# Virtual Prototyping of Automotive Systems: Towards Multi-level Design Space Exploration

Letitia W. Li*, Ludovic Apvrille*, Daniela Genius†
* Télécom ParisTech, Université Paris-Saclay, Institut Mines-Telecom
Email: letitia.li,ludovic.apvrille@telecom-paristech.fr
† Sorbonne Universités, UPMC Paris 06, LIP6, CNRS UMR 7606,
Email: daniela.genius@lip6.fr

***Abstract*—The design methodology of an embedded system should start with a system-level design space exploration 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 TTool, first performing system-level design space exploration, and then assessing these partitioning choices at different levels of abstraction during software design. Exploration and partitioning choices are verified using a press-button approach, enabling simulation and formal verification directly from SysML models. High-level simulations/verification rely on custom model-checkers and abstract models of software and hardware, while low-level 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.**

***Index Terms*—Virtual prototyping, Embedded systems, System-level design, Automotive engineering**

## I. Introduction

A challenge for adaptive, intelligent motor vehicles was formulated by DARPA. In Europe as well, manufacturers became aware that nearly 90% of the road accidents are caused by human error, and 52% due to a collision. To cope with this fact, automated braking systems were developed.

Control elements of such systems have become increasingly complex [3], pushing current system development techniques to their limits. One of the limitations is the lack of integration between the design space exploration of the system, and the design of the software components itself. Since models of software components are generally tested/executed on the local host, and only later integrated once the target is available, errors due to the interaction between hardware and software are discovered very late in the development cycle - e.g., during the integration phase - that may lead to reconsideration of partitioning decisions taken during the design space exploration phase.

Thus, it would be useful to develop a fully integrated method to model critical software components, candidate hardware architectures, and then to evaluate the execution of the former onto the latter using automated model transformation techniques at a low-level of abstraction. Our contribution presents an easy-to-comprehend methodology integrating desgin space exploration and software development, with all stages contained within a single modeling framework TTool. Section II presents the related work, Section III the overall method. Section IV details an automotive case study then used to exemplify the high-level design space exploration (Section V) and the software components design (Section VI). Finally, discussion and perspectives on future work are presented in Section VII.

## II. 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.

Capella [15] 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.

Metropolis [2] 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 [4] 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 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 [14] project originates from heterogeneous platforms in the context of research on multimedia applications in particular. It 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 then read in by the architecture model. However, behavior depending on timers and interrupts cannot be taken into account.

MARTE [18] 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 [7], 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.

MDGen from Sodius [17] starts from Rhapsody, which can automatically generate software, but not hardware descriptions from SysML. SysML in Rhapsody is untimed and sequential. Also, timing and hardware specific artifacts such as clock/reset lines are generated automatically.

The Architecture Analysis & Design Language (AADL [6]) 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, [20] developed a model-based formal integration framework which endows AADL with a language for expressing timing relationships.

## III. Methodology

### A. Modeling Phases

Our approach combines design space exploration and software design that follows in the same environment/toolkit (as shown in Figure 1). The method is organized as follows:

1) The method starts with design space exploration. This phase contains three sub-phases: the modeling of the functions to be realized by the system (functional view), the candidate architecture expressed as an assembly of highly abstracted hardware nodes, and the mapping phase. 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.
2) Once a mapping has been decided, i.e., the system is fully partitioned between software and

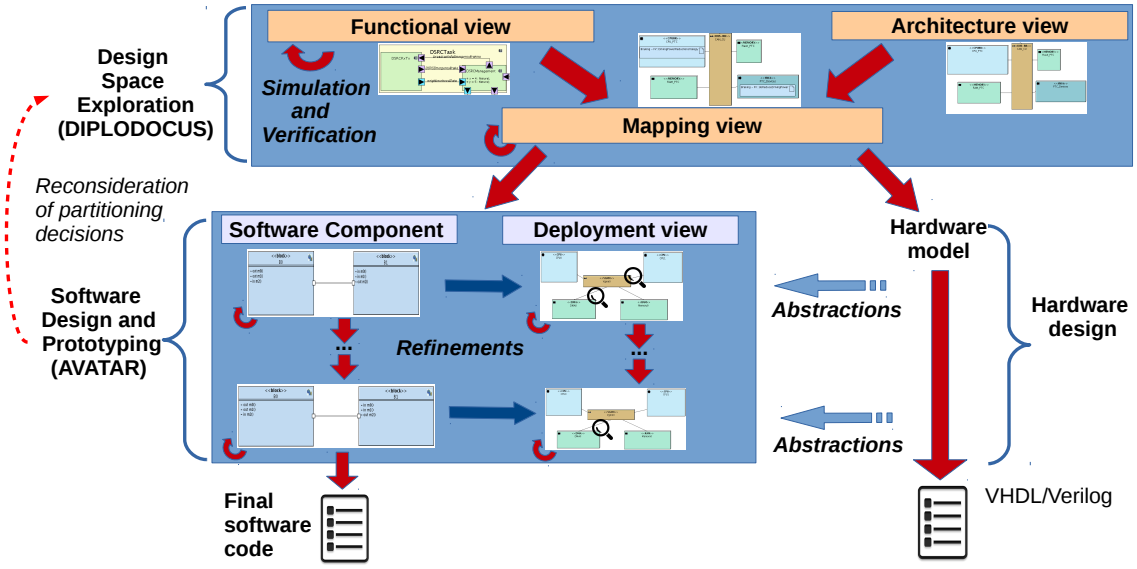


Fig. 1. Overall Approach

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 deployment view displays how the software components relate to the hardware components.

TTool [1], a free and open-source toolkit, supports the entire method with SysML diagrams.

### B. Simulation, Verification and Prototyping

During the methodological phases, simulation and formal verification help in deciding whether 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. Backtracing to models is then performed to better inform the users about the verification results. During functional modeling, verification intends to identify general safety properties (e.g., absence of deadlock situations). At the mapping stage, verification intends to verify 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.

Software components can also 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. 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: a deployment diagram in which software components can be mapped over hardware nodes (see 4), and a press-button approach to transform the deployment diagram into a SoCLib specification built upon virtual component models [16]. SoCLib is a public domain library of models written in SystemC, targeting shared-memory architectures based on the *Virtual Component Interconnect* protocol [19]. Hardware is described at several abstraction levels:

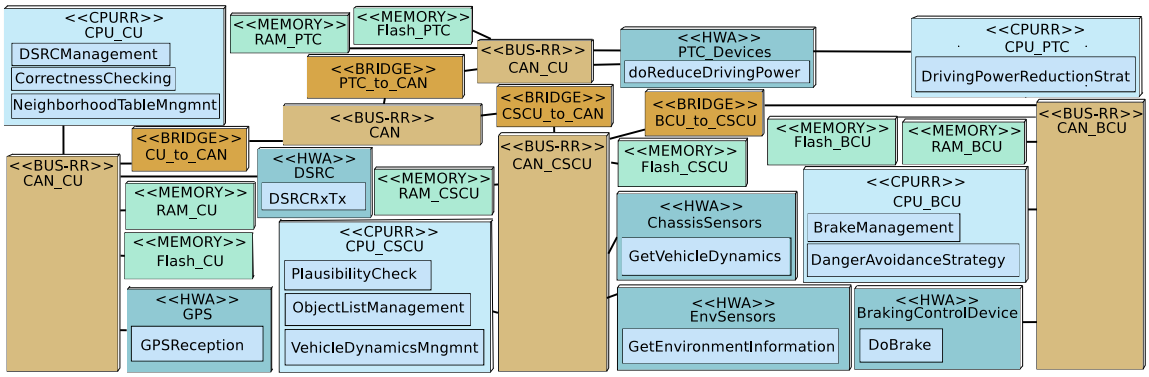


Fig. 2. Automotive Case Study Architecture Diagram in DIPLODOCUS

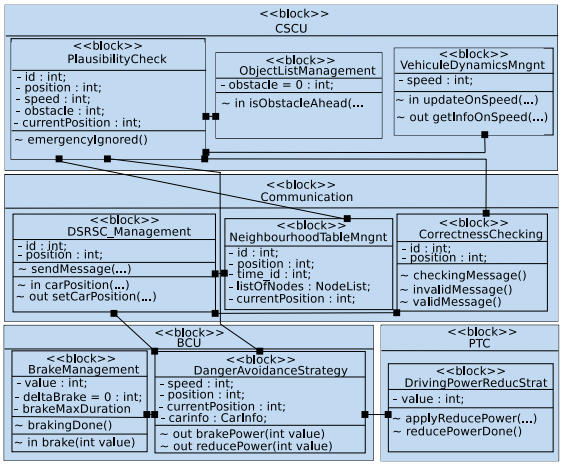


Fig. 3. Active Braking Block Diagram

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 [9], [10]. CABA level simulation potentially allows us to measure cache miss rates, latency of memory accesses and of any transactions on the interconnect, fill state of the buffers, taking/releasing of locks etc.

Since SoCLib hardware models are much more precise than the ones used at design space exploration level, precise timing and hardware mechanisms - e.g. cache miss - can be evaluated. If the performance results differ too greatly from the ones 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 decided 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.

## IV. Automotive Case Study

Our methodology is illustrated by an automotive embedded system designed in the scope of the European EVITA project [5]. 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.

An automatic braking application serves as a case study [11]. 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

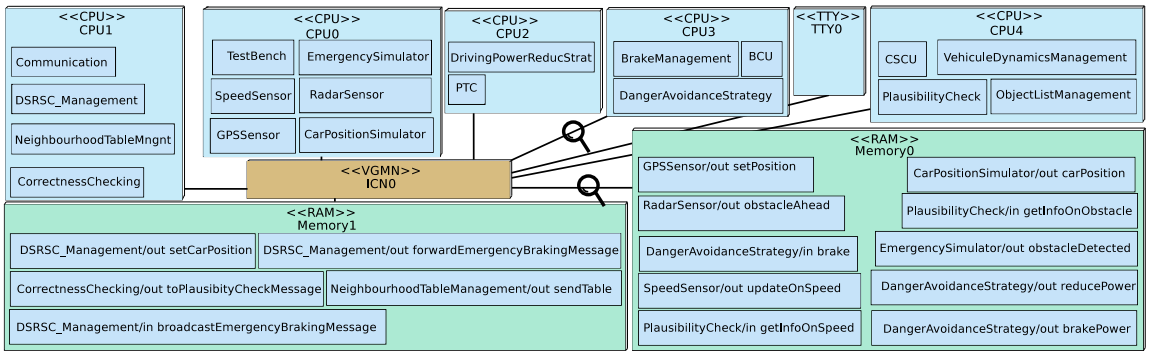


Fig. 4. Deployment Diagram of the Active Braking Application: five CPUs and two RAMs

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. 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: a 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.

## V. Partitioning with DIPLODOCUS

### A. Models

The HW/SW Partitioning phase of SysML-Sec, implemented in the DIPLODOCUS profile of TTool, intends to model the abstract, high-level functionality of a system [13]. It follows the Y-chart approach (as shown in the upper right section of Figure 1), first modeling the abstract functional tasks (Application View), candidate architectures (Architectural View), and finally mapping tasks to the hardware components (Mapping View) [12]. 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.

*1) Application View:* The Application View comprises of a set of communicating tasks. The behavior of tasks is described abstractly. Functional abstraction allows us to ignore the exact calculations and data processing of algorithms, and consider only relative execution time. 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.

*2) Architectural View:* The architectural model (Figure 2) 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. 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.

*3) Mapping View:* Mapping partitions the application into software and hardware as well as specifying the location of their implementation 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.

### B. High-Level Simulation

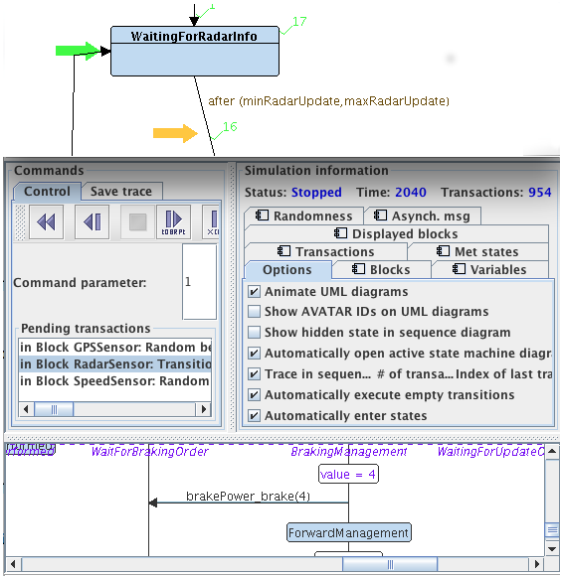


Fig. 5. High Level Simulation of the Active Braking Automotive system

Simulation of DIPLODOCUS partitioning specifications involves executing tasks on the different hardware elements. 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. Users can download a vcd trace to view detailed bus/CPU activity in gtkwave of a single execution sequence. Simulations help users decide on an architecture and mapping. TTool 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. Thus, 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 receving of an emergency message received by *DRSCManagement* and the 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.

## VI. Software design with AVATAR/SoCLib

Once the partitioning is done, the AVATAR methodology allows the user to design the software, perform functional simulation and formal verification, and finally test the software components in a virtual prototyping environment. We will now illustrate this with the case study.

### A. Software Components

Figure 3 represents the software elements of the active braking use case modeled with an AVATAR block diagram. 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.

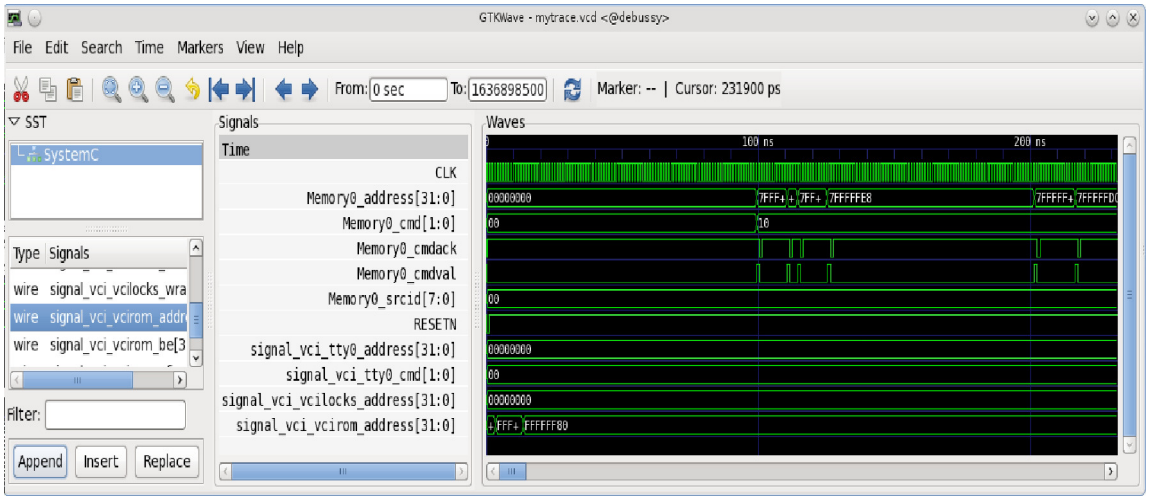


Fig. 6. Active Braking Application: Cycle Accurate Simulation

- **Power Train Controller ECU (PTC)** enforces the engine torque modification request.

The AVATAR model (see Figure 3) can be functionally simulated using the integrated simulator of TTool (see Figure 5). This simulator takes into account temporal operators but completely ignores hardware, operating systems and middleware. While being simulated, the model of the software components is animated.

### *B. 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 [8]. Thus, an AVATAR **deployment diagram** is a SysML representation of hardware components, their interconnection, tasks and channels. Figure 4 shows the deployment diagram of the active braking application on five processors of a MP-SoC (in Figure 2 on five clusters of a CAN) and two memory elements. New extensions to TTool support model transformation from SysML to SoCLib [8]. Some features pertaining to mapping must be explicitly captured in the deployment diagram, such as CPUs and memories, while others, such as simulation infrastructure and interrupt management, are added transparently to the top cell during the transformation to SoCLib.

Figure 6 shows examination of system behavior on the cycle accurate level: the SoCLib top cell can produce a VCD trace of selected signals over time, which can be examined with tools like *gtkwave*. Note that every signal on the VCI interface is detailed on this level, whereas the engineer just places one arc in the SysML diagram and the generated topcell shows one aggregate VCI signal (for example *signal_vci_rom*).

We go one step further by interpreting the sequence of such signals. Accesses to the interconnect are monitored with the help of so-called *hardware spies* which are more precise than simple tracing mechanisms and help reduce the size of the trace. We thus keep track of the simulation cycle; we are also able to identify each software object (for example a channel buffer) by its name by consulting the symbol table. The sequence of read and write operations to the memory location of the software object, the taking and releasing of locks etc. then permits a user to precisely determine operations – for example, that a CPU performs a write access to a channel [10].

Figure 4 shows one possible localization of two

such spy modules on the VCI interface between CPU3 and the interconnect between the interconnect and *Memory0*. The spies are displayed using a small magnifier icon. For example, we wish to monitor the transfers between the *DangerAvoidanceStrategy* block and the *DangerAvoidanceStrategy_brake__PlausibilityCheck_brake* channel. To investigate the cache miss ratio of CPU3, we trace all the read and write accesses on the interconnect between *CPU3* and *ICN*. Similarly, to investigate the latency to the memory *Memory0*, we place a spy between *CPU3* and *Memory0*.

## VII. 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 an automotive case study, we show how different metrics can easily be evaluated at the push of a button. In particular, transformations of the software component model mapped onto a deployment diagram help precisely determine the cache miss of the application. From these evaluations, partitioning choices can be confirmed or invalidated.

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 will implement this increased automation as part of our future work.

## References


[1] L. Apvrille. Webpage of TTool. In *http://ttool.telecom-paristech.fr/*, 2015.

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

[3] M. Broy. Challenges in automotive software engineering. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 33–42. ACM, 2006.

[4] C. Erbas, S. Cerav-Erbas, and A. D. Pimentel. 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, 2006.

[5] EVITA. E-safety Vehicle InTrusion protected Applications. http://www.evita-project.org/.

[6] P. H. Feiler, B. A. Lewis, S. Vestal, and E. Colbert. An overview of the SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In P. Dissaux, M. Filali-Amine, P. Michel, and F. Vernadat, editors, *IFIP-WADL*, volume 176 of *IFIP*, pages 3–15. Springer, 2004.

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

[8] D. Genius and L. Apvrille. Virtual yet precise prototyping : An automotive case study. In *ERTSS'2016*, Toulouse, Jan. 2016.

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

[10] D. Genius and N. Pouillon. Monitoring communication channels on a shared memory multi-processor system on chip. In *ReCoSoC*, pages 1–8. IEEE, 2011.

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

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

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

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

[15] Polarsys. ARCADIA/CAPELLA (webpage). In *https://www.polarsys.org/capella/arcadia.html*, 2008.

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

[17] Sodius Corporation. MDGen for SystemC. http://sodius.com/products-overview/systemc.

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

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

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