of its data memory, μi denotes the instruction memory size and ρ_{idle} denotes the idle power consumption of the PE. Each CR edge also has a property vector: $p(CR \in L) = [\rho, \rho_{idle}, \theta]^T$ where ρ denotes the average power consumption per each unit of data to be transferred, ρ_{idle} denotes idle power consumption and θ denotes the worst case transmission rate or speed per each unit of data.

This model is close to the concept of MoA as stated by Definition 4. However, instead of abstracting the computed cost, it defines many costs altogether in a vector. This approach limits the scope of the approach and CHARMED metrics do not cover the whole spectrum on NFPs shown in Section 2.2. The CHARMED architecture model is combined with a DAG task graph of a stream processing application in order to compute costs for different system solutions. A task in the application

graph is characterized by its required instruction memory μ, its Worst Case Execution Time $WCET$ and its average power consumption $\wp_{avg}$ while a DAG edge is associated with a data size δ. The cost for a system x has 6 dimensions: the area $\alpha(x)$, the price $\kappa(x)$, the number of used inter-processor routes $l_n(x)$, the memory requirements $\mu(x)$, the power consumption $\wp(x)$ and the latency $\tau(x)$. Each metric has an optional maximum value and can be set either as a constraint (all values under the constraint are equally good) or as an objective to maximize.

Cost computation is not fully detailed in the model. We can deduce from definitions that PEs are sequential units of processing where tasks are time-multiplexed and that a task consumes $\wp_{avg} \times WCET$ energy for each execution. The power consumption for a task is considered independent of the PE executing it. The latency is computed after a complete mapping and scheduling of the application onto the architecture. The price and area of the system are the sums of PE prices and areas. Memory requirements are computed from data and instruction information respectively on edges and tasks of the application graph. Using an evolutionary algorithm, the CHARMED framework produces a set of potential heterogeneous architectures together with task mappings onto these architectures.

For performing DSE, the CHARMED quasi-MoA has introduced a model that jointly considers different forms of NFP metrics. The next section presents a third quasi-MoA named System-Level Architecture Model (S-LAM).

5.3 The System-Level Architecture Model (S-LAM) Quasi-MoA

In 2009, the S-LAM model [29] is proposed to be inserted in the PREESM rapid prototyping tool. S-LAM is designed to be combined with an application model based on extensions of the Synchronous Dataflow (SDF) dataflow MoC [14] and a transformation of a UML MARTE architecture description into S-LAM has been conducted in [1].

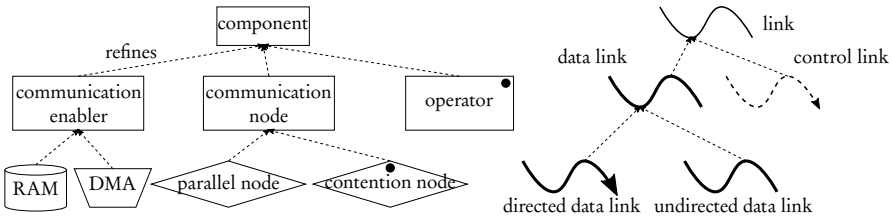

Fig. 15 Typology of the basic components in the S-LAM [29]. Leaf components are instantiable.

S-LAM defines a quasi-MoA $\Lambda = \langle M, L, t, p \rangle$ where M is a set of components, L is a set of links connecting them, and t and p respectively give a type and a property to components. As illustrated in Figure 15, there are five instantiable types

of components: `operator`, `parallel node`, `contention node`, `RAM`, and `DMA`.

`Operators` represent astract processing elements, capable of executing tasks (named actors in dataflow models) and of communicating data through links. Actors' executions are time-multiplexed over operators, as represented by the black dot on the graphical view, symbolizing scheduling. There are also `data links` and `control links`. A `data link` represents the ability to transfer data between components. `Control links` specify that an `operator` can program a `DMA`. Two actors cannot be directly connected by a data link. A route must be built, comprising at least one `parallel node` or one `contention node`. A `parallel node` N_p virtually consists of an infinite number of data channels with a given speed $\sigma(N_p)$ in Bytes/s. As a consequence, no scheduling is necessary for the data messages sharing a `parallel node`. A `contention node` N_c represents one data channels with speed $\sigma(N_c)$. Messages flowing over a `contention node` need to be scheduled, as depicted by the black dot in its representation. This internal component parallelism is the main novelty of S-LAM w.r.t. the AAA model. When transferring a data from operator O_1 to operator O_2, three scenarios are considered:

1. *direct messaging*: the sender operator itself sends the message and, as a consequence, cannot execute code simultaneously. It may have direct access to the receiver's address space or use a messaging component.
2. *DMA messaging*: the sender delegates the communication to a DMA. A `DMA` component must then be connected by a `data link` to a communication node of the route between O_1 and O_2 and a `control link` models the ability of the sender operator to program the DMA. In this case, the sender is free to execute code during message transfer.
3. *shared memory*: the message is first written to a shared memory by O_1, then read by O_2. To model this, a `RAM` component must be connected by a `data link` to a communication node of the route between O_1 and O_2.

An S-LAM representation of an architecture can be built where different routes are possible between two operators O_1 and O_2 [29]. The S-LAM model has for primary purpose system time simulation. An S-LAM model can be more compact than an AAA model because of internal component parallelism. Indeed, there is no representation of a bus or bus arbiter in S-LAM and the same communication facility may be first represented by a `parallel node` to limit the amount of necessary message scheduling, then modeled as one or a set of`contention nodes` with or without DMA to study the competition for bus resources. Moreover, contrary to the AAA model, operators can send data themselves. Figure 16 illustrates such a compact representation on the same platform example than in Figure 14. Local PE memories are ignored because they are considered embedded in their respective `operator`. The TeraNet NoC is modeled with a `parallel node`, modeling it as a bus with limited throughput but with virtually infinite inter-message parallelism.

The transfer latency of a message of M Bytes over a route $R = (N_1, N_2, ..., N_K)$, where N_i are communication nodes, is computed as $l(M) = min_{N \in R}(\sigma(N)) * M$.

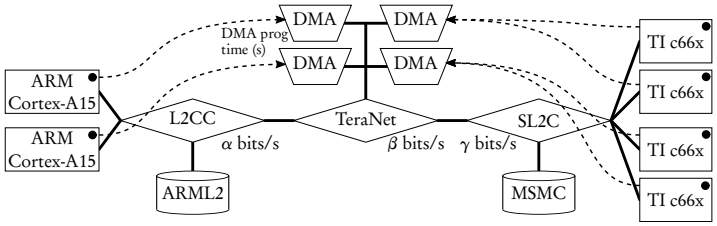

Fig. 16 Example of an architecture model with the S-LAM quasi-MoA.

It corresponds in the linear model presented in Figure 11 where the slope is determined by the slowest communication node. If the route comprises contention nodes involved in other simultaneous communications, the latency is increased by the time multiplexing of messages. Moreover, a `DMA` has an *offset* property and, if a `DMA` drives the transfer, the latency becomes $l(M) = offset + min_{NinR}(\sigma(N)) * M$, corresponding to the affine message cost in Figure 11.

As in the AAA model, an S-LAM `operator` is a sequential PE. This is a limitation if a hierarchical architecture is considered where PEs have internal observable parallelism. S-LAM operators have an operator ISA type (for instance ARMv7 or C66x) and each actor in the dataflow application is associated to an execution time cost for each operator type. S-LAM clearly separates algorithm from architecture but it does not specify cost computation and does not abstract computation cost.

S-LAM has introduced a compact quasi-MoA to be used for DSP applications. The next section presents one last quasi-MoA from literature.

5.4 The MAPS Quasi-MoA

In 2012, a quasi-MoA is proposed in [5] for programming heterogeneous Multiprocessor Systems-on-Chips (MPSoCs) in the MAPS compiler environment. It combines the multi-modality of CHARMED with a sophisticated representation of communication costs. The quasi-MoA serves as a theoretical background for mapping multiple concurrent transformational applications over a single MPSoC. It is combined with Kahn Process Network (KPN) application representations [15] [2] and is limited to the support of software applications.

The MAPS quasi-MoA is a graph $\Lambda = \langle M, L, t, p \rangle$ where M is a set of PEs, L is a set of named edges called Communication Primitives (CPs) connecting them, and t and p respectively give a type and a property to components. Each PE has properties $p(PE \in M) = (CM^{PT}, X^{PT}, V^{PT})$ where CM^{PT} is a set of functions associating NFP costs to PEs. An example of NFP is ζ^{PT} that associates to a *task* T_i in the application an execution time $\zeta^{PT}(T_i)$. X^{PT} is a set of PE attributes such as context switch time of the OS or some resource limitations, and V^{PT} is a set of variables, set late after application mapping decisions, such as the processor scheduling policy. A CP models a software Application Programming Interface (API) that is used to communicate

among *tasks* in the KPN application. A CP has its own set of cost model functions CM^{CP} associating costs of different natures to communication volumes. A function $\zeta^{CP} \in CM^{CP}$ is defined. It associates a communication time $\zeta^{CP}(N)$ to a message of N bytes. Function ζ^{CP} is a stair function modeling the message overhead and performance bursts frequently observed when transferring data for instance with a DMA and packetization. This function, displayed in Figure 11, is expressed as:

$$\zeta^{CP} : N \mapsto = \begin{cases} offset \quad if \quad N < start \\ offset + scale\_height \times \lceil (N - start + 1)/scale\_width \rceil \quad otherwise, \end{cases} \tag{4}$$

where $start$, $offset$, $scale\_height$ and $scale\_width$ are 4 CP parameters. The primary concern of the MAPS quasi-MoA is thus time. No information is given on whether the sender or the receiver PE can compute a task in parallel to communication. A CP also refers to a set of Communication Resources (CRs), i.e. a model of a hardware module used to implement the communication. A CRs has two attributes: the number of logical channels and the amount of available memory in the module. For example, a CR may model a shared memory, a local memory, or a hardware communication queue.

This quasi-MoA does not specify any cost computation procedure from the data provided in the model. Moreover, the MAPS architecture model, as the other architecture models presented in this Section, does not abstract the generated costs. Next section summarizes the results of studying the four formal architecture models.

5.5 Evolution of Formal Architecture Models

The four presented models have inspired the Definition 4 of an MoA. Theses formal models have progressively introduced the ideas of:

- architecture abstraction by the AAA quasi-MoA [13],
- architecture modeling for multi-dimensional DSE by CHARMED [17],
- internal component parallelism by S-LAM [29],
- complex data transfer models by MAPS [5].

The next section concludes this chapter on MoAs for DSP systems.

6 Concluding Remarks on MoA and quasi-MoAs for DSP Systems

In this chapter, the notions of Model of Architecture (MoA) and quasi-MoA have been defined and several models have been studied, including fully abstract models and language-defined models. To be an MoA, an architecture model must capture

efficiency-related features of a platform in a reproducible, abstract and application-agnostic fashion.

The existence of many quasi-MoAs and their strong resemblance demonstrate the need for architecture modeling semantics. Table 1 summarizes the objectives and properties of the different studied models. As explained throughout this chapter, LSLA is, to the extent of our knowledge, the only model to currently comply with the 3 rules of MoA definition (Definition 4).

Table 1 Properties (from Definition 4) and objectives of the presented MoA and quasi-MoAs.

Model	Repro-ducible	Appli. Agnostic	Abstract	Main Objective
AADL quasi-MoA	✓	✗	✗	HW/SW codesign of hard RT system
MCA SHIM quasi-MoA	✗	✗	✗	multicore performance simulation
UML MARTE quasi-MoAs	✗	✓/✗	✗	holistic design of a system
AAA quasi-MoA	✗	✓	✗	WCET evaluation of a DSP system
CHARMED quasi-MoA	✗	✓	✗	DSE of a DSP system
S-LAM quasi-MoA	✗	✓	✗	multicore scheduling for DSP
MAPS quasi-MoA	✗	✓	✗	multicore scheduling for DSP
LSLA MoA	✓	✓	✓	System-level modeling of a NFP

LSLA is one example of an MoA but many types of MoAs are imaginable, focusing on different modalities of application activity such as concurrency or spatial data locality. A parallel with MoCs on the application side of the Y-chart motivates for the creation of new MoAs. MoCs have the ability to greatly simplify the system-level view of a design, and in particular of a DSP design. For example, and as discussed by several chapters in this Handbook, MoCs based on Dataflow Process Networks (DPNs) are able to simplify the problem of system verification by defining globally asynchronous systems that synchronize only when needed, i.e. when data moves from one location to another. DPN MoCs are naturally suited to modeling DSP applications that react upon arrival of data by producing data. MoAs to be combined with DPN MoCs do not necessarily require the description of complex relations between data clocks. They may require only to assess the efficiency of "black box" PEs, as well as the efficiency of transferring, either with shared memory or with message passing, some data between PEs. This opportunity is exploited in the semantics of the 4 formal languages presented in Section 5 and can be put in contrast with the UML MARTE standard that, in order to support all types of transformational and reactive applications, specifies a generic clock relation language named CCSL [25].

The 3 properties of an MoA open new opportunities for system design. While abstraction makes MoAs adaptable to different types of NFPs, cost computation reproducibility can be the basis for advanced tool compatibility. Independence from application concerns is moreover a great enabler for Design Space Exploration methods.

Architecture models are also being designed in other domains than Digital Signal Processing. As an example in the High Performance Computing (HPC) domain, the Open MPI Portable Hardware Locality (hwloc) [11] models processing, memory and communication resources of a platform with the aim of improving the efficiency of HPC applications by tailoring thread locality to communication capabilities. Similarly to most of the modeling features described in this chapter, the hwloc features have been chosen to tackle precise and medium-term objectives. The convergence of all these models into a few generic MoAs covering different aspects of design automation is a necessary step to manage the complexity of future large scale systems.

Acknowledgements I am grateful to François Berry and Jocelyn Sérot for their valuable advice and support during the writing of this chapter.

7 List of Acronyms

AAA Adéquation Algorithm Architecture
AADL Architecture Analysis and Design Language
ADL Architecture Design Language
API Application Programming Interface
BER Bit Error Rate
B/M/D bus/multiplexer/demultiplexer
CCCR Computation to Communication Cost Ratio
CCSL Clock Constraint Specification Language
CN Communication Node
CP Communication Primitive
CPU Central Processing Unit
CR Communication Resource
DAG Directed Acyclic Graph
DMA Direct Memory Access
DPN Dataflow Process Network
DSE Design Space Exploration
DSP Digital Signal Processing
EDF Earliest Deadline First
FIFO First In, First Out data queue
FPGA Field-Programmable Gate Array
FSM Finite State Machine
GALS Globally Asynchronous Locally Synchronous
GCM Generic Component Model
GPP General Purpose Processor
GQAM Generic Quantitative Analysis Modeling
GRM Generic Resource Modeling
HLAM High-Level Application Modeling
HPC High Performance Computing

HRM Hardware Resource Modeling
hwloc Portable Hardware Locality
IP Intellectual Property core
ISA Instruction Set Architecture
KPN Kahn Process Network
LSLA Linear System-Level Architecture Model
LUT Lookup Table
MARTE Modeling And Analysis Of Real-Time Embedded Systems
MCA Multicore Association
MMU Memory Management Unit
MoA Model of Architecture
MoC Model of Computation
MPSoC Multiprocessor System-on-Chip
MSMC Multicore Shared Memory Controller
NFP Non-Functional Property
NoC Network-on-Chip
OMG Object Management Group
OS Operating System
OSI Open Systems Interconnection
PAM Performance Analysis Modeling
PE Processing Element
PLD Programmable Logic Device
PT Processor Type
PU Processing Unit
QoS Quality of Service
RAM Random Access Memory
RM Rate Monotonic
ROM Read-Only Memory
RSM Repetitive Structure Modeling
RTOS Real-Time Operating System
SAM Sequential Access Memory
SAM Schedulability Analysis Modeling (UML MARTE)
SDF Synchronous Dataflow
SHIM Software/Hardware Interface for Multicore/Manycore
S-LAM System-Level Architecture Model
SMP Symmetric Multiprocessing
SNR Signal-to-Noise Ratio
SRM Software Resource Modeling
TLM Transaction-Level Modeling
TU Transfer Unit
UML Unified Modeling Language
WCET Worst Case Execution Time

References

[1] Ammar M, Baklouti M, Pelcat M, Desnos K, Abid M (2016) Automatic generation of s-lam descriptions from uml/marte for the dse of massively parallel embedded systems. In: Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2015, Springer, pp 195–211

[2] Bacivarov I, Haid W, Huang K, Thiele L (2018) Methods and tools for mapping process networks onto multi-processor systems-on-chip. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of Signal Processing Systems, 3rd edn, Springer

[3] Bellard F (2005) QEMU, a Fast and Portable Dynamic Translator. In: USENIX Annual Technical Conference, FREENIX Track, pp 41–46

[4] Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, others (2011) The gem5 simulator. ACM SIGARCH Computer Architecture News 39(2):1–7, URL http://dl.acm.org/citation.cfm?id=2024718

[5] Castrillon Mazo J, Leupers R (2014) Programming Heterogeneous MPSoCs. Springer International Publishing, Cham, URL http://link.springer.com/10.1007/978-3-319-00675-8

[6] Chen Y, Chen L (2013) Video compression. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of Signal Processing Systems, 2nd edn, Springer

[7] Eker J, Janneck JW, Lee E, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y, et al (2003) Taming heterogeneity-the ptolemy approach. Proceedings of the IEEE 91(1):127–144

[8] Faugere M, Bourbeau T, De Simone R, Gerard S (2007) Marte: Also an uml profile for modeling aadl applications. In: Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on, IEEE, pp 359–364

[9] Feiler PH, Gluch DP (2012) Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language. Addison-Wesley

[10] Feiler PH, Gluch DP, Hudak JJ (2006) The architecture analysis & design language (AADL): An introduction. Tech. rep., DTIC Document

[11] Goglin B (2014) Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc). In: High Performance Computing & Simulation (HPCS), 2014 International Conference on, IEEE, pp 74–81

[12] Gondo M, Arakawa F, Edahiro M (2014) Establishing a standard interface between multi-manycore and software tools-SHIM. In: COOL Chips XVII, 2014 IEEE, IEEE, pp 1–3

[13] Grandpierre T, Sorel Y (2003) From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In: Formal Methods and Models for Co-Design, 2003. MEMOCODE'03. Proceedings. First ACM and IEEE International Conference on, IEEE, pp 123–132

[14] Ha S, Oh H (2013) Decidable dataflow models for signal processing: Synchronous dataflow and its extensions. In: Bhattacharyya SS, Deprettere EF,

Leupers R, Takala J (eds) Handbook of Signal Processing Systems, 2nd edn, Springer
[15] Kahn G (1974) The semantics of a simple language for parallel programming. In Information Processing 74:471–475
[16] Keutzer K, Newton AR, Rabaey JM, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE transactions on computer-aided design of integrated circuits and systems 19(12):1523–1543
[17] Kianzad V, Bhattacharyya SS (2004) CHARMED: A multi-objective co-synthesis framework for multi-mode embedded systems. In: Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on, IEEE, pp 28–40
[18] Kienhuis B, Deprettere E, Vissers K, van der Wolf P (1997) An approach for quantitative analysis of application-specific dataflow architectures. In: Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on, IEEE, pp 338–349
[19] Kienhuis B, Deprettere EF, Van Der Wolf P, Vissers K (2002) A methodology to design programmable embedded systems. In: Embedded processor design challenges, Springer, pp 18–37
[20] Larsen M (2016) Modelling field robot software using aadl. Technical Report Electronics and Computer Engineering 4(25)
[21] Lasnier G, Zalila B, Pautet L, Hugues J (2009) Ocarina: An environment for aadl models analysis and automatic code generation for high integrity applications. In: International Conference on Reliable Software Technologies, Springer, pp 237–250
[22] Lattner C, Adve V (2004) Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, IEEE Computer Society, p 75
[23] Lee EA (2006) The problem with threads. Computer 39(5):33–42
[24] Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proceedings of the IEEE 75(9)
[25] Mallet F, André C (2008) Uml/marte ccsl, signal and petri nets. PhD thesis, INRIA
[26] Mallet F, De Simone R (2009) Marte vs. aadl for discrete-event and discrete-time domains. In: Languages for Embedded Systems and Their Applications, Springer, pp 27–41
[27] Multicore Association (2015) Software/Hardware Interface for Multicore/Manycore (SHIM) - http://www.multicore-association.org/workgroup/shim.php/ (accessed 03/2017)
[28] OMG (2011) UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Object Management Group, Needham, MA
[29] Pelcat M, Nezan JF, Piat J, Croizer J, Aridhi S (2009) A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems. In: Proceedings of DASIP conference

[30] Pelcat M, Mercat A, Desnos K, Maggiani L, Liu Y, Heulot J, Nezan JF, Hamidouche W, Menard D, Bhattacharyya SS (2017) Reproducible evaluation of system efficiency with a model of architecture: From theory to practice. Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)
[31] Pimentel AD (2017) Exploring exploration: A tutorial introduction to embedded systems design space exploration. IEEE Design & Test 34(1):77–90
[32] Renfors M, Juntti M, Valkama M (2018) Signal processing for wireless transceivers. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of Signal Processing Systems, 3rd edn, Springer
[33] SAE International (2012) Architecture analysis and design language (aadl) - http://standards.sae.org/as5506c/ (accessed 03/2017)
[34] Shekhar R, Walimbe V, Plishker W (2013) Medical image processing. In: Bhattacharyya SS, Deprettere EF, Leupers R, Takala J (eds) Handbook of Signal Processing Systems, 2nd edn, Springer
[35] Stevens A (2011) Introduction to AMBA 4 ACE and big.LITTLE Processing Technology
[36] Texas Instruments (2015) 66AK2L06 Multicore DSP+ARM KeyStone II System-on-Chip (SoC) - SPRS930. Texas Instruments, URL http://www.ti.com/lit/pdf/sprs866e (accessed 03/2017)
[37] Van Roy P, et al (2009) Programming paradigms for dummies: What every programmer should know. New computational paradigms for computer music 104
[38] Wolf M (2014) High-performance embedded computing: applications in cyber-physical systems and mobile computing. Newnes