Model-Driven Performance Evaluation and Formal Verification for Multi-level Embedded System Design

Daniela Genius, Letitia Li, Ludovic Apvrille

Model-Driven Performance Evaluation and Formal Verification for Multi-level Embedded System Design

Daniela Genius[1], Letitia W. Li[2], Ludovic Apvrille[2]

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

Abstract: The design methodology of an embedded system should start with a system-level partitioning dividing functions into hardware and software. However, since this partitioning decision is taken at a high level of abstraction, we propose regularly validating the selected partitioning during software development. The paper introduces a new model-based engineering process with a supporting toolkit, first performing system-level partitioning, and then assessing the partitioning choices thus obtained at different levels of abstraction during software design. This assessment shall in particular validate the assumptions made on system-level (e.g. on cache miss rates) that cannot be precisely determined without low-level hardware model. High-level partitioning simulations/verification rely on custom model-checkers and abstract models of software and hardware, while low-level prototyping simulations rely on automatically generated C-POSIX software code executing on a cycle-precise virtual prototyping platform. An automotive case study on an automatic braking application illustrates our complete approach.

1 Introduction

Embedded systems are composed of a tightly integrated ensemble of HW and SW components. The design of these systems usually starts with a system-level partitioning phase, continues with separate software and hardware design, and finishes with a HW/SW integration. In fact, this integration can also be performed progressively during software design using prototyping techniques.

At system-level partitioning, properties of embedded applications can be tested rapidly and their description remains somewhat "human-readable". Properties can also be proven formally. Exploration, however, remains at a rather abstract level e.g. many hardware parameters are approximated. For example, the cache miss rate is modeled as a fixed value (e.g., 5%) obtained from the architect's experience.

After partitioning, software is designed at lower abstraction levels. Commonly, the hardware target is not available, leading to the use of simulation techniques with precise hardware models. System parameters, such as the cache miss ratio, can be closely evaluated with simulations and formal proofs. Unfortunately, more details also means much slower simulations, and infeasible formal proofs, even if com-

positional approaches could help handle entire hardware platforms. However, they are costly (Basu et al., 2011; Syed-Alwi et al., 2013) in terms of development time.

To improve both development stages (partitioning, prototyping), we propose to unify them in a common SysML formalism. In fact, prototyping can rely on software and hardware elements that were formally evaluated at partitioning. Partitioning models can be enhanced using precise parameters that can be obtained during simulation at the prototyping level. Our toolkit, TTool (Apvrille, 2015), supports both stages, and makes it possible, at the push of a button, to evaluate the design at a given development stage, and to propagate the results to enhance the system at another development state, thus easing development iterations. We previously described (Li et al., 2016) our approach towards multi-level Design Space Exploration, but without the ability to generate detailed performance metrics during prototyping that we present in this paper.

Section 2 presents related work. Section 3 presents the overall design method. Section 4 details an automotive case study used to exemplify the high-level design space exploration (Section 5), as well as software component design and performance evalua-

tion (Section 6). A final discussion and perspectives on future work are presented in Section 7.

2 System-level Design for Embedded Systems

A number of system-level design tools exist, offering a variety of verification and simulation capabilities at different levels of abstraction.

Ptolemy (Buck et al., 2002) 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.

Metropolis (Balarin et al., 2003) targets heterogeneous systems, and architectural and application constraints are closely interwoven. 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 (Erbas et al., 2006) 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 (Kahn, 1974)) 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 (Pimentel et al., 2001) project is strongly based on the Y-chart approach. Application and architecture are clearly separated: the application produces an event trace at simulation time, which is read by the architecture model. However, behavior depending on timers and interrupts cannot be taken into account.

MARTE (Vidal et al., 2009) 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.

Other works based on UML/MARTE, such as Gaspard2 (Gamatié et al., 2011), 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.

Rhapsody can automatically generate software, but not hardware descriptions from SysML. MDGen from Sodius (Sodius Corporation, 2016) 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 (Feiler et al., 2004) allows the use of formal methods for safety-critical real-time 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, (Yu et al., 2015) developed a model-based formal integration framework which endows AADL with a language for expressing timing relationships.

Capella (Polarsys, 2008) 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.

3 Methodology

3.1 Modeling Phases

Our approach combines partitioning - the partitioning decision relies on design space exploration techniques - and software design. The latter includes the prototyping of the designed software. All stages are supported within the same SysML-based free and open-source environment/toolkit (as shown in Figure 1):

1. The overall method starts with a partitioning phase containing three sub-phases: the modeling of the functions to be realized by the system (functional view), the modeling of the candidate architecture as an assembly of highly abstracted hardware nodes, and the mapping phase. A function mapped on a processor is a software function, a function mapped on a hardware accelerator corresponds to a custom ASIC (Application-specific Integrated Circuit).

2. Once the system is fully partitioned, the second phase starts with the design of the software and the hardware. Our approach offers software modeling while taking into account hardware parame-
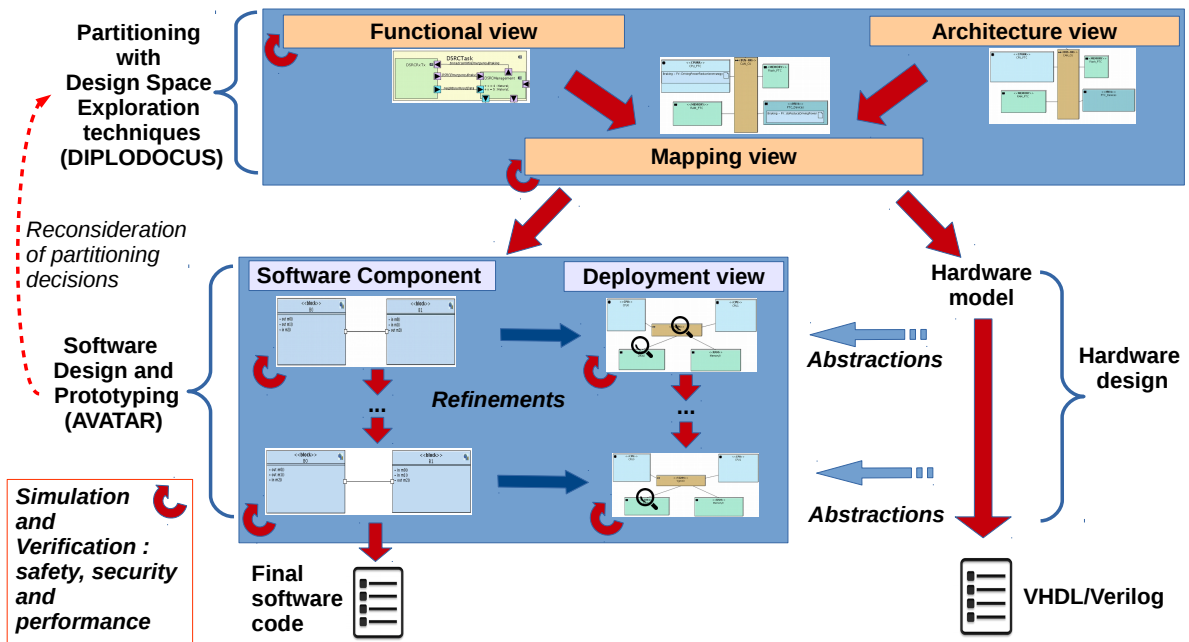
Figure 1: Overall Approach

ters for prototyping purposes. Thus, a deployment view displays how the software components are allocated to the hardware components. Code can then be generated both for the software components of the application (in C/POSIX code) and for the virtual hardware nodes (in SoCLib (So-CLib consortium, 2010) System C format).

Choice of parameters on the higher level is subject to validation or invalidation due to experimental results on the generated prototype. Thus, simulations results at prototyping level could lead to reconsider the partitioning decisions.

3.2 Simulation, Verification and Prototyping

During the methodological phases, simulation and formal verification help in deciding whether safety, performance and security requirements are fulfilled. Our toolkit 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 utilities. Backtracing to models is then performed to better inform the users about the verification results. Proofs of safety involve UPPAAL semantics (Bengtsson and Yi., 2004), and security proofs use ProVerif (Blanchet, 2010). Before the next stage, simulation and formal verification ensure that our design meets performance, behavioral, and schedulabil-

ity requirements. Simulation of partitioning specifications involves executing tasks on the different hardware elements in a transactional high-level way. Each transaction executes for a variable time depending on execution cycles and CPU parameters. The simulation shows performance results like bus usage, CPU usage, execution time, etc., so as to help users decide on an architecture and mapping. For example, singles execution sequences can be investigated with *gtkwave*. Also, our toolkit 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.

During functional modeling, verification intends to identify general safety properties (e.g., absence of deadlock situations). At the mapping stage, verification intends to ascertain if performance and security requirements are met. Hardware components are highly abstracted. For example, a CPU can be defined with a set of parameters such as an average cache-miss ratio, power-saving mode activation, context switch penalty, etc.

After mapping, software components can also be verified independently of 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. When the soft-
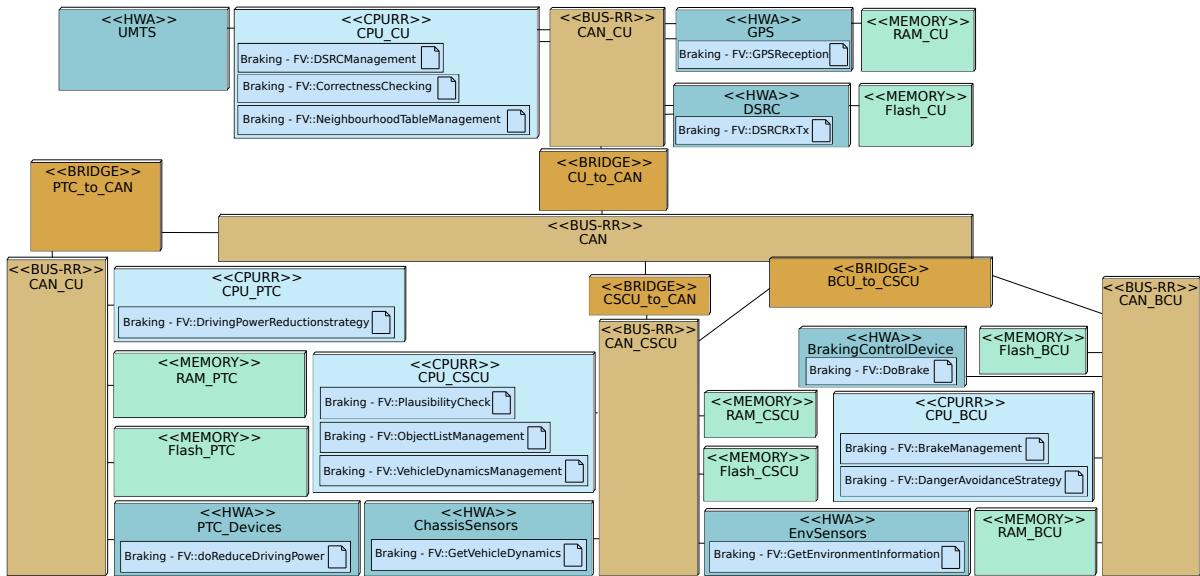
Figure 2: Automotive Case Study Architecture Diagram

ware components become more refined, it becomes important to evaluate their performance when executed on the target platform. Since the target system is commonly not yet available, our approach offers two facilities: a Deployment Diagram in which software components can be mapped over hardware nodes (see Figure 4), and a press-button approach to transform this Deployment Diagram into a specification built upon virtual component models. For this, we use *SoCLib*, a public domain library of component models written in SystemC. SoCLib targets shared-memory *multiprocessor-on-chip system* (MP-SoC) architectures based on the *Virtual Component Interconnect* (VCI) protocol (VSI Alliance, 2000) which separates the components' functionality from communication. Hardware is described at several abstraction levels: TLM (Transaction level), CABA (Cycle/Bit Accurate), and RTL (Register Transfer Level). SoCLib also contains a set of performance evaluation tools (Genius et al., 2011). Last but not least, the SoCLib prototyping platform comes with an operating system well adapted to multiprocessor-on-chip (Becoulet, 2009).

If the performance results of the SystemC simulation differ too greatly from the ones obtained during the design space exploration stage – e.g., a cache miss ratio – then, design space exploration shall be performed again to assess if the selected architecture is still the best according to the system requirements. If not, 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 are finished, software code can be

generated from the most refined software model.

4 Automotive Case Study

Our methodology is illustrated using an automotive embedded system designed in the scope of the European EVITA project (EVITA, 2011). Recent on-board Intelligent Transport (IT) architectures comprise a very heterogeneous landscape of communication network technologies (e.g., LIN, CAN, MOST, and FlexRay) that interconnect in-car Electronic Control Units (ECUs).

The increasing number of such equipment triggers the development of novel applications that are commonly spread among several ECUs to fulfill their goals. Prototyping on multiprocessor architectures, even if they are more generic than the final hardware, is thus very useful.

An automatic braking application serves as a case study (Kelling et al., 2009). The system works essentially as follows: an obstacle is detected by another automotive system which broadcasts that information to neighboring cars. A car receiving such information has to decide if it is concerned with this obstacle. This decision includes a plausibility check function that takes into account various parameters, such as the direction and speed of the car, and also information previously received from neighboring cars. Once the decision to brake has been taken, the braking order is forwarded to relevant ECUs. Also, the presence of this obstacle is forwarded to other neighboring cars in case they have not yet received this information.
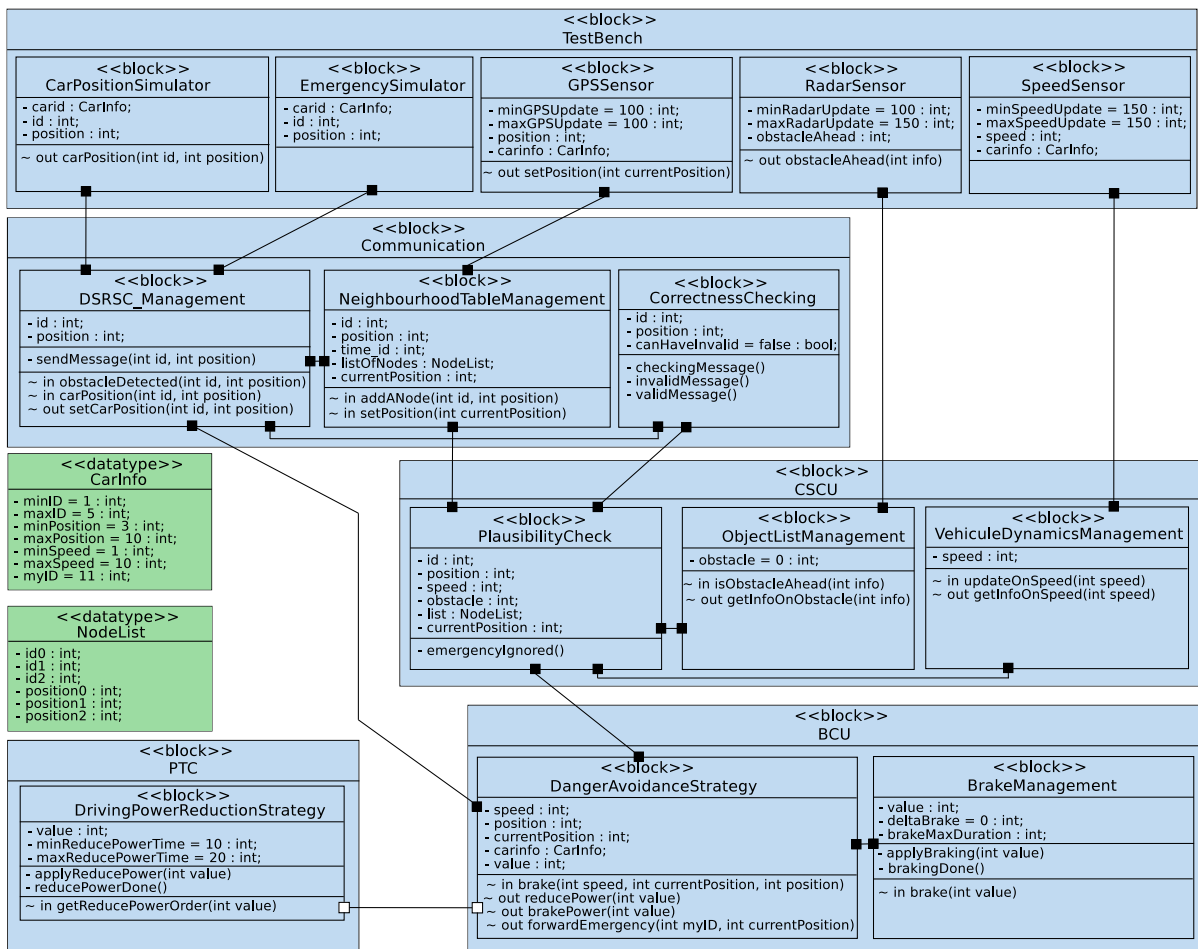
<<block>>
TestBench

<<block>>
CarPositionSimulator
- carid : CarInfo;
- id : int;
- position : int;
~ out carPosition(int id, int position)

<<block>>
EmergencySimulator
- carid : CarInfo;
- id : int;
- position : int;

<<block>>
GPSSensor
- minGPSUpdate = 100 : int;
- maxGPSUpdate = 100 : int;
- position : int;
- carinfo : CarInfo;
~ out setPosition(int currentPosition)

<<block>>
RadarSensor
- minRadarUpdate = 100 : int;
- maxRadarUpdate = 150 : int;
- obstacleAhead : int;
~ out obstacleAhead(int info)

<<block>>
SpeedSensor
- minSpeedUpdate = 150 : int;
- maxSpeedUpdate = 150 : int;
- speed : int;
- carinfo : CarInfo;

<<block>>
Communication

<<block>>
DSRSC_Management
- id : int;
- position : int;
- sendMessage(int id, int position)
~ in obstacleDetected(int id, int position)
~ in carPosition(int id, int position)
~ out setCarPosition(int id, int position)

<<block>>
NeighbourhoodTableManagement
- id : int;
- position : int;
- time_id : int;
- listOfNodes : NodeList;
- currentPosition : int;
~ in addANode(int id, int position)
~ in setPosition(int currentPosition)

<<block>>
CorrectnessChecking
- id : int;
- position : int;
- canHaveInvalid = false : bool;
- checkingMessage()
- invalidMessage()
- validMessage()

<<datatype>>
CarInfo
- minID = 1 : int;
- maxID = 5 : int;
- minPosition = 3 : int;
- maxPosition = 10 : int;
- minSpeed = 1 : int;
- maxSpeed = 10 : int;
- myID = 11 : int;

<<datatype>>
NodeList
- id0 : int;
- id1 : int;
- id2 : int;
- position0 : int;
- position1 : int;
- position2 : int;

<<block>>
CSCU

<<block>>
PlausibilityCheck
- id : int;
- position : int;
- speed : int;
- obstacle : int;
- list : NodeList;
- currentPosition : int;
- emergencyIgnored()

<<block>>
ObjectListManagement
- obstacle = 0 : int;
~ in isObstacleAhead(int info)
~ out getInfoOnObstacle(int info)

<<block>>
VehiculeDynamicsManagement
- speed : int;
~ in updateOnSpeed(int speed)
~ out getInfoOnSpeed(int speed)

<<block>>
PTC

<<block>>
DrivingPowerReductionStrategy
- value : int;
- minReducePowerTime = 10 : int;
- maxReducePowerTime = 20 : int;
- applyReducePower(int value)
- reducePowerDone()
~ in getReducePowerOrder(int value)

<<block>>
BCU

<<block>>
DangerAvoidanceStrategy
- speed : int;
- position : int;
- currentPosition : int;
- carinfo : CarInfo;
- value : int;
~ in brake(int speed, int currentPosition, int position)
~ out reducePower(int value)
~ out brakePower(int value)
~ out forwardEmergency(int myID, int currentPosition)

<<block>>
BrakeManagement
- value : int;
- deltaBrake = 0 : int;
- brakeMaxDuration : int;
- applyBraking(int value)
- brakingDone()
~ in brake(int value)

Figure 3: Active Braking Block Diagram

5 Hardware/Software Partitioning

5.1 Modeling

The stages of the methodology include Partitioning by Design Space Exploration, Software Design, and Prototyping, with different models at each stage. Figure 2 shows the model for Partitioning: an Architecture Diagram with the tasks divided onto different CPUs and Hardware Accelerators. Figure 3 shows the Block Diagram for Software Design. Figure 4 shows the Deployment Diagram. We elaborate in detail on the different stages in the following sections.

The HW/SW Partitioning phase of our methodology intends to model the abstract, high-level functionality of a system (Knorreck et al., 2013). It follows the Y-chart approach, first modeling the abstract functional tasks, candidate architectures, and then finally mapping tasks to the hardware components (Kienhuis et al., 2002). The application is modeled as a set of communicating tasks on the Component Design Diagram (an extension of the SysML Block Instance Diagram). Task behavior is modeled using communication operators, computation elements, and control elements.

The architectural modeling (Figure 2) is displayed as a graph of execution nodes, communication nodes, and storage nodes. Execution nodes, such as CPUs and Hardware Accelerators, include parameters such data size, instruction execution time, and clock ratio (see Figure 5. CPUs also must be defined by task switching time, cache-miss percentage, etc. Communication nodes include bridges and buses. Buses connect execution and storage nodes, and bridges connect buses. Buses are defined by parameters such as 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.
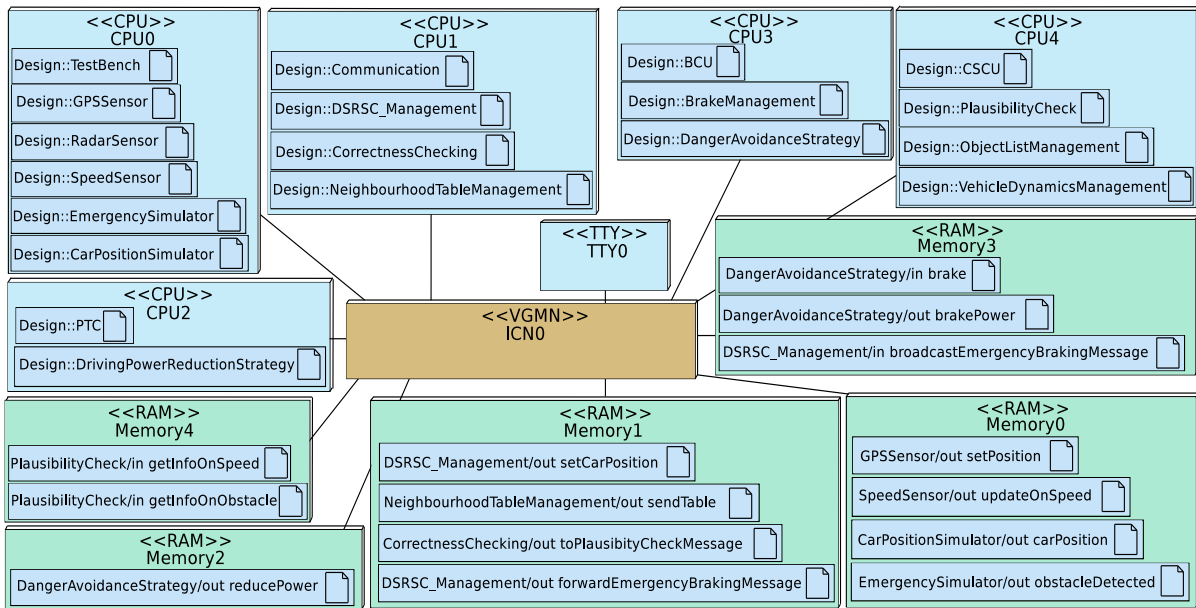
Figure 4: Deployment Diagram of the Active Braking Application: five CPUs and five RAMs

Mapping involves specifying the location of tasks 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 include mapping channels to buses and bridges. Alternatively, if the data path is complex (e.g., DMA transfer), channels can be mapped over communication patterns (Enrici et al., 2014).

Figure 5: Adapting architecture parameters during partitioning

5.2 High-Level Simulation

Using simulation techniques described in section 3.2, we can see that the mapping of tasks of our case study (see Figure 2) ensures that the maximum latency between the decision (*DangerAvoidanceStrategy*) and the resulting actions (*doReduceDrivingPower* and *DoBrake*) respect safety requirements. Similarly, we have verified that the worst latency between the reception of an emergency message by *DRSCManagement* and the consequent actions (e.g., *DoBrake*) is always also below the specified limit. These performance verifications are performed according to the selected functions, operating systems and hardware components. In particular, many parameters of the hardware components are simple values (we have for example selected a cache-miss ratio of 5%) that are meant to be confirmed during the software design phase.

6 Software Design with AVATAR/SoCLib

Once the partitioning is complete, the AVATAR methodology (Pedroza et al., 2011) allows the user to design the software, perform functional simulation and formal verification, and finally test the software components in a virtual prototyping environment.

6.1 Software Components

Figure 3 shows the software components of the active braking use case modeled using an AVATAR block diagram. These modeling elements have been selected during the previous modeling stage (partitioning). Software components are grouped according to their destination ECU:

- **Communication ECU** manages communication with neighboring vehicles.
- **Chassis Safety Controller ECU (CSCU)** processes emergency messages and sends orders to brake to ECUs.
- **Braking Controller ECU (BCU)** contains two blocks: *DangerAvoidanceStrategy* determines how to efficiently and safely reduce the vehicle speed, or brake if necessary. *BrakeManager* operates the brake for a given duration.
- **Power Train Controller ECU (PTC)** enforces the engine torque modification request.

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 6 shows the state machine of *DangerAvoidanceStrategy*, Figure 8 shows a visualization of the generated sequence diagram.
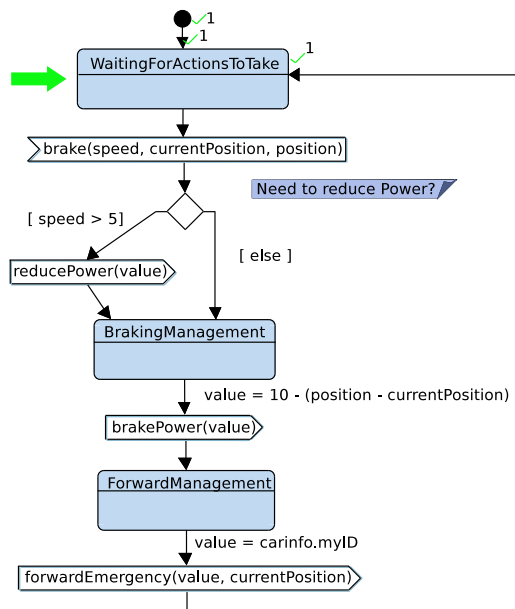
Figure 6: High Level Simulation of the Active Braking Automotive system: State Machine

We show traces for the *CarPositionSimulator* block and for three of the blocks which interact in an emergency braking situation: *DrivingPowerReductionStrategy*, *DangerAvoidanceStrategy* and *BrakeManagement*.

6.2 Formal Verification

During formal verification of safety properties with UPPAAL, 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 state 'Plausibility Check' is always executed after a neighboring car signals that it has detected an obstacle. We can also verify that an order to brake can be received, or state 'Braking Management' in Task 'Danger Avoidance Strategy' is reachable. Figure 7 shows the UPPAAL verification window which allows the user to customize which queries to execute, and then returns the results as shown.
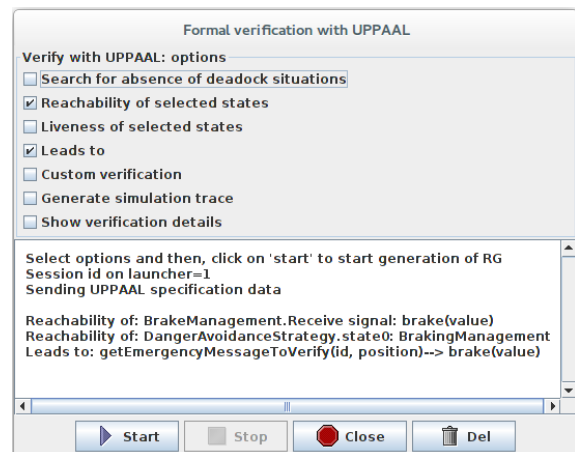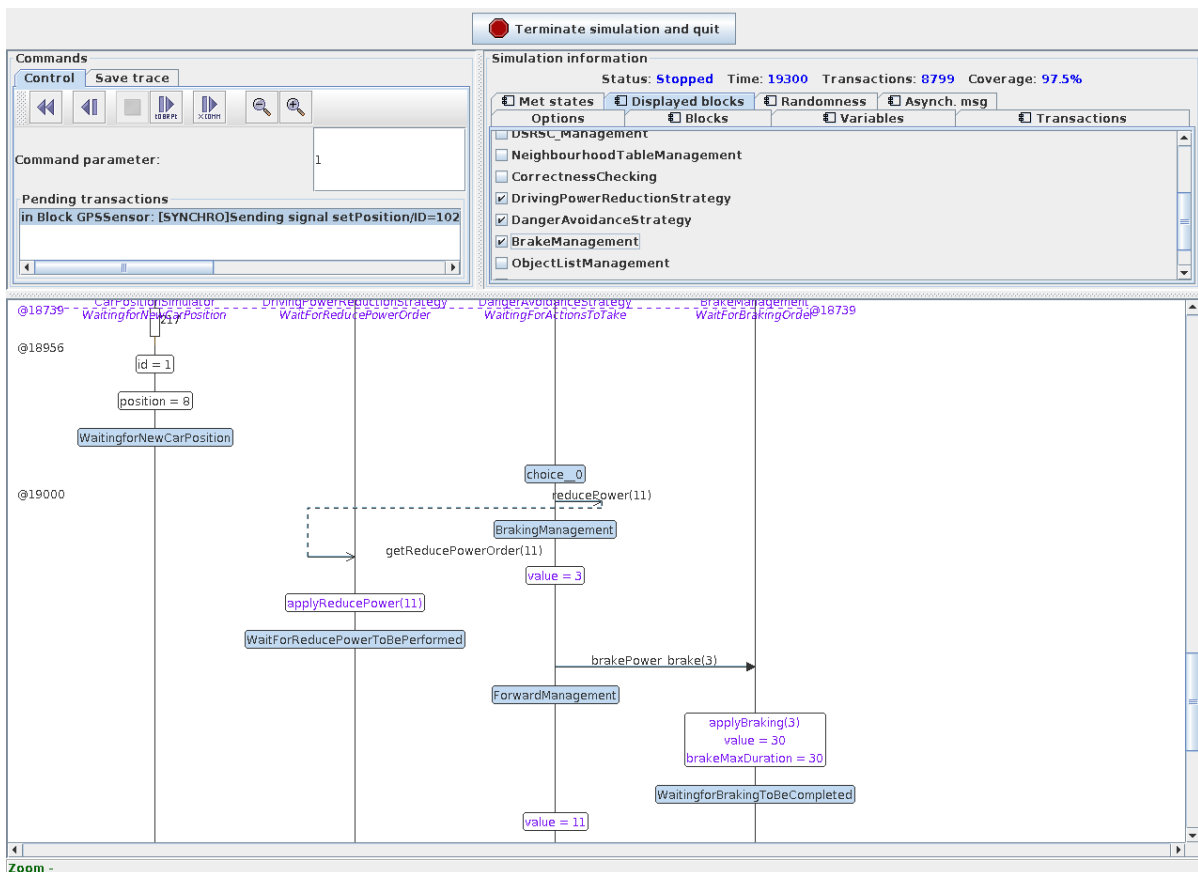

Figure 7: UPPAAL Formal Verification

Figure 8: High Level Simulation of the Active Braking Automotive system: generated Sequence diagram

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 recently introduced in (Genius and Apvrille, 2016). An AVATAR **Deployment Diagram** is used for that purpose. It features a set of hardware components, their interconnection, tasks, and channels.

The partitioning phase selected an architecture with five clusters. Some tasks are destined to be software tasks (they are mapped onto CPUs), and the others are expected to be realized as hardware accelerators. Yet, each specific hardware accelerator in SoCLib needs to be developed specifically which requires a significant effort. We do not consider that case in the paper since all AVATAR tasks are software tasks. The five clusters are represented by five CPUs and the channels between AVATAR tasks are implemented as software channels mapped to on-chip RAM.

Some properties pertaining to mapping must be explicitly captured in the Deployment Diagram, such as CPUs, memories and their parameters, while others, such as simulation infrastructure and interrupt management, are added transparently to the top cell during the transformation to SoCLib. Figure 4 shows the Deployment Diagram of the software components of the active braking application mapped on five processors and five memory elements. From the Deployment Diagram, a SoCLib prototype is then generated. 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 (Figure 9).

6.4 Capturing Performance Information

We now present how performance information can be obtained from the use case simulated with SoCLib. In the experiments shown here, we use PowerPC cores. The cycle accurate bit accurate (CABA)-level simulation allows measurement of cache miss rates, la-
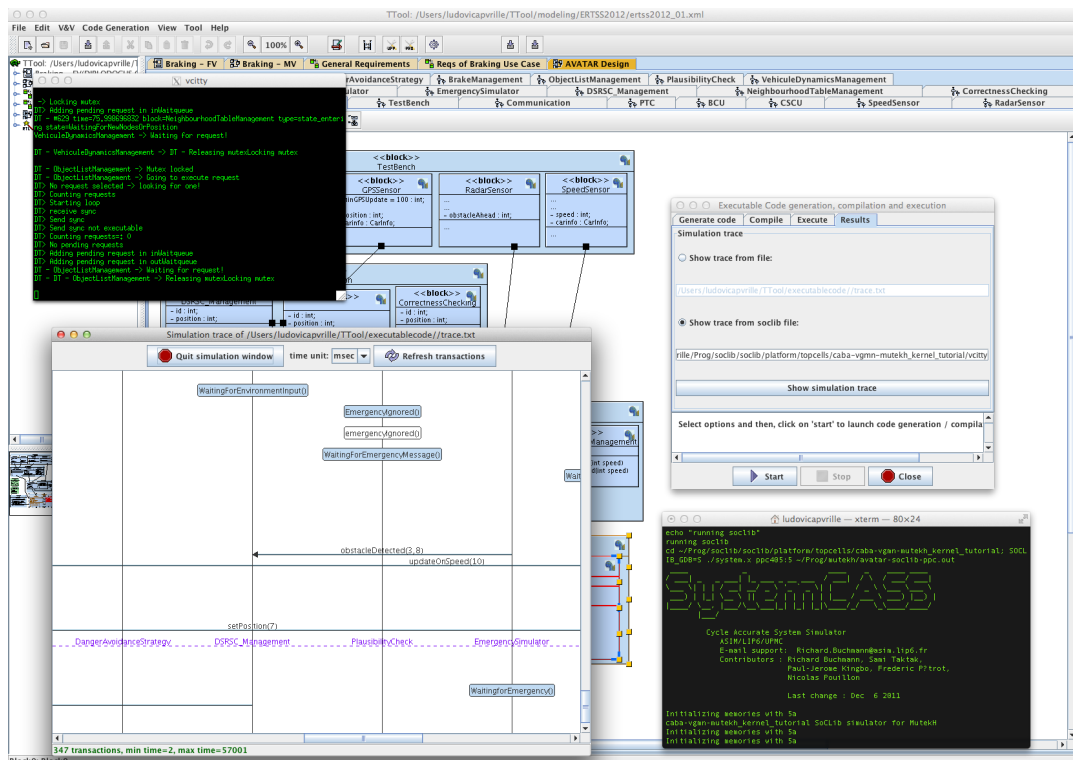
Figure 9: AVATAR/SoCLib Prototyping Environment in TTool

tency of any transaction on the interconnect, taking/releasing of locks, etc. Since SoCLib hardware models are much more precise than the ones used at the design space exploration level, precise timing and hardware mechanisms can be evaluated. However, these evaluations take considerable time compared to high-level simulation/evaluation. We restrict ourselves to using only the hardware counters available in the SoCLib cache module.

We start by an overview of performance problems. For this, we use an overall metric summing up all phenomena that slow down execution of instructions by the processor, such as memory access latency, interconnect contention, overhead due to context switching etc.: *Cycles per Instruction (CPI)*. For bottom line comparison, the CPI is first measured on a mono processor platform (Figure 10). On this platform, the single processor is constantly overloaded (CPI > 16).

Our tool allows per-processor performance evaluation, which is particularly useful in detecting unbalanced CPU loads. Even when prototyping onto five processors (Figure 11) to reflect the DIPLODOCUS partitioning, the CPU loads are not very well balanced. This is due to the fact that currently, a central request manager is required to capture the semantics of AVATAR channels. Requests are stored in waiting queues for synchronous as well as asyn-
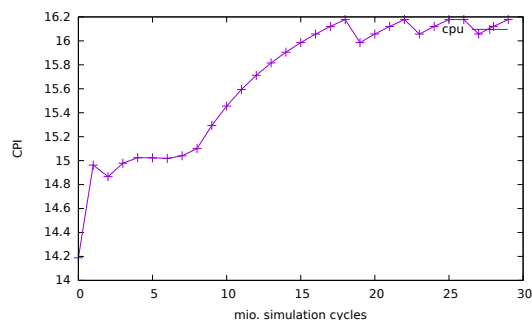

Figure 10: CPI per processor for a mono processor configuration

chronous communication, and, in synchronous communications, cancelled when they became obsolete. CPU0 frequently needs to access memory areas storing the boot sequence and the central request manager. Future work will address a better distribution of these functionalities, called the *AVATAR runtime*, over the MPSoC architecture. Another interesting observation is that in the five processor configuration, CPU4 is more strongly challenged than the others. Looking at the AVATAR block diagram, it becomes clear that the CSCU, mapped on CPU4, is connected by AVATAR channels to all the other ECUs.

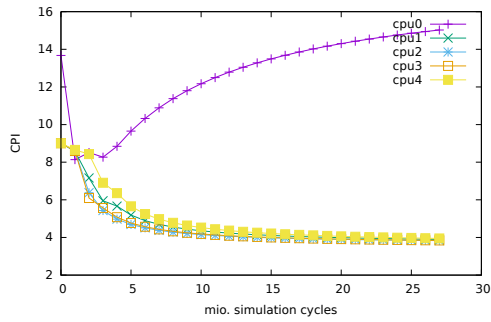We now investigate the cache miss rate. One

Figure 11: CPI per processor for a 5 processor configuration

important parameter of the CPU used in the DIPLODOCUS partitioning is the overall cache miss rate (see line *Cache-miss* in Figure 5). While the estimated 5% of cache misses includes both data and instruction cache misses, SoCLib measures 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.

We vary size and associativity of both caches, initially considering direct mapped caches (Figure 13), then setting associativity to four (Figure 14) for the same size. This action can be performed with a few mouse clicks (see Figure 12).
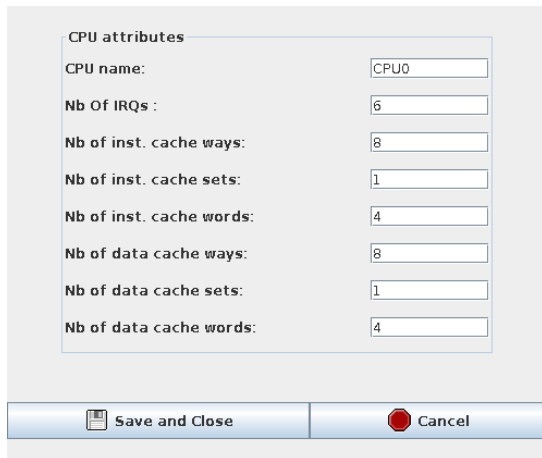
Figure 12: Varying cache associativity with a few mouse clicks

For the instruction cache, using the same parameters (Figures 15 and 16), miss rates are closer to the estimated ones.

Even if we do not explore the cache parameters fully in the work presented here, we can already conclude from this first exploration that data cache misses were overestimated; they are below 10^{-7}. As for instruction cache misses, they are below 10% for the cache of CPU0, below 2% for the other four caches.
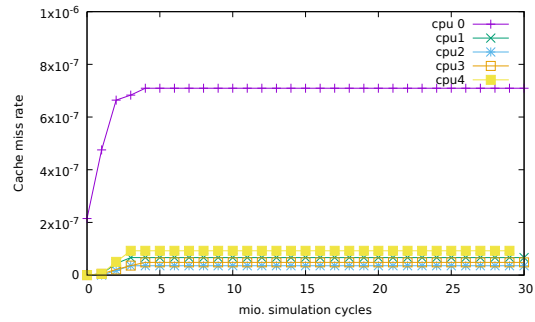

Figure 13: Data cache misses per processor for a 5 processor configuration with a direct mapped cache
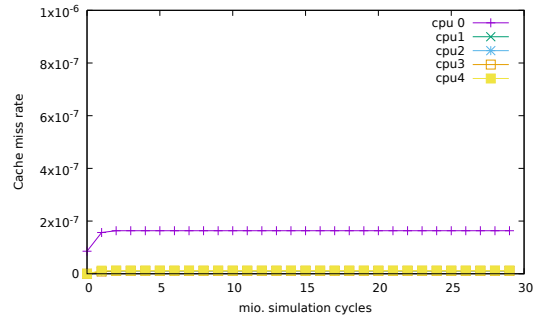

Figure 14: Data cache misses per processor for a 5 processor configuration with 4 cache sets

We can thus lower the estimations, distinguishing between CPU0 and the others. Since our toolkit does not distinguish between data and instruction cache misses during partitioning, we take the less favorable case of instruction cache misses and raise the miss rate for CPU0 to 10%, and lower it to 2% for the others. Figure 5 shows the window for customizing the CPU during partitioning, where we can now adapt the cache miss rate (and redo the partitioning).

We finally compare the influence of the interconnect latency (10 and 20 cycles, see Figures 17 and 18). We observe a significant influence on the cost of a cache miss; latency of data cache misses is generally
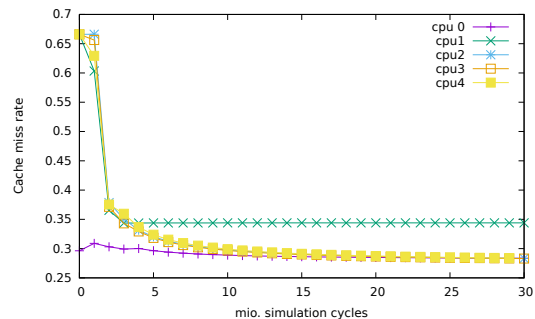

Figure 15: Instruction cache misses per processor for a 5 processor configuration with a direct mapped cache
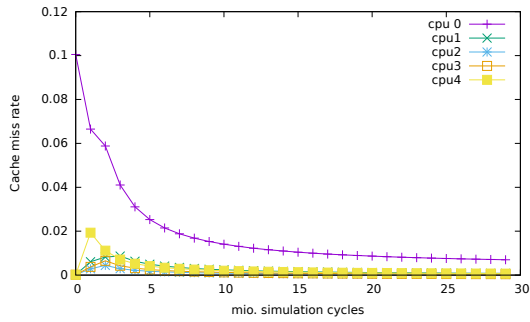
Figure 16: Instruction cache misses per processor for a 5 processor configuration with 4 cache sets

Figure 18: Cost of data cache miss in cycles for a 5 processor configuration with 4 cache sets

higher.

We observe after these first exploration steps that apart from correcting the estimated cache miss rate in DIPLODOCUS, adding another CPU in order to take some of the load from CPU4 would improve the performance.

As we can see in the CPU attributes window of Figure 5, our toolkit potentially allows a designer to improve estimates of several more hardware parameters like branch misprediction rate and go idle time. Until now, we used only the hardware counters implemented in the SoCLib components. Taking into account the OS, over which we have full control, we will soon be able to address other issues such as task switching time.
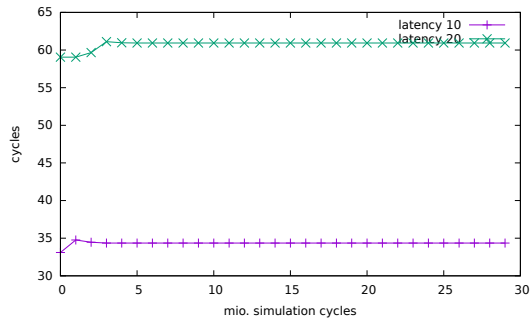

Figure 17: Cost of instruction cache miss in cycles for a 5 processor configuration with 4 cache sets

7 Discussion and Future Work

Our model-driven approach with a SysML-based methodology and supporting toolkit enables designers to capture systems at multiple levels and facilitates the transitions between embedded system design stages. Prototyping from AVATAR enables the user to take into account performance results in a few clicks in the
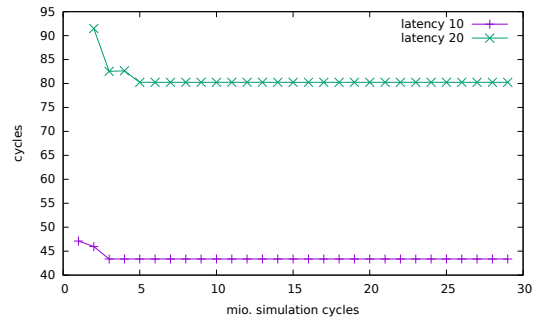
Deployment Diagram, though the process is not yet fully automated.

In order to deliver more realistic results, we are currently working on integrating clustered architectures. These architectures are supported in So-Clib, but various details make top cell generation much more complex (two-level mapping table, address computation complexity, etc.).

To help backtrace low level results (prototyping) to a higher level (partitioning), we are currently working on providing the performance graphs shown in the paper directly and automatically in the toolkit. Also, most metrics we have exemplified are CABA-based. We could also propose two other abstraction levels of SoCLib: TLM (Transaction Level) and TLM-T (Transaction Level with Time). Future work will focus on adding these intermediate levels, considerably speeding up prototypes at the cost of loss of precision to be evaluated. However, using this intermediate level of abstraction would smooth the development gap between system-level and low-level prototyping.

REFERENCES

Apvrille, L. (2015). Webpage of TTool. In *http://ttool.telecom-paristech.fr/*.

Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., and Sangiovanni-Vincentelli, A. L. (2003). Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52.

Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., and Sifakis, J. (2011). Rigorous component-based system design using the BIP framework.

Becoulet, A. (2009). Mutekh operating system (webpage). http://www.mutekh.org.

Bengtsson, J. and Yi, W. (2004). Timed automata:

Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*, pages 87–124. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag.

Blanchet, B. (2010). Proverif automatic cryptographic protocol verifier user manual. Technical report, CNRS, Département d'Informatique École Normale Supérieure, Paris.

Buck, J., Ha, S., Lee, E. A., and Messerschmitt, D. G. (2002). Ptolemy: a framework for simulating and prototyping heterogeneous systems. *Readings in hardware/software co-design*, pages 527–543.

Enrici, A., Apvrille, L., and Pacalet, R. (2014). A uml model-driven approach to efficiently allocate complex communication schemes. In *MODELS conference*, Valencia, Spain.

Erbas, C., Cerav-Erbas, S., and Pimentel, A. D. (2006). Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Transactions on Evolutionary Computation*, 10(3):358–374.

EVITA (2011). E-safety Vehicle InTrusion protected Applications. http://www.evita-project.org/.

Feiler, P. H., Lewis, B. A., Vestal, S., and Colbert, E. (2004). 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., and Vernadat, F., editors, *IFIP-WADL*, volume 176 of *IFIP*, pages 3–15. Springer.

Gamatié, A., Beux, S. L., Piel, É., Atitallah, R. B., Etien, A., Marquet, P., and Dekeyser, J.-L. (2011). A model-driven design framework for massively parallel embedded systems. *ACM Trans. Embedded Comput. Syst*, 10(4):39.

Genius, D. and Apvrille, L. (2016). Virtual yet precise prototyping : An automotive case study. In *ERTSS'2016*, Toulouse.

Genius, D., Faure, E., and Pouillon, N. (2011). Mapping a telecommunication application on a multiprocessor system-on-chip. In Gogniat, G., Milojevic, D., and Erdogan, A. M. A. A., editors, *Algorithm-Architecture Matching for Signal and Image Processing*, chapter 1, pages 53–77. Springer LNEE vol. 73.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In Rosenfeld, J. L., editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY.

Kelling, E., Friedewald, M., Leimbach, T., Menzel, M., Sieger, P., Seudié, H., and Weyl, B. (2009). Specification and evaluation of e-security relevant use cases. Technical Report Deliverable D2.1, EVITA Project.

Kienhuis, B., Deprettere, E., van der Wolf, P., and Vissers, K. (2002). A Methodology to Design Programmable Embedded Systems: The Y-Chart Approach. In *Embedded Processor Design Challenges*, pages 18–37. Springer.

Knorreck, D., Apvrille, L., and Pacalet, R. (2013). Formal System-level Design Space Exploration. *Concurrency and Computation: Practice and Experience*, 25(2):250–264.

Li, L., Apvrille, L., and Genius, D. (2016). Virtual prototyping of automotive systems: Towards multi-level design space exploration. In *Conference on Design and Architectures for Signal and Image Processing*.

Pedroza, G., Knorreck, D., and Apvrille, L. (2011). 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.

Pimentel, A. D., Hertzberger, L. O., Lieverse, P., van der Wolf, P., and Deprettere, E. F. (2001). Exploring embedded-systems architectures with artemis. *IEEE Computer*, 34(11):57–63.

Polarsys (2008). ARCADIA/CAPELLA (webpage).

SoCLib consortium (2010). SoCLib: an open platform for virtual prototyping of multi-processors system on chip (webpage). In *http://www.soclib.fr*.

Sodius Corporation (2016). MDGen for SystemC. http://sodius.com/products-overview/systemc.

Syed-Alwi, S.-H., Braunstein, C., and Encrenaz, E. (2013). *Efficient Refinement Strategy Exploiting Component Properties in a CEGAR Process*, volume 265 of *Lecture Notes in Electrical Engineerin g*, chapter 2, pages 17–36. Springer.

Vidal, J., de Lamotte, F., Gogniat, G., Soulard, P., and Diguet, J.-P. (2009). A co-design approach for embedded system modeling and code generation with UML and MARTE. In *DATE'09*, pages 226–231.

VSI Alliance (2000). Virtual Component Interface Standard (OCB 2 2.0). Technical report, VSI Alliance.

Yu, H., Joshi, P., Talpin, J.-P., Shukla, S. K., and Shiraishi, S. (2015). The challenge of interoperability: model-based integration for automotive control software. In *DAC*, pages 58:1–58:6. ACM.