Virtual Yet Precise Prototyping: An Automotive Case Study

Daniela Genius, Sorbonne Universités, UPMC Paris 06, LIP6, CNRS UMR 7606,
daniela.genius@lip6.fr

Ludovic Apvrille, LTCI, CNRS, Telecom ParisTech, Université Paris-Saclay, IMT,
ludovic.apvrille@telecom-paristech.fr

Abstract

The paper overviews a joint framework for validating and exploring complex embedded systems. The framework indeed combines AVATAR and SocLib. AVATAR is a Model Driven Engineering approach relying on SysML, and SoCLib is a virtual prototyping platform.

The main contribution lies in the new possibility to map AVATAR SysML blocks onto hardware nodes, within a newly conceived Deployment Diagram, and then to transform the latter into SoCLib models. At the AVATAR level, diagrams can be formally verified and simulated in a functional way only, that is, without considering any underlying hardware execution environment. On the contrary, the SocLib models can be simulated taking into account hardware components in an explicit way with a cycle-accurate bit accurate approach.

An automotive system is used to present the two abstraction levels, and their support by the TTool toolkit.

1. Introduction

The complexity of recent systems pushes current design techniques to their limits. While model-oriented design of complex embedded systems is nowadays a current practice in software development for embedded systems, the hardware aspects of such systems are less frequently designed using this kind of approach. Hardware is described on several abstraction levels, that are rarely represented with graphical modeling formalisms: TLM-DT (Transaction level with distributed time), CABA (Cycle/Bit Accurate), and RTL (Register Transfer Level).

The models of software components of complex embedded systems are generally tested/executed on the local host, and integrated on the target once the latter is available. Even if several virtual prototyping platforms are available for to evaluate the software in a quite realistic way before the target is available purpose, they require to re-model software elements in a different input format from the one used in models or programming languages.

AVATAR is a SysML-based environment to model the software components of complex embedded systems [20]. It is particularly adapted to these systems because (i) it proposes new temporal operators to better describe the temporal constraints of these systems and (ii) its models are formally defined, that is, formal simulations and verifications can be performed from the models. From AVATAR models, it is also possible to generate executable C/POSIX code that can be executed either on the local development platform, or on a target. In a former contribution, we already presented how that code can also be executed in the SoCLib virtual prototyping environment [2]. Yet, the joint use of both AVATAR and SocLib was not well integrated since the description of the hardware execution environment could be done only in a textual form, that is, totally outside of the SysML models.

Thus, we now offer a fully integrated way to model critical software components, to model the candidate support hardware architectures, and then to evaluate the execution of the former onto the latter using automated model-to-soclib transformation techniques. The automation of this process in a toolkit (TTool) paves the way for a easy-to-used model-based approach for design space exploration at a low-level of abstraction (e.g., CABA simulation). Figure 1 highlights the novel contributions among the existing AVATAR framework. These new contributions are displayed in red and within a red dotted line.

Section 2 presents the related work. Section 3 describes our methodology. Section 4 presents the automotive case study. Section 5 explains the details of our approach, while Section 6 discusses its limitations. The conclusions and perspective on future work is finally presented in Section 7.

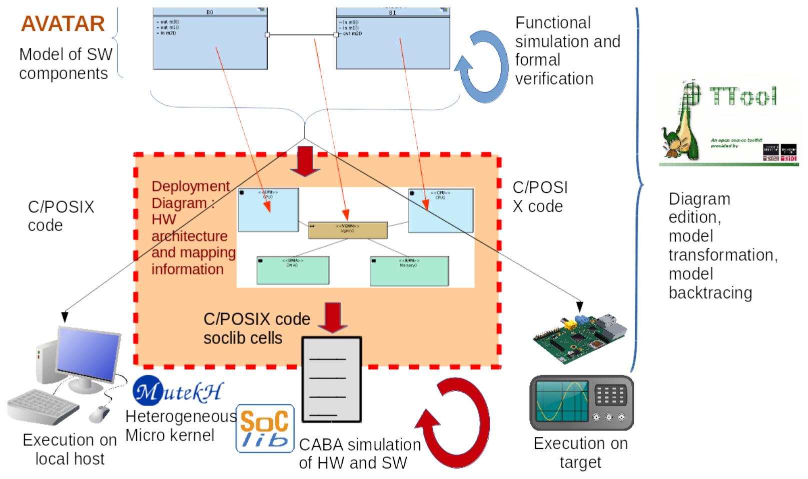

AVATAR
Model of SW components
Functional simulation and formal verification
C/POSIX code
Deployment Diagram : HW architecture and mapping information
C/POSI X code
Diagram edition, model transformation, model backtracing
C/POSIX code soclib cells
MutekH
Heterogeneous Micro kernel
Execution on local host
CABA simulation of HW and SW
Execution on target
TTool

Figure 1. Overview of model transformations for simulation, verification and code generation/execution. Bracktracing to models is not presented, but is effectively implemented in TTTool.

2. Related Work

There are several modeling environments targeting the evaluation/prototyping of functions mapped on complex hardware architectures.

Ptolemy [8] 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.

In Polis [19], 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 [3] is an extension of Polis. It targets heterogeneous systems and offers various execution models. 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).

In SPADE [18], applications are modeled as Kahn processes [16], then mapped to hardware architectural models before being precisely simulated. SPADE is essentially based on RISC processor models.

Sesame [9] proposes modeling and simulation features at several abstraction levels. Preexisting virtual components are combined to form a complex hardware architecture. In contrast to Metropolis, application and architecture are clearly separated in the modeling process. Models' Semantics vary according to the levels of abstraction, ranging from Kahn process networks (KPN) for application modeling, to data flow for model refinement, and to discrete events for simulation purpose. Currently, Sesame is limited to the allocation of processing resources to application processes. It neither models memory mapping nor the choice of the communication architecture.

The ARTEMIS [21] project originates from heterogeneous platforms in the context of Philips research It addresses multimedia applications in particular, thus justifying the acronym (ARchitecTurEs and Methods for embedded MedIa Systems). It is strongly based on the Y-chart approach which has a long tradition at Philips. Simulation is done with SPADE, Sesame or a proprietary Philips tool named TSS (Tool for System Simulation). Application and architecture are clearly separated: the application produces an event trace in a file at simulation time, which is then read in by the architecture model. However, behaviors depending on timers and interrupts cannot be taken into account.

MARTE [26] shares many commonalities with our approach, in terms of the capacity to separately model communications from the pair application-architecture. For such a purpose, MARTE proposes Behavior Scenarios- and Steps (Communication Steps). However, these assets are designed for performance and timing analysis, rather than DSE. Consequently, they intrinsically lack a separation between control aspects and message exchanges as we proposed in Activity and Sequence Diagrams. Even if the UML profile for MARTE adds capabilities to model Real Time and Embedded Systems, it is not specifically targeting architectural exploration: it does not offer any methodology for that purpose, nor selected models, nor model transformation for simulation or formal verification. On the contrary, this is one important goal of our approach.

Other works based on UML/MARTE such as GASPARD [12] 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 communications.

MDGen from Sodius [25] 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. Yet, this approach is probably closest to our present contribution, apart from the lack of hardware description.

The Architecture Analysis & Design Language (AADL [11]) is a standard from the International Society of Automotive engineer (SAE). It 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. It does not a priori contain tool support for code generation. The architecture is modeled in a similar way as we do, e.g., the description of hardware components whose interaction is modeled by connections. Similarly to our environment, a processor model can have different underlying implementations and its characteristics can easily be changed at modeling stage. In the case of our contribution, four components are defined: processors, memories, devices, and buses. The Deployment Diagrams we present in this paper proposes a much larger variety of hardware components. Moreover, the SoClib library, already very rich in detailed models, can easily be enriched by additional components, e.g. with specific coprocessors or other interconnects than a bus, allowing for very detailed simulation with the desired degree of specificity. It does not a priori contain tool support for code generation. An approach that generates a system implementation from models using Simulink has recently been presented [5].

Capella [22] is relying on Arcadia, a comprehensive model-based engineering method. Originating from Thales and widespread in the domains of defense, space and transportation within the company, it provides architecture diagrams allocating functions to components, allocation of Behavioral Components onto Implementation Components (typically hardware, but not necessarily). The basic idea is to check the feasibility of customer requirements, called *needs*, for very large systems. On the contrary, TTool/Soclib is oriented towards co-design for a given case, like the one deonstrated in this paper. As in AVATAR, Capella also provides sequence diagrams and state machines. Capella also provides advanced mechanisms to model bit-precise data structures and relate them to Functional Exchanges, Component or Function Ports, Interfaces, etc. In this sense, it goes further than AVATAR which does not model data structures in such a precise way.

3. Methodology

The methodology of our approach is now better explained (see Figure 1). It can be seen as en extended version with regards to the one presented in [2]. New stages are denoted with a starting "*".

1. **Requirements**. Both safety and security requirements of the system are first captured with SysML Requirements Diagrams.

2. **Design**. The general structure of the software components is modeled with SysML block Diagrams. The behavior of each block is described by a state machine. That behavior can range from a quite abstract, to a more precise one manipulating e.g. data types.

3. **Functional simulation and formal verification**. The press-button approach of TTool makes it possible to perform simulations with model animation. Safety and security formal proofs can also be performed directly from the design models without prior knowledge about underlying formal verification techniques. Safety and security proofs rely on UPPAAL [6] and on ProVerif [7], respectively.

4. **Software code generation**. TTool can generate C/POSIX code from design models. The code can then be compiled and executed either for the localhost or for a given target. This code generation assumes a simple hardware system (One CPU, one memory, etc.).

5. ***Hardware architecture description and mapping**. A deployment diagram can be used to define and interconnect hardware nodes, e.g., processors, memories, buses/interconnects, I/O, interrupts and timers. The execution part of software components can be mapped onto processors. The data part of software components can be mapped onto memory banks.

6. ***Software and hardware code generation**. TTool can now generate the virtual prototyping

code from design and deployment models. The code contains both a SoCLib hardware description of the mapped platform and the software code (C/-POSIX) to be executed on that platform.

7. ***Prototyping with SoCLib**. The SoCLib simulator can be started and the code - generated and compiled during previous step - is loaded and executed like on real hardware. Debugging can be performed at two levels using both the GNU debugger, simulation traces directly displayed by TTool.

Simulation and formal proofs are meant to be executed during the first iterations on the system model. On the contrary, the prototyping of the system is expected to be performed during the last iterations, that is, on more refined models. In all cases (simulation, verification and prototyping), results are directly displayed by TTool in a SysML fashion, therefore facilitating the identification of problems directly on SysML models.

Our environment is built upon two existing modeling and simulation environments: AVATAR, a Model Driven Engineering approach relying on SysML, and SoCLib, a SystemC based virtual prototyping platform.

3.1. AVATAR

The AVATAR environment [20] is a model-oriented solution for the analysis and design of embedded software. AVATAR relies on SysML diagrams to describe the software aspects of the system, as well as its safety and security properties. AVATAR is fully supported by the free software TTool [1]. With TTool, one can edit AVATAR models, simulate or verify them formally in a push-button approach. At last, just like in most UML approaches, simulation and formal verification rely on a purely timed functional model, i.e. without considering any hardware target (CPU, bus, etc.).

TTool implements an AVATAR-to-C/POSIX model transformation as shown in [2]. This code generation permits to validate the software models, but it does not offer any facility for Hardware/Software co-design or design space exploration. Said differently, hardware considerations cannot be captured in the original AVATAR model.

3.2. SoCLib

SoCLib [24] is 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 [27] which separates the components' functionality from communication. SoCLib allows for timed TLM and cycle-accurate bit-accurate (CABA) simulation, so that we have a very detailed level of simulation, using instruction set simulators [23] and modeling cache behavior.

Design space explorations are also addressed in the scope of SoCLib, initially in the context of video streaming and telecommunication applications [14]. Mapping of software objects to memory banks is very fine-grained (stack, lock, buffers can be mapped separately if required), which is a significant improvement over tools like SPADE [18] where only functional tasks can be explicitly mapped.

SoCLib top cells are either hand-written, which is a cumbersome process, or generated from a Python specification, which are complex to describe and debug. Our approach combines both readability and ease of use, taking the latter one step further by proposing SysML/AVATAR as input format.

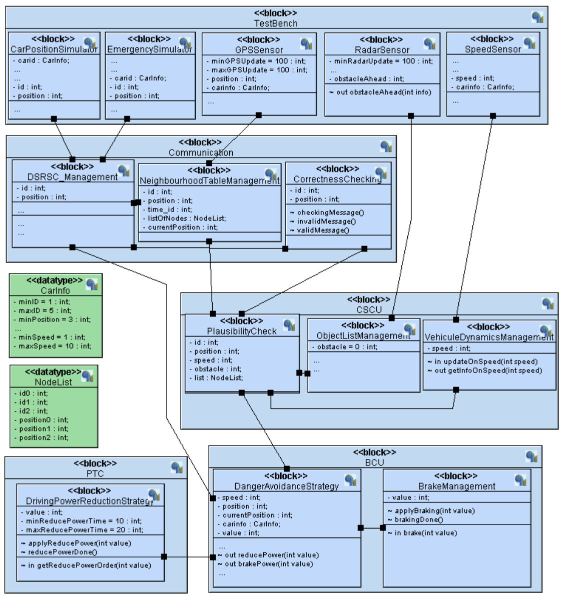


Figure 2. Block diagram of the Active Braking Use Case

4. Automotive Case Study

The AVATAR methodology is illustrated here by the same automotive embedded system designed in the scope of the European EVITA project [10] that was shown in [2]. 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 equipments - sometimes more than a

hundred - triggers the development of novel applications that are commonly spread among several ECUs to fulfill their goals.
The case study, an automatic braking application [17], works basically as follows: an obstacle is detected by another automotive system which broadcasts that information to neighboring cars. A car receiving such an information has to decide whether it is concerned with this obstacle, or not. This verification 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 ECUs responsible for performing the emergency braking. Also, the presence of this obstacle is forwarded to other neighbor cars in case they have not yet received that information. Safety and security requirements were already given in [2].

Figure 2 represents the internal block diagram of the active braking use case. This internal block diagram comprises two kinds of blocks:

- **Blocks dedicated to the modeling of the environment**. They model messages received via wireless connections, data received from sensors, and data output to actuators. For instance, the block *CarPositionSimulator* models the car traffic around the considered automotive system. This car traffic generates location information to the system. The *GPSSensor* regularly records the car position.

- **Blocks dedicated to the modeling of the system itself**. Blocks are grouped within a parent block whose name is the one of the modeled ECU. Basically, the system model contains four ECUs: **Communication ECU** (receiving information, broadcasting information), **Chassis Safety Controller ECU (CSCU)**, **Braking Controller ECU (BCU)**, and **Power Train Controller ECU (PTC)**.

The *DrivingPowerReductionStrategy* block, part of the **Power Train Controller ECU (PTC)**, is not too complex to be shown on the limited space of this paper, and will serve as a running example. For our current experimentation, the security-related functions specified in the original application are left out, but not the safety ones, obviously. Also, the simulation and formal verification aspects have been presented in [2], which explains that the present paper focuses only on prototyping aspects.

5. Contribution

5.1. The AVATAR deployment diagram

Again, our main contribution is to enhance AVATAR with a hardware platform modeling capability, including the mapping of tasks and channels onto this platform. To do this, we rely on a newly defined **deployment diagram**, containing a SysML representation of hardware components, their interconnection, tasks and channels.

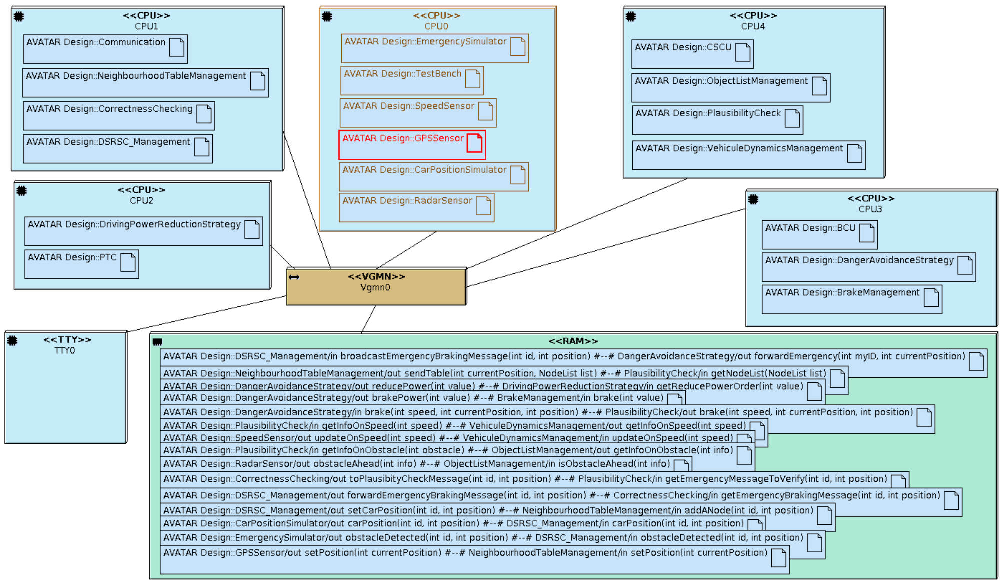
The AVATAR deployment diagram is this new modeling facility that allows a design to capture its hardware constraints. Figure 3 shows the main window of TTool with the deployment diagram of our example, the active braking application. We use a generic interconnect, a VGMN (Virtual generic Micro Network), along with five CPUs and one memory bank, which is not so realistic with regards to the real architecture, but far more convenient to explain the technical issues on code transformations. Platforms can be modified in a matter of minutes, for example by adding a second RAM and mapping part of the channels onto it. A valid platform must contain at least one CPU, one memory bank and one TTY.

5.2. New Tool Chain

From the deployment diagram, and from the previously existing software component diagrams, a fully executable SoCLib specification for a MPSoC can be generated. A new tool chain has been defined in order to support that model transformation (see Figure 4). The main components of this model transformation are:

- **libavatar** Runtime for SoCLib, implements the AVATAR operators.
- **DDSyntaxChecker** checks the syntax of the deployment diagrams and identifies their elements.
- **AVATAR2SOCLIB** translates AVATAR blocks (i.e., software components) into C POSIX tasks and generates the main program.
- **TopcellGenerator** generates a SystemC top cell for cycle accurate bit accurate simulation.
- **LdscriptGenerator** generates the linker script taking into account the mapping specified in the deployment diagram.

Note that the arc crossing over between *AVATARDDSpecification* and *AVATAR2SOCLIB* is necessary: information from the deployment diagram is required to generate the application code.



<<CPU>>
CPU1
AVATAR Design::Communication
AVATAR Design::NeighbourhoodTableManagement
AVATAR Design::CorrectnessChecking
AVATAR Design::DSRSC_Management
<<CPU>>
CPU0
AVATAR Design::EmergencySimulator
AVATAR Design::TestBench
AVATAR Design::SpeedSensor
AVATAR Design::GPSSensor
AVATAR Design::CarPositionSimulator
AVATAR Design::RadarSensor
<<CPU>>
CPU4
AVATAR Design::CSCU
AVATAR Design::ObjectListManagement
AVATAR Design::PlausibilityCheck
AVATAR Design::VehiculeDynamicsManagement
<<CPU>>
CPU2
AVATAR Design::DrivingPowerReductionStrategy
AVATAR Design::PTC
<<CPU>>
CPU3
AVATAR Design::BCU
AVATAR Design::DangerAvoidanceStrategy
AVATAR Design::BrakeManagement
<<VGMN>>
Vgmn0
<<TTY>>
TTY0
<<RAM>>
AVATAR Design::DSRSC_Management/in broadcastEmergencyBrakingMessage(int id, int position) #--# DangerAvoidanceStrategy/out forwardEmergency(int myID, int currentPosition)
AVATAR Design::NeighbourhoodTableManagement/out sendTable(int currentPosition, NodeList list) #--# PlausibilityCheck/in getNodeList(NodeList list)
AVATAR Design::DangerAvoidanceStrategy/out reducePower(int value) #--# DrivingPowerReductionStrategy/in getReducePowerOrder(int value)
AVATAR Design::DangerAvoidanceStrategy/out brakePower(int value) #--# BrakeManagement/in brake(int value)
AVATAR Design::DangerAvoidanceStrategy/in brake(int speed, int currentPosition, int position) #--# PlausibilityCheck/out brake(int speed, int currentPosition, int position)
AVATAR Design::PlausibilityCheck/in getInfoOnSpeed(int speed) #--# VehiculeDynamicsManagement/out getInfoOnSpeed(int speed)
AVATAR Design::SpeedSensor/out updateOnSpeed(int speed) #--# VehiculeDynamicsManagement/in updateOnSpeed(int speed)
AVATAR Design::PlausibilityCheck/in getInfoOnObstacle(int obstacle) #--# ObjectListManagement/out getInfoOnObstacle(int info)
AVATAR Design::RadarSensor/out obstacleAhead(int info) #--# ObjectListManagement/in isObstacleAhead(int info)
AVATAR Design::CorrectnessChecking/out toPlausibityCheckMessage(int id, int position) #--# PlausibilityCheck/in getEmergencyMessageToVerify(int id, int position)
AVATAR Design::DSRSC_Management/out forwardEmergencyBrakingMessage(int id, int position) #--# CorrectnessChecking/in getEmergencyBrakingMessage(int id, int position)
AVATAR Design::DSRSC_Management/out setCarPosition(int id, int position) #--# NeighbourhoodTableManagement/in addANode(int id, int position)
AVATAR Design::CarPositionSimulator/out carPosition(int id, int position) #--# DSRSC_Management/in carPosition(int id, int position)
AVATAR Design::EmergencySimulator/out obstacleDetected(int id, int position) #--# DSRSC_Management/in obstacleDetected(int id, int position)
AVATAR Design::GPSSensor/out setPosition(int currentPosition) #--# NeighbourhoodTableManagement/in setPosition(int currentPosition)

Figure 3. Deployment Diagram of the Active Braking Application

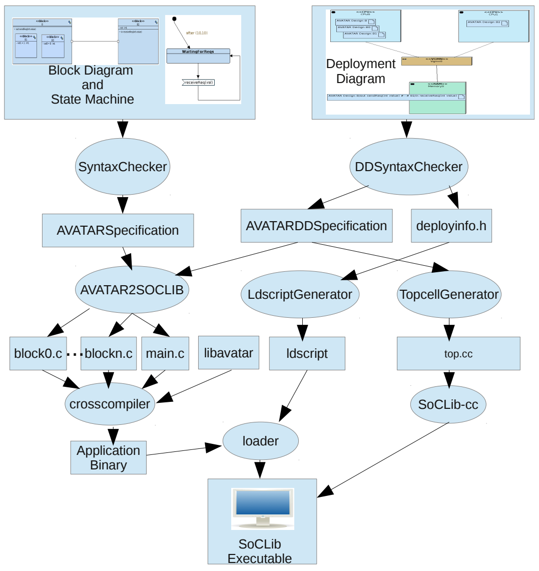


Figure 4. Tool chain

We now go through the main stages/elements involved in the model transformation.

5.3. The AVATAR Runtime ("libavatar")

The *AVATAR runtime* is a library of functions which capture the semantics of the AVATAR operators that appear in the code of the tasks (*delay, asyncRead*, etc.) and implements them using C/POSIX primitives. MutekH [4] is a free portable operating system for embedded platforms.

The AVATAR runtime more particularly focuses on channels, since the interconnect latencies and cache effects make the channels difficult to implement efficiently on a MPSoC platform. SoClib provides an efficient implementation of asynchronous channels as sowtware objects stored in on-chip memory, based on the Kahn [16] model, that can be accessed by any number of hardware or software reader and writer tasks alike [14]. Synchronous communications however require a central manager to resolve conflicts. In order to run on a SoCLib platform, instead of the local workstation, the runtime had to be adapted. Of course, the main model transformation issue is to maintain that precise semantics after transformation.

5.4. Code Generation

AvatarDeploymentPanelTranslator traverses the graphical elements of the DDiagram, extracts the data in form of objects (example: AvatarCPU) and adds it to a *AvatarDDSpecification object*.

5.4.1. Code Generation for AVATAR Blocks. Each mapped block is translated into a C POSIX thread. As an example, Figure 5 shows the state machine of the *DrivingPowerReductionStrategy* block. On receiving a signal *getReducePowerOrder* in the *WaitForReducePowerOrder* state, a waiting time within a given interval (minimum, maximum) is taken into consideration before the order is executed (state *WaitForReducePowerToBePerformed*). Figure 6 shows the generated code for

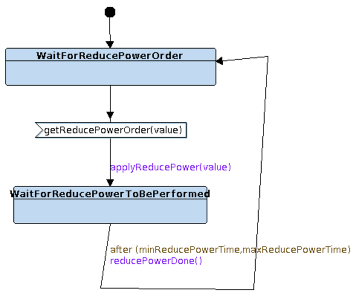

WaitForReducePowerOrder
getReducePowerOrder(value)
applyReducePower(value)
WaitForReducePowerToBePerformed
after (minReducePowerTime,maxReducePowerTime)
reducePowerDone()

Figure 5. State Machine of the Block DrivingPowerReductionStrategy

the *DrivingPowerReductionStrategy* block. The state machine has three states, including the start state. State *WaitForReducePowerOrder* awaits a synchronous message, which has been enqueued in a list of pending requests of the overall system by another block named *DangerAvoidanceStrategy* (see lower left of Figure 2). Then, *WaitForReducePowerToBePerformed* waits for a delay randomly selected between 10 and 20 milliseconds. That delays corresponds to the actuators work to really reduce the power.

```
#include "DrivingPowerReductionStrategy.h"
 static uint32_t _getReducePowerOrder;

#define STATE__START__STATE 0
#define STATE__WaitForReducePowerToBePerformed 1
#define STATE__WaitForReducePowerOrder 2
#define STATE__STOP__STATE 3
...
void *mainFunc__DrivingPowerReductionStrategy(void *arg){
  int value = 0;
  int minReducePowerTime = 10;
  int maxReducePowerTime = 20;

  int __currentState = STATE__START__STATE;
  ...
  pthread_cond_init(&__myCond, NULL);
  fillListOfRequests(&__list, __myname, &__myCond, &__mainMutex);

  while(__currentState != STATE__STOP__STATE) {
    switch(__currentState) {
      case STATE__START__STATE:
      __currentState = STATE__WaitForReducePowerOrder;
      break;

      case STATE__WaitForReducePowerOrder:
      __params0[0] = &value;
      makeNewRequest(&__req0, 853, RECEIVE_SYNC_REQUEST,0,0,0,1, __params0);
      __req0.syncChannel = &__DangerAvoidanceStrategy_reducePower\
__DrivingPowerReductionStrategy_getReducePowerOrder;
      __returnRequest = executeOneRequest(&__list, &__req0);
      clearListOfRequests(&__list);
      DrivingPowerReductionStrategy__applyReducePower(value);
      __currentState = STATE__WaitForReducePowerToBePerformed;
      break;

      case STATE__WaitForReducePowerToBePerformed:
      waitFor(minReducePowerTime, maxReducePowerTime);
      DrivingPowerReductionStrategy__reducePowerDone();
      __currentState = STATE__WaitForReducePowerOrder;
      break;
    }
  }
  return NULL;
}
```

Figure 6. Extract from the generated code for the DrivingPowerReductionStrategy block

5.4.2. Main Program Generation. The main program instanciates all necessary elements, e.g. the POSIX threads of the AVATAR blocks, and the SoCLib channels translated as software objects stored in the on-chip memory: these channels correspond to the AVATAR channels. Threads, corresponding to an AVATAR block each, are spawned from the main thread. Via *pthread_attr*, they are forced onto the CPU indicated in the Deployment Diagram. For example, *attr_t->cpucount=2* forces thread *DrivingPowerReductionnStrategy* onto CPU2. In order to map channels to specific memory areas, we name a software object whose mapping will be performed in the linker script using the same identifier.

Figure 7 shows an extract from the main program, focusing on our example block. First, the POSIX threads are initialized. Next, the signals belonging to the channel are associated to their input and output ports (for lack of space, we show this only for the channel *DrivingPowerReductionStrategy_getReducePowerOrder*). The threads are then created, attributes set beforehand, for example *attr_t->cpucount* to force a thread upon CPU 2. Finally, all threads are joined to wait for the program completion.

```
/* Synchronous channels */
syncchannel __DangerAvoidanceStrategy_reducePower\
__DrivingPowerReductionStrategy_getReducePowerOrder;
...
int main(int argc, char *argv[]) {
  void *ptr;
  pthread_barrier_init(&barrier,NULL, NB_PROC);
  pthread_attr_t *attr_t = malloc(sizeof(pthread_attr_t));
  pthread_attr_init(attr_t);
  pthread_mutex_init(&__mainMutex, NULL);

  /* Synchronous channels */
  __DangerAvoidanceStrategy_reducePower\
__DrivingPowerReductionStrategy_getReducePowerOrder.inname
    ="getReducePowerOrder";
  __DangerAvoidanceStrategy_reducePower\
__DrivingPowerReductionStrategy_getReducePowerOrder.outname
    ="reducePower";
  ...
  /* Threads of tasks */
  pthread_t thread__DrivingPowerReductionStrategy;
  ...
  ptr =malloc(sizeof(pthread_t));
  thread__DrivingPowerReductionStrategy= (pthread_t)ptr;
  attr_t = malloc(sizeof(pthread_attr_t));
  attr_t->cpucount = 2;

  pthread_create(&thread__DrivingPowerReductionStrategy, NULL,
    mainFunc__DrivingPowerReductionStrategy, NULL);
  ...
  pthread_join(thread__DrivingPowerReductionStrategy, NULL);
  return 0;
}
```

Figure 7. Extract from generated main program

5.5. Top Cell Generation

SoCLib is based on the *shared memory* paradigm, where a target is identified by the most significant bits of its address in a common memory space. The top cell generator thus must determine, for each target component, its unique target number associated to the segment address. We opted for the generic platform described in Figure 8: the platform considered in the case study has been derived from this generic platform. The platform features five PowerPC cores with integrated data and instruction cache (Xcache), one TTY, and one memory bank. Components are interconnected by a VGMN (see Section 5.1). VCI target interfaces are depicted with lighter arrows, initiators with darker arrows.

Some features must be explicitly captured in the Deployment Diagram, like CPUs and memory banks, as shown in Figure 3, while others are totally hidden to the TTool user, e.g. the numbering of target segments. Moreover, a timer, ICU and simhelper containing some simulation support facilities are currently generated *transparently* for the user. In the same way, the size of memory segments is given by a default value and the starting addresses are calculated by the top cell generator tool. Specific use cases may need other formats and must therefore modify the generated code.

The mapping of tasks to processors has no impact on the top cell, the sections containing channels however have to be listed in the call to the loader, see upper part of Figure 9.

The top cell generation takes as input the *AvatarDDSpecification* object (see Figure 4), and proceeds as follows:

1. We generate all segment addresses transparently, with exception of the segments containing channels which are visible in the Deployment Diagram.

2. All platforms use a *flattened device tree* (FDT) stored in ROM and contain a *simhelper* component for simulation support.

3. We systematically add a multi-timer and an interrupt control unit (ICU) target, even if not used by the application, and generate connection of all possible interrupt lines.

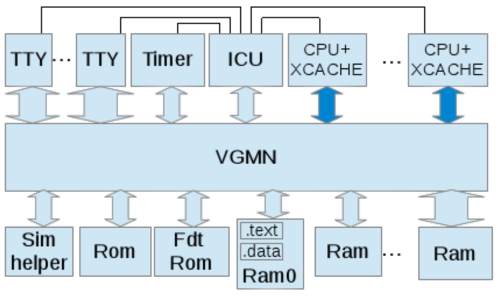


Figure 8. Generic SoCLib Platform

Top cell

```
data_ldr.load_file(std::string(kernel_p)
  + ";.data;.channel0;  ...
       .channel14;.cpudata;.contextdata");
```

Excerpt of the *ldscript*
Mapping channels on a single RAM:

```
.channel0  :  {  *(section_channel0)}  >  mem_ram
...
.channel14  :  {  *(section_channel14)}  >  mem_ram
```

Mapping on two RAMs:

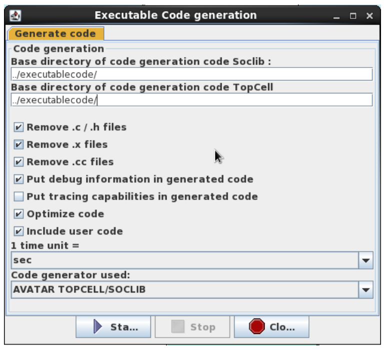
```
.channel0  :  {  *(section_channel0)}  >  mem_ram0
...
.channel14  :  {  *(section_channel14)}  >  mem_ram1
```

Figure 9. Top cell (top) and ldscript (bottom)

5.6. The Linker Script Generator

Handling the mapping of channels is rather difficult since it is related to the main program, the top cell and the ldscript. The linker script (*ldscript*) is a file which defines the memory layout; it associates each entry section to an output section, which receives its address in memory. It is generated by a tool developed in the context of [2] and uses essentially the C preprocessor.

The mapping information of channels in *AvatarDeploymentPanelTranslator* are then used to generate a file *deployinfo.h*: the latter is meant to be included by the ldscript generator. the lower part of Figure 9 shows an excerpt from the generated ldscript for our example. Thus, we first map all channels on one memory bank, then on two different banks. The section *.channel* of


Executable Code generation

Generate code

Code generation

Base directory of code generation code Soclib : ../executablecode/

Base directory of code generation code TopCell ../executablecode/

- [✓] Remove .c / .h files
- [✓] Remove .x files
- [✓] Remove .cc files
- [✓] Put debug information in generated code
- [ ] Put tracing capabilities in generated code
- [✓] Optimize code
- [✓] Include user code

1 time unit = sec

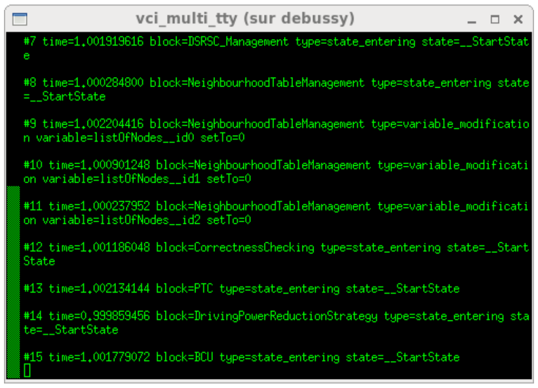
Code generator used: AVATAR TOPCELL/SOCLIB

Figure 10. Menu for SoCLib code generation

the *ldscript* is made from *section_channel* which in turn contains the software object specified in the main program in order to represent the channel. Several channels can be placed on the same memory bank. The loader invoked in the top cell (lower part of figure 9 then places it on the RAM as specified in the Deployment Diagram.

5.7. Using TTool/SoCLib

The generation of the code, top cell and ldscript has been integrated into TTool with a press-button approach. When checking the syntax of the deployment diagram, TTool also now checks for related blocks, interconnections between blocks, and state machines diagrams. Then, TTool displays the code generation dialog window from which executable code can now be generated, as explained in previous section, for the SoCLib platform. Figure 10 shows the code generation menu of TTool. Figure 11 shows the resulting SoCLib simulation. The application and main program code genera-



```
#7 time=1.001919616 block=DSRSC_Management type=state_entering state=__StartStat
e

#8 time=1.000284800 block=NeighbourhoodTableManagement type=state_entering state
=__StartState

#9 time=1.002204416 block=NeighbourhoodTableManagement type=variable_modificatio
n variable=listOfNodes__id0 setTo=0

#10 time=1.000901248 block=NeighbourhoodTableManagement type=variable_modificati
on variable=listOfNodes__id1 setTo=0

#11 time=1.000237952 block=NeighbourhoodTableManagement type=variable_modificati
on variable=listOfNodes__id2 setTo=0

#12 time=1.001186048 block=CorrectnessChecking type=state_entering state=__Start
State

#13 time=1.002134144 block=PTC type=state_entering state=__StartState

#14 time=0.999859456 block=DrivingPowerReductionStrategy type=state_entering sta
te=__StartState

#15 time=1.001779072 block=BCU type=state_entering state=__StartState
```

Figure 11. Using SoCLib from TTool

tion takes less than one second. The generation of the SystemC executable takes around 30 seconds on a 64 bit 4 core Xeon 2 Gbyte RAM machine under scientific Linux 64 bit (Table 1). Among others, CABA level sim-

top cell and ldscript generation	main and block code generation	compilation to application executable	soclib platform compilation
0.31 s	1.14 s	14.61 s	28.74 s

Table 1. Performance results for the case study

ulation potentially allows to measure cache miss rates, latency of each step of the individual cache miss, traffic on the interconnect, latency of each transaction on the interconnect, fill state of the buffers, knowing which lock is taken/released and the cycle when this happens. For example, the effect of a remapping of channels as shown in the upper part of Figure 9, or a change to the cache parameters can be analyzed in full detail.

6. Current Limitations

The approach is already available in TTool an experimental branch (-experimental option). Yet, it will be publicly available before the ERTS'2016 edition (a live demonstration will be performed).

From the model and translation point of view, synchronous channels currently require the use of a central manager, thus generating a significant overhead due to synchronization traffic on the interconnect. Adding a specific support for specific automotive interconnects (CAN, flexray) is planned.

From the simulation perspective, we opted for a prototyping environment with a low-level abstraction level (CABA level). Thus, an obvious limitation is the simulation speed. We need to get comparative results and work on speeding up the simulation. The simulation speed drawback is probably due to other factors. Currently, we use a flat, generic interconnect, however with contention and cache effects. Also, instead of a detailed CABA model of the processor, we use a instruction set simulator [23] to speed up simulation. Due to simulation complexity, we are still limited to some dozen processors, yet, it should be enough for most embedded applications.

A variety of low level performance measuring tools exists for SoCLib, among others giving cycle level information on the channel fill state and latency on the interconnect [13, 15]; these will have to be integrated, opening the way to automated feedback of performance results and, ultimately, Design Space Exploration.

7. Discussion and Future Work

The paper demonstrates the use of a recent extension to the TTool/AVATAR environment, by a larger case study stemming from automotive systems. One strength of our approach is that it offers a prototyping and exploration solution for engineers from industry, accustomed to the use of UML/SysML diagrams, while maintaining precise simulation results in addition to formal proofs, all in a joint framework. Our framework targets embedded systems with complex platforms, such as the ones found in telecommunication and transportation applications, but also automotive [10] and avionics providers, in particular those who already use AVATAR.

The next technical steps will consist in supporting more hardware components (e.g., co-processor wrappers, other types of interconnect, DMA), and more mapping capabilities (e.g., stacks, locks, see [14]). The support of synchronous channels requires a central request manager as such a semantics is not natively supported by SoCLib. Also, the current trade-off between simplicity and functionality might be reconsidered w.r.t. the usage of this new prototyping environment.

In TTool, the animation of the model in the prototyping phase is currently limited to a sequence diagram displaying transactions between software components. It would also be useful to provide information about hardware nodes, e.g., the load of processors, the state of buffers and the traffic on the interconnection network during simulation. Last by not least, for the moment, design space exploration is done by hand, The roadmap of our project envisages to integrate the feedback from detailed simulation such that it can be taken into account by the high level models.

Acknowledgments We would like to thank Alexandre Becoulet for his assistance on MutekH.

References

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

[2] L. Apvrille and A. Becoulet. Prototyping an embedded automotive system from its UML/SysML models. In *ERTSS'2012*, Toulouse, Feb. 2012.

[3] 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.

[4] A. Becoulet. Mutekh. http://www.mutekh.org.

[5] M. Ben Youssef, J.-F. Boland, G. Nicolescu, G. Bois, and J. Hugues. Bridging the high-level model to execution platform for design space exploration and implementation. In *ERTSS*, 2014.

[6] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag, 2004.

[7] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, July 2009.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] D. Genius. Measuring Memory Latency for Software Objects in a NUMA System-on-Chip Architecture. ReCoSoc, Darmstadt, Germany, July 2013.

[14] 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.

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

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

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

[18] P. Lieverse, T. Stefanov, P. van der Wolf, and E. F. Deprettere. System level design with spade: an M-JPEG case study. In *ICCAD*, pages 31–38, 2001.

[19] P. Lieverse, P. van der Wolf, K. A. Vissers, and E. F. Deprettere. A methodology for architecture exploration of heterogeneous signal processing systems. *VLSI Signal Processing*, 29(3):197–207, 2001.

[20] G. Pedroza, D. Knorreck, and L. Apvrille. 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, May 2011.

[21] 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.

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

[23] N. Pouillon, A. Becoulet, A. V. de Mello, F. Pêcheux, and A. Greiner. A generic instruction set simulator API for timed and untimed simulation and debug of MP2-socs. In *RSP*, pages 116–122. IEEE, 2009.

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

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

[26] 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.

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