Formal and Virtual Multi-level Design Space Exploration

Letitia W. Li[1,3](✉), Daniela Genius[2](✉), and Ludovic Apvrille[1](✉)

[1] Télécom ParisTech, Université Paris-Saclay, Biot, France
{letitia.li,ludovic.apvrille}@telecom-paristech.fr
[2] Sorbonne Universités, UPMC Paris 06, LIP6, CNRS UMR 7606, Paris, France
daniela.genius@lip6.fr
[3] Institut VEDECOM, Versailles, France

Abstract. With the growing complexity of embedded systems, a systematic design process and tool are vital to help designers assure that their design meets specifications. The design of an embedded system evolves through multiple modeling phases, with varying levels of abstraction. A modeling toolkit should also support the various evaluations needed at each stage, in the form of simulation, formal verification, and performance evaluation. This chapter introduces our model-based engineering process with the supporting toolkit TTool, with two main design stages occurring at a different level of abstraction. A system-level design space exploration selects the architecture and partitions functions into hardware and software. The subsequent software design phase then designs and assesses the detailed functionality of the system, and evaluates the partitioning choices. We illustrate the design phases and supported evaluations with a Smart Card case study.

Keywords: Virtual prototyping · Embedded systems
System-level design · Telecommunications

1 Introduction

A systematic design methodology with supporting toolkit can help designers with the modeling and evaluation of the system, and involves supporting multiple design phases at varying levels of abstraction and different evaluation tools. The design of embedded systems is complicated by the need to design both its hardware and software components. Their design methodology can therefore be separated into two main phases.

A system-level design space exploration divides functions into hardware and software, based on system performance, safety and security requirements. Next, the software design phase includes the development of the detailed system functionality, and generation of code. However, since partitioning decisions are taken at a high level of abstraction – e.g., with highly abstracted hardware components –, it might be useful to validate – and possibly reconsider – partitioning choices during the software and hardware development phase.

© Springer International Publishing AG, part of Springer Nature 2018
L. F. Pires et al. (Eds.): MODELSWARD 2017, CCIS 880, pp. 47–71, 2018.
https://doi.org/10.1007/978-3-319-94764-8_3

Several works of research and tools have addressed system-level partitioning and evaluation of hardware platforms during the software development stage. However, the lack of integration between partitioning and system development makes it difficult to reconsider partitioning choices. Also, it is very common practice to test/execute software components on the local host, and to integrate them later on the target. Consequently, errors due to the interaction between hardware and software are discovered very late in the development cycle – e.g., during the integration phase. Unfortunately, these errors may lead to reconsidering partitioning decisions. To minimize time needed for a designer to re-do a partitioning, a toolkit should ideally minimize the manual work needed to take in consideration those errors and better connect the two abstraction levels.

Thus, our work focuses on the development of a fully integrated model-driven approach to handle both partitioning and software development. Our contribution supports both the selection of candidate hardware and software architectures, and a software development approach that allows designers to evaluate the relevance of the previously selected architectures early in the development process. Automated model transformation and verification techniques - formal verification, simulation, virtual prototyping - are supported for that purpose. Our contribution presents an easy-to-comprehend methodology integrating these two stages contained within a single modeling framework (*TTool*) [1]. Previous work [2] described our design process at multiple levels, but lacked detailed automated performance analysis regarding performance metrics such as latency. In Sect. 2, we present the related work of other system-level design toolkits. Section 3 describes our overall methodology. Section 4 details the Smart Card case study that is then used to exemplify the high-level design space exploration (Sect. 5) and the software component design (Sect. 6). Finally, we present discussion and perspectives on future work in Sect. 7.

2 Design Techniques for Embedded Systems

Many frameworks have been proposed for the design of embedded systems. They offer modeling capabilities at different levels of abstraction and using various approaches, such as Platform-Based Design, Model-Driven Engineering, etc. These tools offers model edition capabilities and can verify models with different simulation and verification tools. Some of them also target executable code generation.

2.1 Design Space Exploration Approaches (with Simulation and Formal Techniques)

Ptolemy [3,4] proposes a modeling environment for the integration of diverse execution models, in particular hardware and software components. If design space exploration can be performed with Ptolemy, its first intent is the simulation of the modeled systems. The co-simulation facility of Ptolemy II is demonstrated in [5]. Their approach relies on both a System-C architecture model and

a functional model. The paper describes how to use different abstraction levels to model systems.

Virtual prototyping of MPSoC is often hampered by slow simulation. Among approaches generating SystemC code, the virtual prototyping of [6] generates code for the TLM (transactional) level, which is more efficient to simulate but less detailed. A team from KIT [7] proposes a methodology for fast parallel simulation, which is based on TLM and though with lost accuracy, even if clock cycles can be taken into account. MPSoCSim [8], recently presented, proposes OVP processor models to simulate NoC-based System-on-chip. Bus simulation is TLM2.0 based. We chose to perform our simulations on cycle accurate bit accurate level and use a simulator based on fully static scheduling [9], which makes it 10 to 20 times faster than the SystemC event-based simulator.

Capella [10] relies on Arcadia, a comprehensive model-based engineering method. It is intended to check the feasibility of customer requirements, called *needs*, for very large systems. Capella provides architecture diagrams allocating functions to components, and advanced mechanisms to model bit-precise data structures. Capella is however more business focused, and lacks formal verification capabilities.

In POLIS [11], applications are described as a network of state machines. Each element of the network can be mapped on a hardware or a software node. This approach is more oriented towards application modeling, even if hardware components are closely associated to the mapping process. Metropolis [12], an extension of POLIS, targets heterogeneous systems, and architectural and application constraints are closely interwoven. Metropolis is based on a meta-model of a *network of concurrent objects*, with a formal semantics. Applications are described in detail and simulated with the help of instruction set simulators (ISS). This approach is more oriented towards application modeling, even if hardware components are closely associated to the mapping process. While our approach uses Model-Driven Engineering, Metropolis uses Platform-Based Design.

Sesame [13] proposes modeling and simulation features at several abstraction levels for Multiprocessor System-on-Chip architectures. Pre-existing virtual components are combined to form a complex hardware architecture. Models' semantics vary according to the levels of abstraction, ranging from Kahn process networks (KPN [14]) to data flow for model refinement, and to discrete events for simulation. Currently, Sesame is limited to the allocation of processing resources to application processes. It models neither memory mapping nor the choice of the communication architecture.

The ARTEMIS [15] project originates from heterogeneous platforms in the context of research on multimedia applications in particular. It is strongly based on the Y-chart approach [16]. Application and architecture are clearly separated: the application produces an event trace at simulation time, which is then read in by the architecture model. However, behavior depending on timers and interrupts cannot be taken into account.

MARTE [17] shares many commonalities with our approach, in terms of the capacity to separately model communications from the pair application-

architecture. However, it intrinsically lacks a separation between control and message exchange. Even if the UML profile for MARTE adds capabilities to model Real Time and Embedded Systems, it does not specifically support architectural exploration. Other works based on UML/MARTE, such as Gaspard2 [18], are dedicated to both hardware and software synthesis, relying on a refinement process based on user interaction to progressively lower the level of abstraction of input models. However, such a refinement does not completely separate the application (software synthesis) or architecture (hardware synthesis) models from communication.

Saxena and Karsai [19] introduce an abstract design space exploration framework, and its integration into design space exploration solvers, thus paving the way for customized embedded systems explorations. They define metrics (e.g., memory size) that are related to WCET. On the contrary, DIPLODOCUS does not assume any WCET, but closely evaluates possible scenarios with simulation and formal verification techniques.

The capacity of languages and models to support abstractions for designing embedded systems is discussed in [20]. In particular, MARTE is evaluated against the Y-Chart scheme. Our papers enhances their discussion with the refinement between two abstraction levels (partitioning and prototyping).

2.2 Code Generation Approaches

Rhapsody [21] can automatically generate software, but not hardware descriptions from SysML. MDGen from Sodius [22] adds timing and hardware specific artifacts such as clock/reset lines automatically to Rhapsody models, generates synthesizable, cycle-accurate SystemC implementations, and automates exploration of architectures.

The Architecture Analysis & Design Language (AADL [23]), a standard from the International Society of Automotive engineer (SAE), allows the use of formal methods for safety-critical real-time systems in avionics, automotive among other domains. It comprises a textual and a graphical representation but does not a priori contain tool support for code generation, even if specific contributions proposes code generator for specific domains, e.g. for avionics systems. In that case, the generated code can be executed for within a specific platforms, for instance for ARINC653 systems. Similar to our environment, a processor model can have different underlying implementations and its characteristics can easily be changed at the modeling stage. Recently, [24] developed a model-based formal integration framework which endows AADL with a language for expressing timing relationships.

Bombieri et al. [25] propose a method ranging from system specification to code generation, with an intermediate HW/SW partitioning stage. Their method is compliant with SW components, device driver generation, a software wrapper – e.g., to handle interrupts – and High-level synthesis for HW components. While being more advanced on code generation issues, simulation and formal verification, as well as iterations between partitioning and prototyping is not addressed as deeply as in our contribution.

Batori [26] proposes a design methodology for telecommunication applications. From use cases, the method proposes several formalisms to capture the application structure ("interaction model") and behavior (Finite State Machine) and for its deployment from which executable code can be generated. The platform seems limited to specific components ("Runes component") – we could call it Specific Platform-Based design – and no design exploration seems possible. Additionally, the code generation process targets a real platform, and not a prototyping environment.

As we explain in the next section, our approach combines both HW/SW partitioning and software development and prototyping, with formal verification and simulation offered for most views and abstraction levels, including safety, performance and security evaluation.

3 Methodology

3.1 Modeling Phases

The advantages of our methodology lie in its support of multiple phases of the design process, and its ability to evaluate a design with a diverse range of tools. These advantages have allowed our methodology to be applied for the modeling of a wide range of real-world systems, including automotive systems, telecommunications, security protocols, etc. [27–29]. Our method relies on a set of UML/SysML views supported with the same environment/toolkit (as shown in Fig. 1. The method is organized as follows:

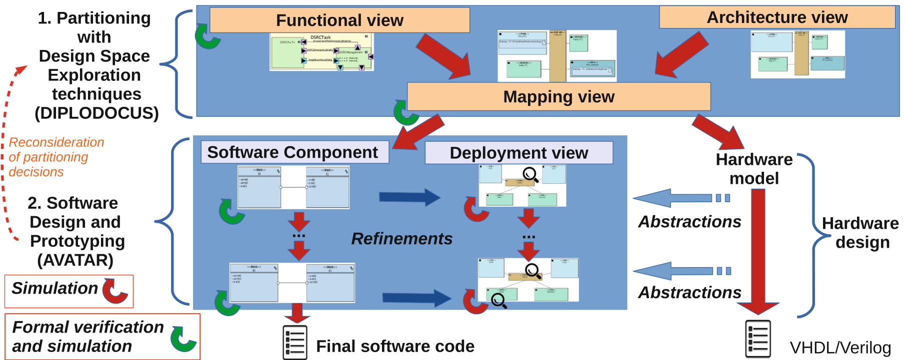

Fig. 1. Overall approach.

1. We start with system-level hardware/software partitioning based on design space exploration techniques. This phase contains three sub-phases: modeling of the functions to be realized by the system ("functional view"), modeling of the candidate architecture ("architecture view") expressed as an assembly of highly abstracted hardware nodes, and the mapping phase ("mapping

view"). A function mapped over a processor is considered a software function. On the contrary, a function mapped over a hardware accelerator corresponds to a custom ASIC. At this stage, we are concerned mostly with how communications and function affect the performance of a mapping, so we do not need to concern ourselves with the exact behavior of internal task behavior or contents of communications. Logical communication between functions are also expected to be mapped on a "communication path" consisting of buses, bridges, memories, Direct Memory Access controllers, networks-on-chip, etc.
2. Once a mapping has been decided, i.e., the system is fully partitioned between software and hardware functions, the design of the software and the hardware can start. Our approach offers software modeling while taking into account hardware parameters. Thus, a software component view is used to build the system software architecture and behavior, and a deployment view displays how the software components relate to the hardware components. The model of software and hardware components is more refined than in partitioning, which means that simulations and proofs are much more complex and take more time.

TTool [1], a free and open-source toolkit, supports the entire method with SysML diagrams. TTool includes UML/SysML diagram editors, compilers to perform model-to-specifications transformations, model-checkers and simulation engines.

3.2 Simulation, Verification and Prototyping

During the methodological phases, simulation and formal verification help to determine if safety, performance and security requirements are fulfilled. TTool offers a press-button approach for performing these proofs. Model transformations translate the SysML models into an intermediate form that is sent into the underlying simulation and formal verification toolkits - some of them are built into TTool, while others are third party toolkits. In all cases, backtracing to UML/SysML models is performed to better inform the users about the verification results.

During functional modeling – our highest abstraction level –, verification intends to identify general safety properties (e.g., absence of deadlock situations). At the mapping stage, verification intends to check if performance and security requirements are met. As researchers demonstrate the increasing number of hacks on embedded systems, it becomes important to detect their security flaws before mass-production. The security of communications depends on the architecture, as we explain in Subsect. 6.2.

During software design, software components can be verified independently from any hardware architecture in terms of safety and security. For example, when designing a component implementing a security protocol, the reachability of the states and absence of security vulnerabilities can be verified. TTool support for integrated formal verification tools helps a designer ensure the safety and security of his/her design.

When the software components are more refined, it becomes important to evaluate performance. Since the target system is commonly not yet available, our approach offer two facilities. (i) A deployment view is used to map software components over hardware nodes. Their semantics is much more concrete – i.e., less abstract – than the one used for partitioning. (ii) A press-button approach can transform the deployment view into a SoCLib specification built upon virtual component models [30].

SoCLib is a public domain library of models written in SystemC, targeting shared-memory architectures based on the *Virtual Component Interconnect* protocol [31]. Hardware is described at several abstraction levels: TLM-DT (Transaction level with distributed time), CABA (Cycle/Bit Accurate), and RTL (Register Transfer Level). SoCLib also contains a set of performance evaluation tools [32,33]. CABA level simulation allows measurement of cache miss rates, latency of memory accesses and of any transactions on the interconnect, fill state of the buffers, taking/releasing of locks etc. in the context of video streaming and telecommunication applications [33].

A variety of low level performance measuring tools exist for SoCLib, as described in [32,34]. However, such approaches are purely based on simulating on the virtual prototype i.e. at a low level of abstraction, and lack the possibility to formally verify the application model and give it precise semantics. Moreover, they are more accessible to researchers than to engineers, nowadays very much at ease in the UML/SysML world. Hardware elements – i.e. *topcells* – are either described by hand, which is error-prone, or generated, making them not easily readable.

Since SoCLib hardware models are much more precise than partitioning models, precise timing and hardware mechanisms – e.g. cache miss – can be evaluated. If the performance results differ too greatly from the results obtained during the design space exploration stage – e.g., a cache miss ratio – then, the design space exploration shall be performed again to assess if the selected architecture is still the best according to the system requirements. If not, the definition of software components may be (re)designed. Once the iterations over the high-level design space exploration and the low level virtual prototyping of software components finished, software code can be generated from the most refined software model.

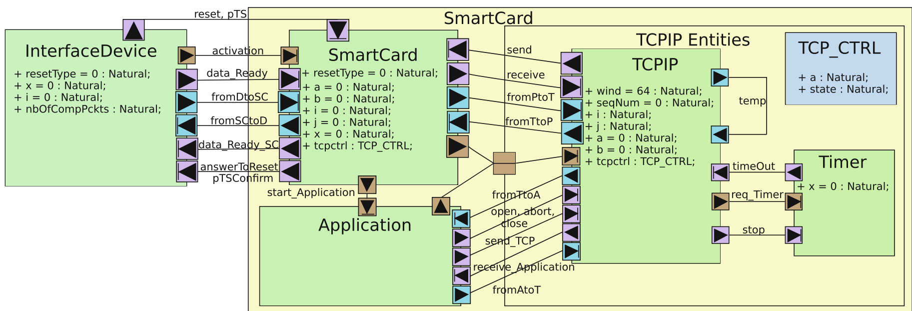

Fig. 2. Functional view (DIPLODOCUS) of the SmartCard application.

4 Case Study

Our methodology is illustrated by a "Smart Card" application. The smart card is meant to be plugged into a reader that exchanges information with the interior of the application by TCP formatted packets. The data transfer can be aborted, for example, because the smart card was unplugged. The reader (*InterfaceDevice*) signals the smart card to start, while the smart card controller handles the initialization of the other functions (e.g., the application and the network stack). In the next sections, we present modeling and analysis of the Smart Card application at the different design phases.

5 Partitioning with DIPLODOCUS

5.1 Models

The HW/SW partitioning phase, implemented in the DIPLODOCUS profile of TTool, models the abstract, high-level functionality of a system [35] and general architecture. It follows the Y-chart approach (as shown in the upper right section of Fig. 1), first modeling the abstract functional tasks (Application View), candidate architectures (Architectural View), and finally mapping tasks to the hardware components (Mapping View) [16]. Before the next stage, simulation and formal verification ensure that our design meets performance, behavioral, and schedulability requirements.

Application Modeling, Architectural Modeling, and Mapping are presented in detail in the rest of this section, using the smart card application as an example.

Figure 2 displays the functional view built upon 5 functional blocks: *InterfaceDevice* represents the interface with the reader and the internals of the smart card. *SmartCard* features the main controller. *Application* mostly models data exchanges that can occur with the reader. *TCPIP* and its *Timer* model the network stack. Exchanges between blocks are modeled with events, requests and data exchanges.

Application View. The Application View comprises of a set of communicating tasks, as shown in Fig. 2. The behavior of tasks is described abstractly. Functional abstraction allows us to ignore the exact computations and data processing of algorithms, and considers only computation complexity and data transfer size. Each individual task describes its abstract functional behavior using communication operators, computation elements, and control elements. Data abstraction allows us to consider only the size of data sent or received, and ignore details such as type, values, or names. On the Component Design Diagram, an extension of the SysML Block Instance Diagram, the designer specifies the list of tasks, and within the task, attributes and ports indicating communication.

Architectural View. The architectural model (consider only hardware components of Fig. 3, i.e. without the artifacts) displays the underlying architecture as a network of abstract execution nodes, communication nodes, and storage

nodes. Execution nodes consist of CPUs and Hardware Accelerators, defined by parameters for simulation. All execution nodes must be described by data size, instruction execution time, and clock ratio. CPUs can further be customized with scheduling policy, task switching time, cache-miss percentage, etc. Figure 4 shows processor parameters. Communication nodes include bridges and buses. Buses connect execution and storage nodes for task communication and data storage or exchange, and bridges connect buses. Buses are characterized by their arbitration policy, data size, clock ratio, etc., and bridges are characterized by data size and clock ratio. Storage nodes are Memories, which are defined by data size and clock ratio.

Mapping View. Mapping partitions the application into software and hardware and also specifies the location of their implementation and of their communication on the architectural model. A task mapped onto a processor will be implemented in software, and a task mapped onto a hardware accelerator will be implemented in hardware. The exact physical path of a data/event write may also be specified by mapping channels to buses and bridges. More complex communication schemes can be modeled with another view, which is part of recent work [36]. The mapping of Fig. 3 shows that the InterfaceDevice is mapped to a specific hardware execution node, while TCPIP, SmartCard and Application

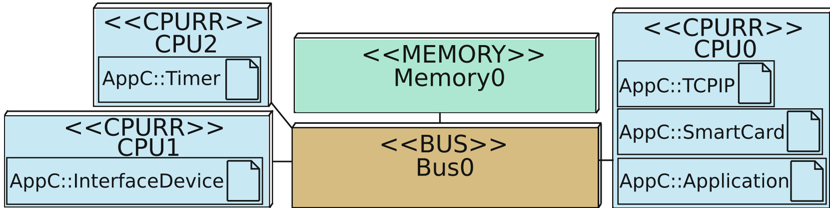

Fig. 3. Mapping view (DIPLODOCUS).

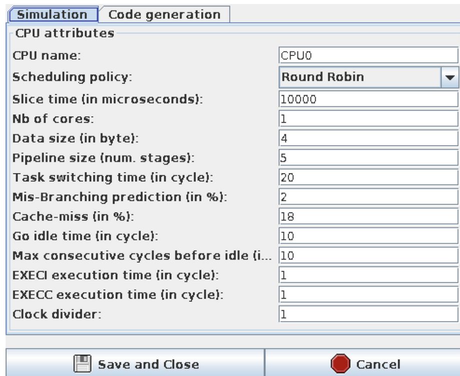

Fig. 4. Adapting processor parameters in DIPLODOCUS.

are mapped to a general purpose processor - actually, the main microcontroller of the smart card. Also, the timer is implemented with a dedicated execution node.

5.2 High-Level Simulation and Verification

Simulation of DIPLODOCUS partitioning specifications involves executing tasks on the different hardware elements. Each computation transaction executes for a variable time depending on execution cycles, CPU parameters and bus/memory behavior when transactions require data exchanges. The simulation shows performance results like bus usage, CPU usage, execution time, etc. Results are backtraced to the different views, with an example shown in Fig. 5. One can notice the high average load of the main microcontroller (91%). Also, TTool can generate a *vcd* trace to view detailed bus/CPU activity in gtkwave of a single execution sequence. TTool can also assists the user by automatically generating all possible architectures and mappings, and summarizes performance results of each possible mapping. Users are provided with the "best" architecture under specified criteria, such as minimal latency or bus/CPU load.

For a given mapping, the user can also generate the system reachability graph. The entire graph along with an enhanced excerpt is given in Fig. 6. All paths are terminated with a red state. The last actions before each red or termination state specifies the number of cycles corresponding to the path leading to that termination state. For example, the termination state "84" is preceded by an action "allCPUsTerminated<166>" which means that this system path contains 166 cycles.

TTool also makes it possible to list all termination or deadlock states (see Fig. 7): the graph contains 10 terminations states with a duration in number of cycles ranging from 20 to 247. In the shortest path, the connection was aborted after a few exchanges. On the contrary, in the longest execution path, the smart card exchanges several TCP packets. These timings are to be confirmed with more concrete software and hardware components in the design stage.

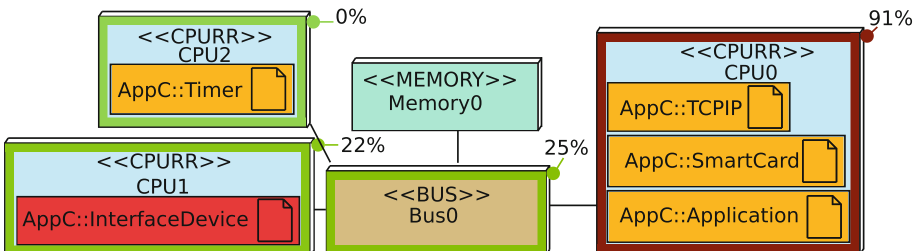

Fig. 5. Load of CPUs and buses after a simulation.

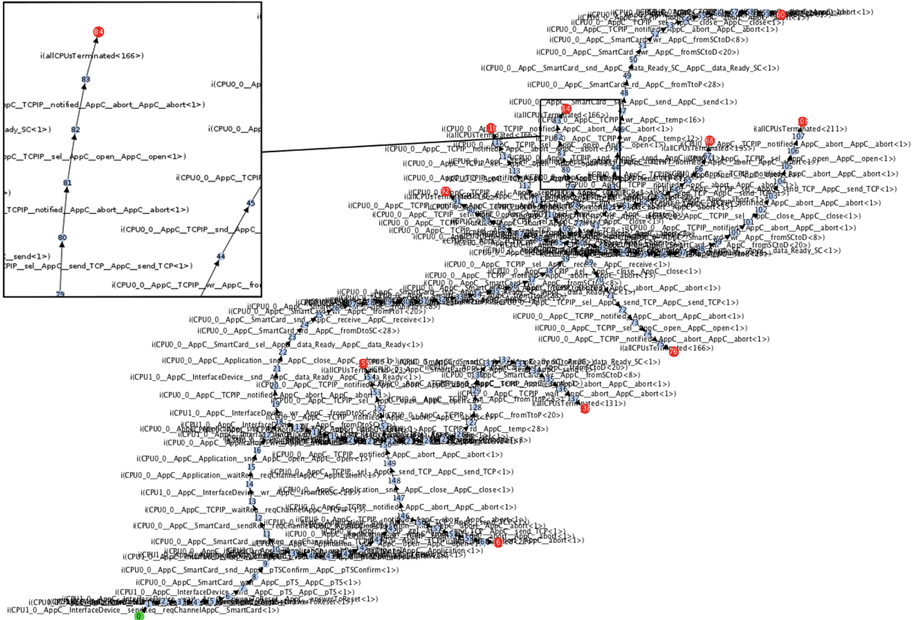

Fig. 6. Reachability graph of the mapping view. (Color figure online)

6 Software Design with AVATAR/SoCLib

Once partitioning is complete, the AVATAR methodology [37] allows the user to design the software, perform functional simulation and formal verification, and finally test the software components in a virtual prototyping environment. Where partitioning models represent an algorithm as an abstract execution spanning a duration, the software design models details of algorithms, including their attributes, int/float operations, control operators, etc.

Shortest Paths | Longest Paths
General info. | Statistics | Deadlocks

States	(origin, action)	
108	(107, i(allCPUsTerminated<211>))	[0] -- i(CPU1_0_AppC_
116	(115, i(allCPUsTerminated<166>))	[0] -- i(CPU1_0_AppC_
138	(137, i(allCPUsTerminated<131>))	[0] -- i(CPU1_0_AppC_
155	(154, i(allCPUsTerminated<23>))	[0] -- i(CPU1_0_AppC_
161	(160, i(allCPUsTerminated<20>))	[0] -- i(CPU1_0_AppC_
60	(59, i(allCPUsTerminated<247>))	[0] -- i(CPU1_0_AppC_
68	(67, i(allCPUsTerminated<195>))	[0] -- i(CPU1_0_AppC_
76	(75, i(allCPUsTerminated<166>))	[0] -- i(CPU1_0_AppC_
84	(83, i(allCPUsTerminated<166>))	[0] -- i(CPU1_0_AppC_
92	(91, i(allCPUsTerminated<166>))	[0] -- i(CPU1_0_AppC_

Close

Fig. 7. List of termination states in the reachability graph. The number of cycles on each path is given along with the last action before a termination state.

6.1 Software Components

Figure 8 shows the software components of the smart card case study modeled using an AVATAR block diagram. These modeling elements have been selected as software elements during the previous modeling stage (partitioning). Software components are grouped into the different applications running on the Smart Card using a hierarchical block called *SmartCard*.

- **Interface Device** initiates the connection and then communicates with the Smart Card.
- **SmartCard Controller** manages communication between the Interface Device, application, and TCPIP.
- **Application** communicates with the TCPIP application and sends and receives packets.
- **TCPIP** manages the TCP connection.
- **TCPPacketManager** manages packet transmission and storage.

The AVATAR model can be functionally simulated using the integrated simulator of our toolkit, which takes into account temporal operators but completely ignores hardware, operating systems and middleware. While being simulated, the model of the software components is animated. This simulation aims at identifying logical modeling bugs. Figure 9 shows the state machine of the Smart Card Controller, and Fig. 10 shows a visualization of a generated sequence diagram. Also, a reachability graph can be generated and analyzed.

6.2 Formal Verification

As previously described, TTool includes its own formal verification tools to e.g. generate a reachability graph, minimize the graphs, and check if a given reachability of liveness property is satisfied.

Alternatively, UPPAAL [38] may also be used from TTool to evaluate safety and liveness properties. UPPAAL is a a model checker for networks of timed automata, the behavioral model of a system to be verified is first translated into a UPPAAL specification to be checked for desired behavior. For example, UPPAAL may verify the lack of deadlock, such as two threads both waiting for the other to send a message. Behavior may also be verified through "Reachability", "Leads to", and other general statements. The designer can indicate which states in the Activity Diagram or State Machine Diagram should be checked if they can be reached in any execution trace. "Leads to" allows us to verify that one state must always be followed by another. Other user-defined UPPAAL queries can check if a condition is always true, is true for at least one execution trace, or if it will be true eventually for all execution traces. These statements may be entered directly on the UPPAAL model checker, or permanently stored on the model as pragma to be verified in UPPAAL.

For example, for our case study, we can verify that the TCP Packet Manager is capable of sending the storePacket signal, that the Application can abort and thereby stop, and the Smart Card Controller can send a packet. Figure 11 shows

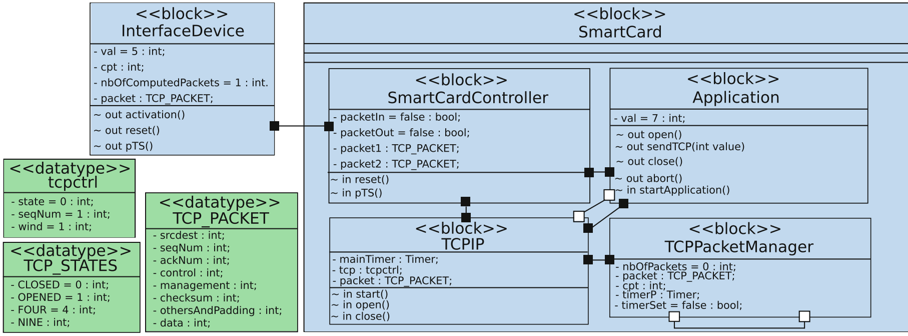

Fig. 8. Avatar block diagram.

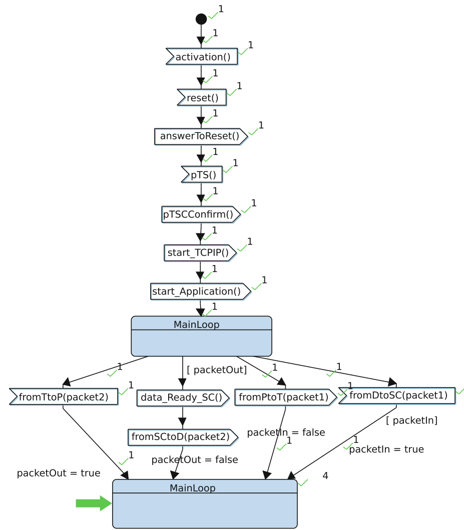

Fig. 9. High level simulation: annotated automaton.

the UPPAAL verification window which allows the user to customize which queries to execute, and then returns the results regarding whether each query is satisfied or not. In our example, the three states we queried are all reachable.

Formal verification of security is performed using ProVerif, a verification tool operating on pi-calculus specifications [39]. A ProVerif specification consists of a set of processes communicating on public and private channels. Processes can split to create concurrently executing processes, and replicate to model multiple executions (sessions) of a given protocol. Cryptographic primitives such as symmetric and asymmetric encryption or hash can be modeled through constructor and destructor functions. ProVerif assumes a Dolev-Yao attacker, which is a threat model in which anyone can read or write on any public channel, create new messages or apply known primitives.

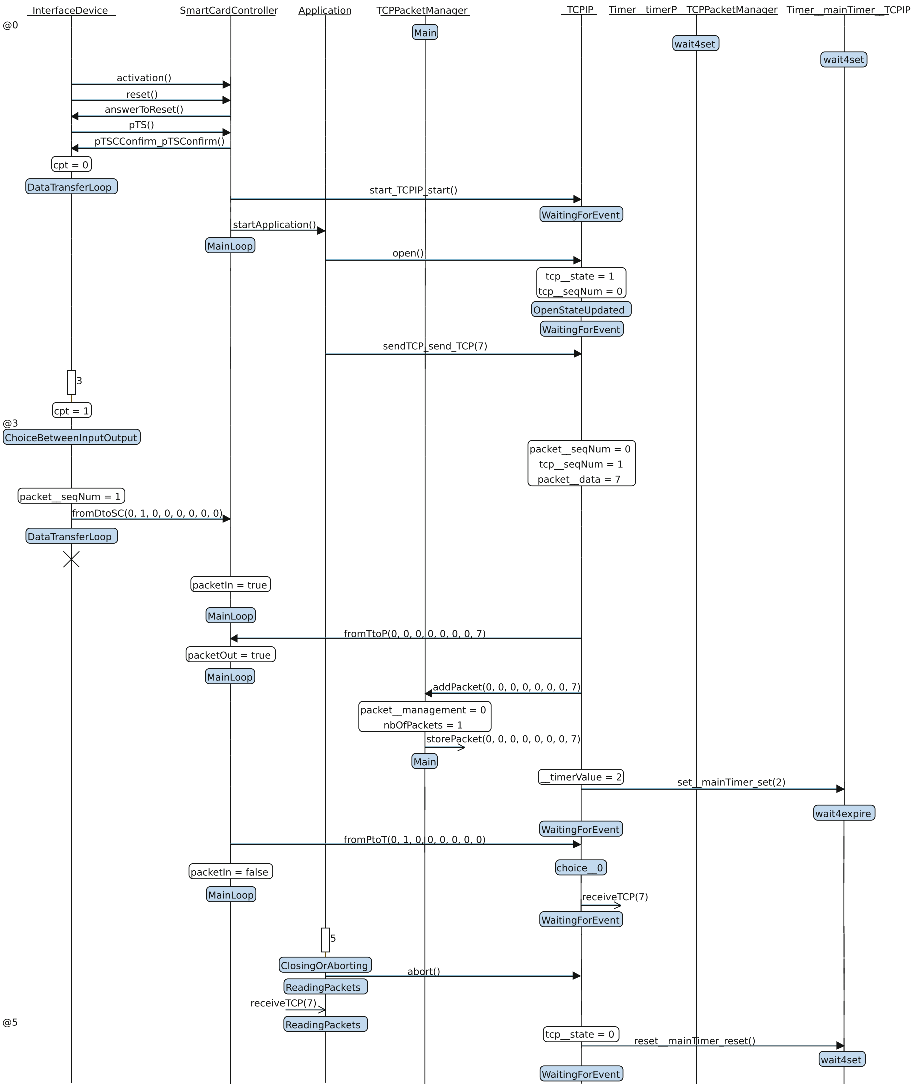

Fig. 10. High level simulation: generated sequence diagram.

ProVerif verifications query the properties of *reachability*, *confidentiality*, and *authenticity*. Reachability of an element (within the Activity Diagram or State Machine) determines if there exists an execution trace of the model in which this element is reached. Confidentiality of data refers to if the attacker can recover that data by listening and sending messages, and performing computations. Authenticity determines if the data received during a message exchange is necessarily the same as the data sent.

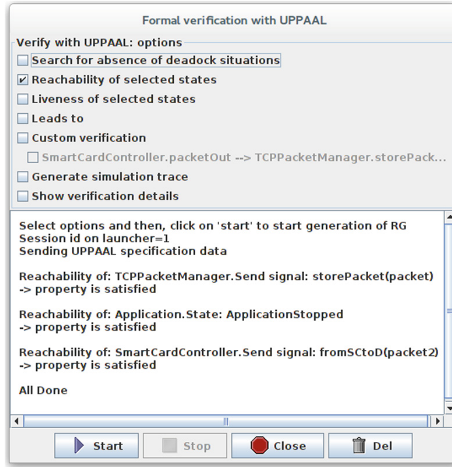

Fig. 11. UPPAAL formal verification.

In DIPLODOCUS models, security modeling and verification determines the security mechanisms required to secure critical data based on an architecture and mapping, and also impact on performance due to the added security. Certain architectural buses can be modeled to be accessible to an attacker. Abstract security operators then model the encryption/decryption of channel data and the impact of security on performance. Recent work [40] describes how the architecture and mapping selected during HW/SW Partitioning affects the security of communications, and security-related operations impacts the safety and performance of a system.

AVATAR models describe the detailed implementations of security mechanisms, and verifies the security of critical attributes [41]. The security verification determines the confidentiality of keys and specific attributes, the authenticity of encrypted exchanges over public channels accessible to an attacker, and the ability of a encryption algorithm to terminate correctly.

6.3 Prototyping

To prototype the software components with the other elements of the destination platform (hardware components, operating system), a user must first map them to a model of the target system. Mapping can be performed using the new deployment features introduced in [27]. Our toolkit thus supports use of AVATAR Deployment Diagrams. It features a set of hardware components, their interconnection, tasks, and channels.

In the partitioning phase, an architecture with two CPUs was selected. Tasks destined to become software tasks are mapped onto the CPUs, which is the case for all tasks in our example; it is also possible to realize other tasks as hardware accelerators. Now, in the prototyping phase, things are different since the

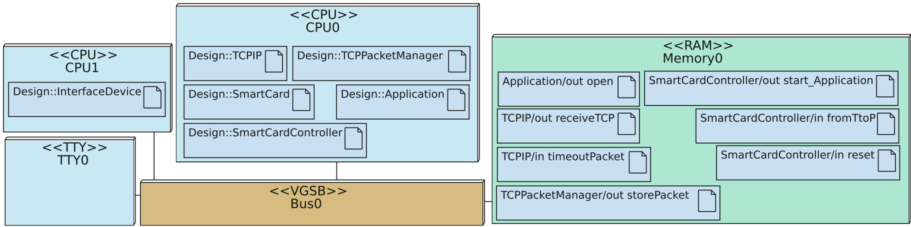

Fig. 12. Smart card deployment diagram.

AVATAR models includes only software tasks that are thus mappable only on general-purpose processors. Consequently, each hardware accelerator of the prototyping platform in SoCLib needs to be specifically developed. Which requires a significant effort. We do not consider that case because the smartcard is fully software implemented.

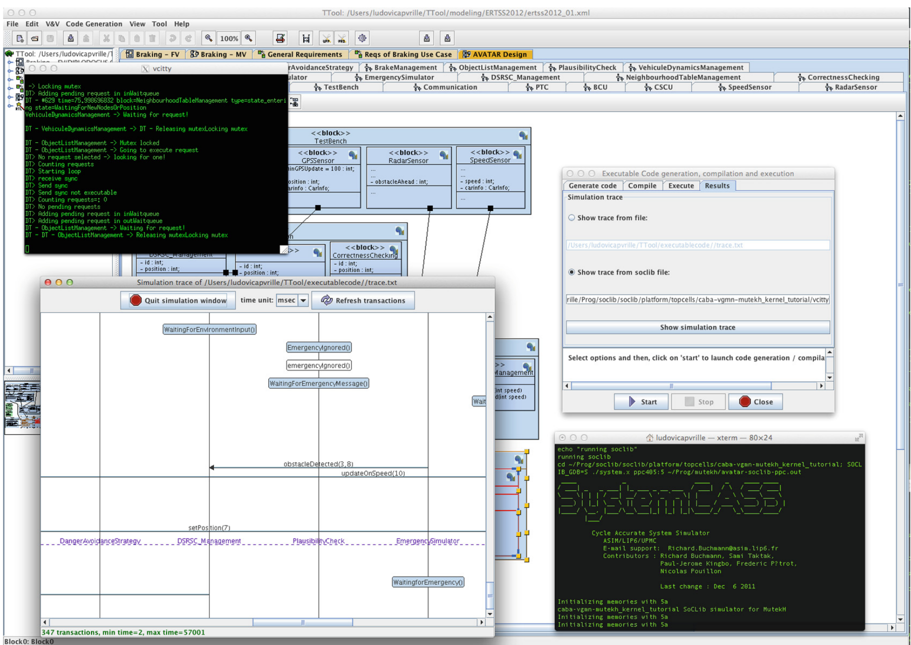

Fig. 13. AVATAR/SoCLib prototyping environment in TTool.

If the user has to explicitly model several properties pertaining to mapping, e.g. CPUs and memories parameters, the simulation infrastructure and interrupt management are added transparently to the top cell during the transformation into a SoCLib platform.

Figure 12 shows the Deployment Diagram, containing two CPUs, one memory bank and one TTY. The *InterfaceDevice* block is mapped onto CPU1, and the other five blocks are mapped onto CPU0. Each signal between AVATAR blocks is translated into a software channel mapped to on-chip RAM (for more detail, see [42]). In the case study, there are twenty-nine such signals, translated into twenty-nine SoCLib channels, which are all mapped on the single RAM, also containing the AVATAR runtime and the operating system.

From the Deployment Diagram, a SoCLib prototype is generated as described in [27]. This prototype consists of a SystemC top cell, the embedded software in the form of POSIX threads compiled for the target processors, and the embedded operating system [43]. Figure 13 from [2] shows an overview of the prototyping tool, with the simulation trace, code generation, and SocLib windows, and model in the back.

Fig. 14. Panel for varying cache associativity in SoCLib prototype.

6.4 Capturing Performance Information

We now show how performance information can be obtained by running simulations with the SoCLib virtual prototype of the SmartCard use case. The experiments shown here use a MP-SoC based on two general purpose PowerPC 405 processor cores running with 800 MHz. Later on, we plan to rely on a microcontroller, which would be more realistic for the SmartCard example. As a central interconnect, we use the VCI Generic Serial Bus (VGSB).

Although accelerated using the technique described in [9], the cycle accurate bit accurate (CABA)-level simulation is quite slow. It allows however detailed measurement of per-processor cache miss rates, latency of any transaction on the interconnect, etc. Since SoCLib hardware models are much more precise than the ones used at the design space exploration level, precise timing of the use of hardware mechanisms such as locks can be evaluated. However, these evaluations take considerable time compared to high-level simulation/evaluation.

As previously stated, the SoCLib prototype allows a designer to evaluate each processor separately, which is particularly useful for detecting unbalanced CPU loads, indicated by the Cycles per Instruction (CPI) metric.

In the following three paragraphs, we investigate three performance metrics: CPI, cache misses and latencies.

CPI. An overview of performance problems can be obtained using the numbers of Cycles per Instruction (CPI). It represents all phenomena that can slow down execution of instructions by the processors, such as memory access latency, interconnect contention, overhead due to context switching, etc.

Using these metrics, we can observe that CPU0 has a high average load – this issue was similarly noticed during the partitioning stage. Figure 15 shows that this CPU is far more challenged than CPU1 containing only the InterfaceDevice. The reason for this is due to the fact that implementing the semantics of synchronous channels requires a central request manager. Requests are stored in waiting queues for synchronous communications, in order to be canceled when they became obsolete. Requests that observe a delay before execution have to be waken up. Future work will address a better distribution of this functionality, called the *AVATAR runtime*, over the entire MPSoC architecture.

We also observe that adding cache associativity does not automatically improve the CPI. The application is characterized by an uneven mixture of small accesses (for example, *open* or *abort signals* which take one byte), and accesses to data of *packet* type which, as can be seen in the *Data Type* Block of the Block Diagram of Fig. 8, are composed of eight integers.

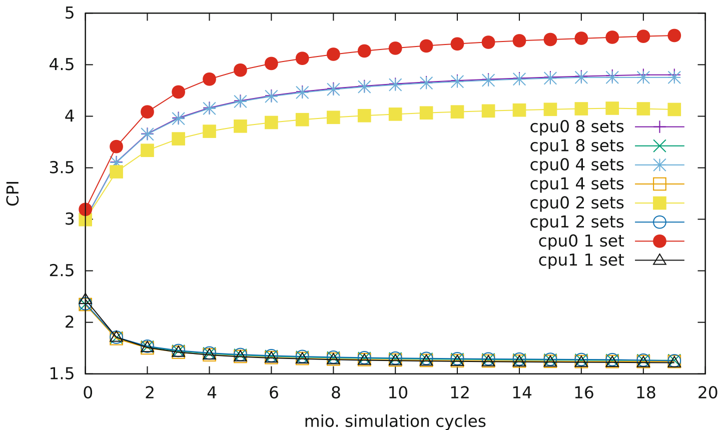

Fig. 15. CPI measured at CABA level.

Cache Misses. One important parameter of the CPU used in DIPLODOCUS partitioning is the overall cache miss rate, which is initially estimated to be 18% in DIPLODOCUS (see line *Cache-miss* in Fig. 4). While the estimate of

cache misses includes both data and instruction cache misses, we measure them separately. Instruction cache miss rates will be higher for the cache of CPU0 because the central request manager runs on this CPU, as noted in the previous paragraph.

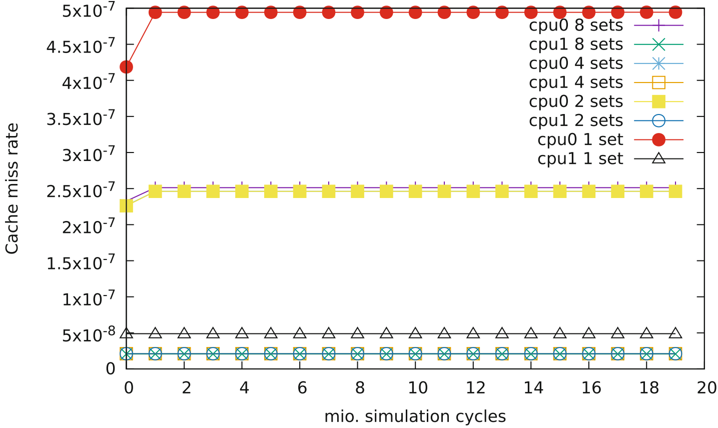

Fig. 16. Data cache miss rates measured at CABA level.

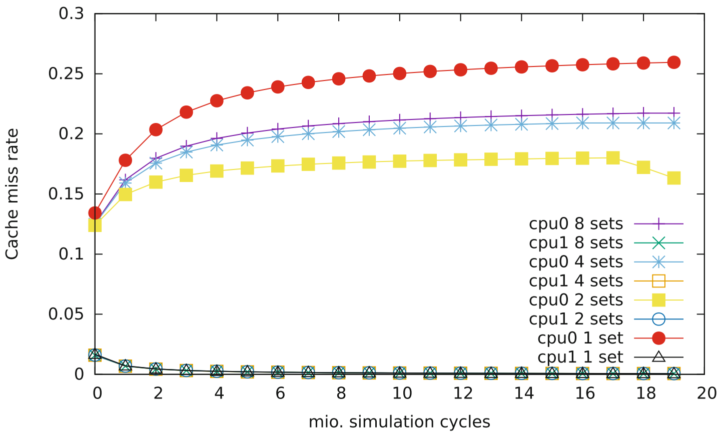

Fig. 17. Instruction cache miss rates measured at CABA level.

We vary associativity of both caches for the same cache size. Figure 14 shows the processor parameters of the Deployment Diagram. Parameters are the degree of associativity (*instruction/data cache ways*, in the Figure set to 2), the number of lines in the cache (*instruction/data cache sets*, here set to 16) and the number of words in a line (*instruction/data cache words*, here 4). Figure 16 shows the

data cache miss rates, and Fig. 17 shows instruction cache miss rates for set associativities of 1, 2, 4 and 8, using the same overall cache size, and same block size.

We observe clearly that from two cache sets onwards, cache misses are divided by two. The improvement is still around 30% for the instruction cache; we also note that CPU1 is much less challenged, as already shown for the CPI. In the worst case of a direct mapped cache, we have an instruction cache miss rate of 25% on the cache of CPU0, less than 1% for CPU1 (which essentially contains the interface and has a very small memory footprint). Thus, we can provide significantly more useful detail for a hardware implementation.

A first exploration presented in [2] for a different case study showed that cache misses can only be imprecisely estimated at the DIPLODOCUS model. However, that case study lacked the detailed modeling that we present here, both at the DIPLODOCUS and the AVATAR level. The Smart Card case study remedies this shortcoming.

We can now go back to the DIPLODOCUS level and customize the CPU by adapting the cache miss rate (Fig. 4): we were thus able to check that the partitioning result is still the same.

Latencies. In previous work [2], performance results were limited to those obtained using the hardware counters of the SoCLib modules. A recent update to TTool added support for automatically measuring latencies for channel transfers/between events during simulation. Activity elements can be marked as potential checkpoints on the model.

Events and channels in DIPLODOCUS both translate into signals in AVATAR. The left side of Fig. 18 shows a DIPLODOCUS activity diagram for the *Application* task, with two checkpoints set, one on the *open* and the *send_TCP* event. On the right side, it shows the timed automaton of the *Application* block. Again, we place one checkpoint on the *open* and another on the *send_TCP* signal.

The latencies panel of DIPLODOCUS is shown in Fig. 19. Our toolkit allows the user to choose which checkpoints he or she would like to analyze, and then displays the minimum, maximum, and average latencies in execution cycles between those two checkpoints.

On the MPSoC prototype for which the code is generated from AVATAR, latencies can be determined by hardware counters added to the SoCLib models. These counters allow identification of the processor that triggered the transfer, but not on which of the communication channels it took place. A recent improvement integrated the SoCLib logging mechanism presented in [34]. Thus, the MPSoC platform is enhanced with spies that can record all transfers on the interconnect, retrieve the names of software objects from the loader, and match them to the steps of the channel access protocol. This module is added to the VCI interface and does not impact performance results. Thus, we can now measure latencies on the MPSoC platform that are due to contentions on the interconnect, to the time spent waiting to obtain a lock, etc.

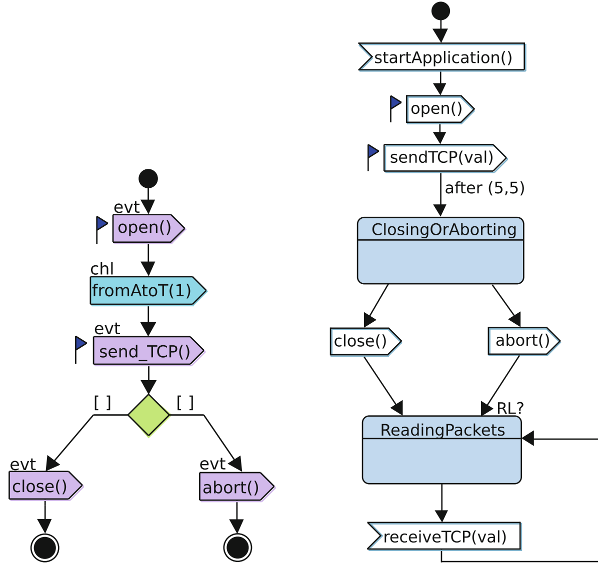

Fig. 18. Latency checkpoints (left) DIPLODOCUS (right) AVATAR.

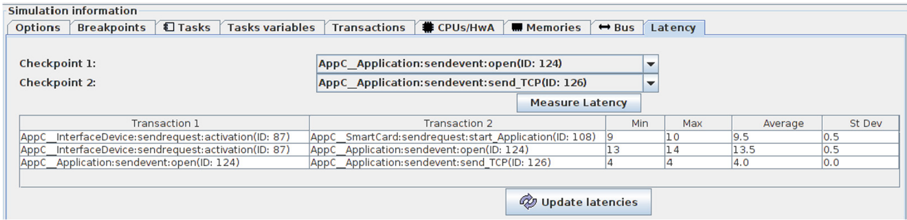

Fig. 19. DIPLODOCUS latencies panel.

Table 1 shows the latencies for selected channels corresponding to four signals in AVATAR. While *reset*, *start* and *open* signals have no parameters, fromTtoP conveys a packet (eight bytes in the case study).

Table 1. Latencies (milliseconds) for DIPLODOCUS simulation and SoCLib prototype.

AVATAR signal	DIPLODOCUS		MPSoC	
	Min	Max	Min	Max
`SmartCardController_reset__InterfaceDevice_reset`	2	2	0.56	0.64
`SmartCardController_start_Application__Application_startApplication`	4	4	0.56	0.58
`Application_open__TCPIP_open`	4	4	0.56	0.59
`SmartCardController_fromTtoP__TCPIP_fromTtoP`	38	75	1.6	1.7

As these first results show, there is no apparent correlation between the latency measured on the MPSoC platform and the latency obtained by DIPLODOCUS simulation. In fact for code generated from AVATAR running multiprocessor platform, cache effects, contention on the interconnect and others have to be taken into account. In particular, the storing and retrieving time of packets varies strongly. We are currently working on establishing correlations where this is possible, together with even more in-depth performance evaluation. It would be important to extend the latency measurement capability to AVATAR simulations, which should relate more closely to tests in SoCLib.

Other Performance Metrics. As can be seen in the CPU attributes window of Fig. 4, our toolkit potentially allows a designer to improve estimates of several more hardware parameters like branch misprediction rate and go idle time. Channels play a particular role: for asynchronous channels, they may overflow or otherwise be empty most of the time, slowing down or even blocking the application. Dimensioning of the channels is thus an important issue. Better understanding of the state of communication channels (fill state, evolution of read and write operations over time etc.) will be achieved by integrating new performance measuring functionality based on the work described in [32].

7 Discussion and Future Work

Our approach integrates both system-level design space exploration and the design and prototyping of refined software components in the same toolkit. Using a Smart Card case study, we show how the different metrics can easily be evaluated at the push of a button in the two abstraction levels. In particular, transformations of the software component model mapped onto a deployment diagram help precisely determine the CPI, as well as the finer metrics as cache miss rate and latencies of the application. From these evaluations, partitioning choices can be confirmed or invalidated.

We are currently working on a more complete method to determine and compare performance metrics in particular latencies at the AVATAR and DIPLODOCUS level and hope to establish correlations. Relating partitioning and software level simulations may also help us determine the accuracy of the estimates of execution duration of functions in partitioning.

The close integration of partitioning and software design facilitates the invalidation of partitioning decisions. The current backtracing to models assists the engineer in investigating how to better partition the model or to reconsider the software components. Ideally, once an invalidation has been encountered, it would be helpful for the toolkit to automatically suggest another partitioning. We propose increased automation as part of our future work, to better support designers between the different stages of the design process.

References

1. Apvrille, L.: Webpage of TTool (2015). http://ttool.telecom-paristech.fr/
2. Genius, D., Li, L.W., Apvrille, L.: Model-driven performance evaluation and formal verification for multi-level embedded system design. In: Conference on Model-Driven Engineering and Software Development (Modelsward 2017), Porto, Portugal (2017)
3. Buck, J., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: a framework for simulating and prototyping heterogeneous systems. In: Readings in Hardware/Software Co-design, pp. 527–543 (2002)
4. Ptolemaeus, C.: System Design, Modeling, and Simulation: Using Ptolemy II. Ptolemy.org, Berkeley (2014)
5. Kim, H., Guo, L., Lee, E.A., Sangiovanni-Vincentelli, A.: A tool integration approach for architectural exploration of aircraft electric power systems. In: IEEE Proceedings of the 1st International Conference on Cyber-Physical Systems, Networks, and Applications, pp. 38–43. IEEE (2013)
6. Zimmermann, J., Stattelmann, S., Viehl, A., Bringmann, O., Rosenstiel, W.: Model-driven virtual prototyping for real-time simulation of distributed embedded systems. In: SIES, pp. 201–210. IEEE (2012)
7. Roth, C., Bucher, H., Reder, S., Buciuman, F., Sander, O., Becker, J.: A SystemC modeling and simulation methodology for fast and accurate parallel MPSoC simulation. In: 2013 26th Symposium on Integrated Circuits and Systems Design (SBCCI), pp. 1–6. IEEE (2013)
8. Real, M.M., Wehner, P., Rettkowski, J., Migliore, V., Lapotre, V., Göhringer, D., Gogniat, G.: MPSoCSim extension: an OVP simulator for the evaluation of cluster-based multi and many-core architectures. In: Proceedings of the 4th Workshop on Virtual Prototyping of Parallel and Embedded Systems (ViPES) as Part of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XVI), Samos, Greece (2016)
9. Buchmann, R., Greiner, A.: A fully static scheduling approach for fast cycle accurate SystemC simulation of MPSoCs. In: Proceedings of the ICEEC, Cairo, Egypt, pp. 35–39. IEEE (2007)
10. Polarsys: ARCADIA/CAPELLA (2008). https://www.polarsys.org/capella/arcadia.html
11. Lieverse, P., van der Wolf, P., Vissers, K.A., Deprettere, E.F.: A methodology for architecture exploration of heterogeneous signal processing systems. VLSI Signal Process. **29**, 197–207 (2001)
12. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: an integrated electronic system design environment. IEEE Comput. **36**, 45–52 (2003)
13. Erbas, C., Cerav-Erbas, S., Pimentel, A.D.: Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. IEEE Trans. Evol. Comput. **10**, 358–374 (2006)
14. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information Processing 1974: Proceedings of the IFIP Congress, pp. 471–475. North-Holland, New York (1974)
15. Pimentel, A.D., Hertzberger, L.O., Lieverse, P., van der Wolf, P., Deprettere, E.F.: Exploring embedded-systems architectures with Artemis. IEEE Comput. **34**, 57–63 (2001)

16. Kienhuis, B., Deprettere, E.F., van der Wolf, P., Vissers, K.: A methodology to design programmable embedded systems. In: Deprettere, E.F., Teich, J., Vassiliadis, S. (eds.) SAMOS 2001. LNCS, vol. 2268, pp. 18–37. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45874-3_2
17. Vidal, J., de Lamotte, F., Gogniat, G., Soulard, P., Diguet, J.P.: A co-design approach for embedded system modeling and code generation with UML and MARTE. In: DATE 2009, pp. 226–231 (2009)
18. Gamatié, A., Beux, S.L., Piel, É., Atitallah, R.B., Etien, A., Marquet, P., Dekeyser, J.L.: A model-driven design framework for massively parallel embedded systems. ACM Trans. Embed. Comput. Syst **10**, 39 (2011)
19. Saxena, T., Karsai, G.: MDE-based approach for generalizing design space exploration. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 46–60. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_4
20. Gérard, S., Espinoza, H., Terrier, F., Selic, B.: 6 modeling languages for real-time and embedded systems. In: Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (eds.) MBEERTS 2007. LNCS, vol. 6100, pp. 129–154. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16277-0_6
21. IBM Corporation: Rational Rhapsody. https://www.ibm.com/us-en/marketplace/rational-rhapsody
22. Sodius Corporation: MDGen for SystemC. http://sodius.com/products-overview/systemc
23. Feiler, P.H., Lewis, B.A., Vestal, S., Colbert, E.: An overview of the SAE architecture analysis & design language (AADL) standard: a basis for model-based architecture-driven embedded systems engineering. In: Dissaux, P., Filali-Amine, M., Michel, P., Vernadat, F. (eds.) IFIP WCC TC2 2004. IFIP The International Federation for Information Processing, vol. 176, pp. 3–15. Springer, Boston (2004). https://doi.org/10.1007/0-387-24590-1_1
24. Yu, H., Joshi, P., Talpin, J.P., Shukla, S.K., Shiraishi, S.: The challenge of interoperability: model-based integration for automotive control software. In: DAC, pp. 58:1–58:6. ACM (2015)
25. Bombieri, N., Fummi, F., Vinco, S., Quaglia, D.: Automatic interface generation for component reuse in HW-SW partitioning. In: 2011 14th Euromicro Conference on Digital System Design, pp. 793–796 (2011)
26. Batori, G., Theisz, Z., Asztalos, D.: Domain specific modeling methodology for reconfigurable networked systems. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 316–330. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_22
27. Genius, D., Apvrille, L.: Virtual yet precise prototyping: an automotive case study. In: ERTSS 2016, Toulouse (2016)
28. Genius, D., Apvrille, L.: System-level design for communication-centric task farm applications. In: 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip, pp. 1–8. IEEE (2017). https://ieeexplore.ieee.org/document/8016145/
29. Schweppe, H., Roudier, Y., Weyl, B., Apvrille, L., Scheuermann, D.: C2x communication: securing the last meter. In: The 4th IEEE International Symposium on Wireless Vehicular Communications, WIVEC 2011, San Francisco, USA (2011)
30. SoCLib Consortium: SoCLib: an open platform for virtual prototyping of multi-processors system on chip. http://www.soclib.fr (2010)
31. VSI Alliance: Virtual component interface standard (OCB 2 2.0). Technical report, VSI Alliance (2000)

32. Genius, D., Pouillon, N.: Monitoring communication channels on a shared memory multi-processor system on chip. In: ReCoSoC, pp. 1–8. IEEE (2011)
33. Genius, D., Faure, E., Pouillon, N.: Mapping a telecommunication application on a multiprocessor system-on-chip. In: Gogniat, G., Milojevic, D., Morawiec, A., Erdogan, A. (eds.) Algorithm-Architecture Matching for Signal and Image Processing. LNEE, vol. 73, pp. 53–77. Springer, Dordrecht (2011). https://doi.org/10.1007/978-90-481-9965-5_3
34. Genius, D.: Measuring memory access latency for software objects in a NUMA system-on-chip architecture. In: ReCoSoC, pp. 1–8. IEEE (2013)
35. Knorreck, D., Apvrille, L., Pacalet, R.: Formal system-level design space exploration. Concurr. Comput.: Pract. Exp. **25**, 250–264 (2013)
36. Enrici, A., Apvrille, L., Pacalet, R.: A model-driven engineering methodology to design parallel and distributed embedded systems. ACM Trans. Des. Autom. Electron. Syst. **22**, 34:1–34:25 (2017)
37. Pedroza, G., Knorreck, D., Apvrille, L.: AVATAR: a SysML environment for the formal verification of safety and security properties. In: The 11th IEEE Conference on Distributed Systems and New Technologies (NOTERE 2011), Paris, France (2011)
38. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
39. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Proceedings of the 14th IEEE Workshop on Computer Security Foundations, CSFW 2001, Washington, D.C., USA, p. 82. IEEE Computer Society (2001)
40. Li, L.W., Lugou, F., Apvrille, L.: Security-aware modeling and analysis for HW/SW partitioning. In: Conference on Model-Driven Engineering and Software Development (Modelsward 2017), Porto, Portugal (2017)
41. Lugou, F., Li, L.W., Apvrille, L., Ameur-Boulifa, R.: SysML models and model transformation for security. In: Conference on Model-Driven Engineering and Software Development (Modelsward 2016), Rome, Italy (2016)
42. Etienne Faure: Communications matérielles-logicielles dans les systèmes sur puce orientés télécommunications (HW/SW communications in telecommunication oriented MPSoC). Ph.D. thesis, UPMC (2007)
43. Becoulet, A.: MutekH. http://www.mutekh.org