Project acronym:	EVITA
Project title:	E-safety vehicle intrusion protected applications
Project reference:	224275
Program:	Seventh Research Framework Program (2007–2013) of the European Community
Objective:	ICT-2007.6.2: ICT for cooperative systems
Contract type:	Collaborative project
Start date of project:	1 July 2008
Duration:	42 months

Deliverable D4.2.3: LLD Modeling, Verification and Automatic C-Code Generation

Authors:	Gabriel Pedroza, Ludovic Apvrille (Institut Télécom)
Reviewers:	Dirk Scheuermann (Fraunhofer Institute SIT)

Dissemination level:	Public
Deliverable type:	Report
Version:	1.0
Submission date:	January 5, 2012

Abstract

The objective of the EVITA project is to design, verify, and prototype an architecture for automotive on-board networks where security-relevant components are protected against tampering, and sensitive data are protected against compromise. Thus, EVITA will provide a basis for the secure deployment of electronic safety aids based on vehicle-to-vehicle and vehicle-to-infrastructure communication.
This document summarizes Unified Modeling Language (UML) modeling and verification of EVITA low level SW components (Low Level Drivers). Verification is focused on functional and safety aspects and aims to increase the level of assurance and reliability of components. By implementing an UML-to-C code generator, verified UML-models can be automatically translated to C-code to be finally integrated into the EVITA prototype architecture. The experience and results within this approach are described.

Contents

1 Introduction **1**
1.1 Objectives and Tasks 1
1.2 Outline 2

2 AVATAR Approach Overview **3**
2.1 The AVATAR Profile 3
2.1.1 Methodology 3
2.1.2 Block and State Machine Diagrams 4
2.2 Formal Verification 5

3 UML Modeling and Formal Verification **6**
3.1 LLD Requirement Analysis 6
3.2 LLD Modeling Overview 6
3.3 LLD Verification 10
3.3.1 Abnormal Scenario: Race Condition in Queuing 16
3.3.2 Nominal Scenario: Queuing Parallel Requests 17
3.3.3 Formal Verification with Observers 18
3.4 Conclusions 21

4 Automatic C-Code Generation **22**
4.1 Code Generation 22
4.1.1 AVATAR to C/POSIX 22
4.1.2 AVATAR Runtime 23
4.1.3 Use of the Generated Code 23
4.1.4 Implementation and Results 24
4.1.5 Applying the Code Generator to LLD 25
4.1.6 Adaptation of code Generation for Drivers 26
4.2 Full Code Generation 29
4.3 Conclusion 29

5 Conclusions and Future Work **30**

List of figures

1 Five stage methodology associated to AVATAR profile 3
2 Block showing its list of attributes and communication signals 4
3 Overview of an State Machine Diagram (SMD) 5
4 Overview of Low Level Driver (LLD) model 7
5 Overview of Middleware block diagram 8
6 Overview of LLD block diagram 8
7 Driver overall behavior 9
8 Excerpt of Resources block diagram showing HSM model 10
9 Stimuli/Response Model verification 12
10 LLD verification approach 13
11 Overview of a block Observer 20
12 SMD of an Observer verifying correct queuing of requests 20
13 Prototyping environment based on TTool, SoCLib and MutekH 25
14 First excerpt of execution traces of LLD 26
15 Second excerpt of execution traces of LLD 27

List of tables

1 Test Cases for Simulation 13
2 Verification goals and Computational Tree Logic (CTL) formulas 14
3 Verification results for the abnormal scenario 17
4 Verification results for the nominal scenario 19

List of abbreviations

API	Application Programming Interface
CCS	Communication Control and Status module
CTL	Computational Tree Logic
DIO	Digital Input/Output
HSM	Hardware Security Module
ICU	Input Capture Unit
LLD	Low Level Driver
MISO	Master Input Slave Output
MOSI	Master Output Slave Input
SBB	Security Building Block
SMD	State Machine Diagram
SPI	Serial Peripheral Interface
UML	Unified Modeling Language

Document history

Version	**Date**	**Changes**
0.1	2011-08-26	Version 0.1 created
0.2	2011-12-01	Section including LLD modeling and verification finished
0.3	2011-12-16	Section including code generation finished
1.0	2012-01-05	Final Version

1 Introduction

As stated in the EVITA Description of Work and in the Annex I, the following technical work was proposed to be achieved by *Institut Télécom* as part of T4200:

- Demonstrate that the use of high-level models and formal techniques can enhance low-level software components reliability.
- Demonstrate that the use of code generation techniques from high-levels models can also enhance low-level software components reliability.

This report gives a closer look on the work performed by Institut Télécom within D4.2.2. It also serves as a public report about non-confidential T4200 issues.
For high-level models, the description of work proposes to rely on the use of UML-like languages. For formal proof or code generation purpose, we suggested to reuse our previous work on the TURTLE UML profile [3]. Indeed, TURTLE includes formal proof facilities as well as code generation techniques. However, TURTLE suffers several drawbacks. In particular, it was based on an "old" version of UML: AVATAR [13] now supersedes TURTLE, and is fully supported with a free software toolkit named TTool [10]. AVATAR/TTool offers high-level modeling edition facilities in SysML. It also includes simulation, formal verification and code generation capabilities. In particular, formal verification capabilities were introduced in the scope of EVITA [6], and code generation techniques were introduced in the scope of the work reported in this document. Along with that, results on modeling and verification of EVITA components with regard to security have been published [14]. Necessary adaptations of AVATAR for the specific purpose of drivers reliability are presented in this report.

1.1 Objectives and Tasks

The two main objectives in this work are listed just below:

1. **Perform formal verifications of the Basic Software"LLD"** described in [5], and provided by Fujitsu. This task is achieved with the formally defined AVATAR SysML environment. In AVATAR, formal verification can indeed be performed at the push of a button. AVATAR also comes with simulation techniques that are useful for a fast model debugging.

2. **Provide automatic generation of C code from UML models** (i.e., from AVATAR models). To accomplish this objective, an automatic UML-to-C code generator should be specified and implemented. Ideally, the automatic generation of code shall be evaluated with regards to reliability of automatically generated code vs. handmade code, and with regards to code integration. The main challenges are: to make the code suitable for the target EVITA architecture and to ensure that some properties proved at UML level are preserved by the C code. According to the manpower dedicated to this task, we do not intend to settle a certified / qualified code generator, but **simply to demonstrate that code generation can be practically applied to high-level models**.

To reach previous objectives next tasks were carried out:

T. 1: Closely understand Driver specification D4.2.2 [5] and related EVITA specifications (D4.1.1 [7], D3.2[16])

T. 2: Analyze Driver C code made by Fujitsu

T. 3: Identify/choose relevant subparts of Drivers, e.g., control parts

T. 4: Define how the functional correctness of the model can be evaluated

T. 5: Design a model compliant with Driver specification and code

T. 6: From the driver model, evaluate Driver functional correctness

T. 7: Propose Driver improvements and design an improved model of Drivers

T. 8: Provide feedback to D4.2.2 [5] according to evaluation results

T. 9: Propose and analyze candidate translations of AVATAR models into C code

T. 10: Select a translation, implement it, and test it

T. 11: Work on how the generated code could be integrated into the EVITA platform

T. 12: Elaborate this report and get partners feedback

T. 13: Iterate on previous tasks until verification and code generation goals are reached

1.2 Outline

This report is structured as follows. A brief description of the SysML/UML profile AVATAR is presented in section 2. AVATAR profile was proposed in the scope of EVITA, but in fact AVATAR targets the modelling and verification of embedded systems in general. The overall methodology and its limitations are presented in this section. Section 3 briefly presents the Driver model as well as an overview of formal proofs on the model. The overall verification results are accordingly presented at the end of the section. In section 4 the UML-to-C code generation is presented, first in its general approach, and then on examples, including LLD. Finally in section 5, the conclusions of our work are presented.

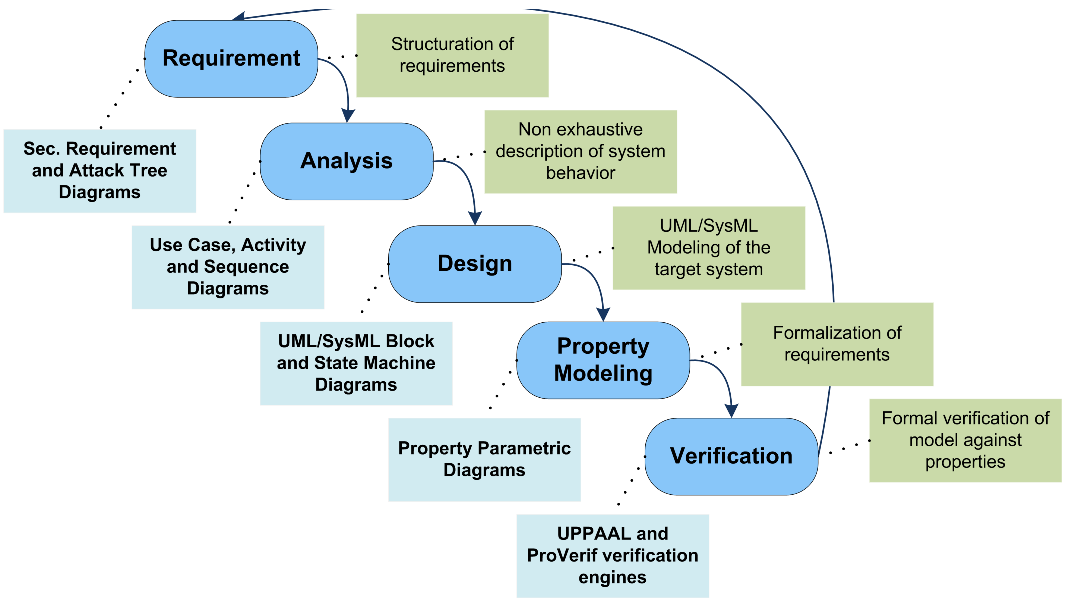
2 AVATAR Approach Overview

2.1 The AVATAR Profile

The AVATAR environment reuses eight of the SysML diagrams [12]. It further structures Sequence Diagrams using an Interaction Overview Diagram (a diagram defined in UML2 [12], not by SysML). The AVATAR profile is syntactically and semantically defined by a meta-model. Besides a syntax, a semantics and a tool support, a profile is also characterized by a methodology.

2.1.1 Methodology

The AVATAR methodology comprises the following stages (see figure 1):



Requirement
Structuration of requirements
Sec. Requirement and Attack Tree Diagrams
Analysis
Non exhaustive description of system behavior
Use Case, Activity and Sequence Diagrams
Design
UML/SysML Modeling of the target system
UML/SysML Block and State Machine Diagrams
Property Modeling
Formalization of requirements
Property Parametric Diagrams
Verification
Formal verification of model against properties
UPPAAL and ProVerif verification engines

Figure 1 Five stage methodology associated to AVATAR profile

1. **Requirement capture**. Requirements and properties are structured using AVATAR Requirement Diagrams. At this step, properties are just defined with a specific label as test cases.

2. **System analysis**. A system may be analyzed using usual UML diagrams, such as Use Case Diagrams, Interaction Overview Diagrams and Sequence Diagrams.

3. **System design**. The system is designed in terms of communicating SysML blocks described in an AVATAR Block Diagram, and in terms of behaviors described with AVATAR SMDs.

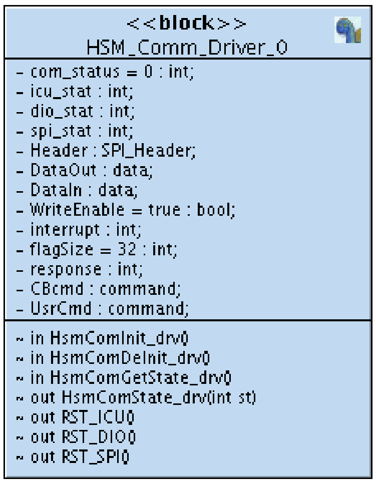
4. **Property modeling**. The formal semantics of properties is defined within TEPE [9] Parametric Diagrams (PDs). Since TEPE PDs involve elements defined during the system design phase (e.g, a given integer attribute of a block), TEPE PDs may be defined only after a first system design has been performed.

5. **Formal verification**. Safety properties can finally be verified over the system design, and for each test case.

Once all properties are proved to hold, requirements, system analysis and design, as well as properties may be further refined. Thereafter, and similarly to most UML profiles for embedded systems, the AVATAR methodological stages are reiterated. Having reached a certain level of detail, refined models may not be amenable to formal verification any more. Therefore the generation of prototyping code may become the only realistic option.

2.1.2 Block and State Machine Diagrams

Apart from their formal semantics, AVATAR Block and State Machine Diagram (SMD) only have a few characteristics which differ from the SysML ones.
An AVATAR block defines a list of attributes, methods and signals. Signals can be sent over synchronous or asynchronous channels. Channels are defined using connectors between ports. Those connectors contain a list of signal associations. Figure 2 gives the example of a block defining attributes, and in/out signals.



```
<<block>>
HSM_Comm_Driver_0
- com_status = 0 : int;
- icu_stat : int;
- dio_stat : int;
- spi_stat : int;
- Header : SPI_Header;
- DataOut : data;
- DataIn : data;
- WriteEnable = true : bool;
- interrupt : int;
- flagSize = 32 : int;
- response : int;
- CBcmd : command;
- UsrCmd : command;
~ in HsmComInit_drv()
~ in HsmComDeInit_drv()
~ in HsmComGetState_drv()
~ out HsmComState_drv(int st)
~ out RST_ICU()
~ out RST_DIO()
~ out RST_SPI()
```

Figure 2 Block showing its list of attributes and communication signals

A block defining a data structure merely contains attributes. On the contrary, a block defined to model a sub-behavior of the system must define an AVATAR State Machine. AVATAR SMDs are built upon SysML State Machines, including hierarchical states. Figure 3 presents a typical AVATAR state machine made upon states, message sending and receiving, and transitions enriched with guards, temporal constraints and operations.
AVATAR State Machines further enhance the SysML ones with temporal operators:

- **Delay:** $after(t_{min}, t_{max})$. It models a variable delay during which the activity of the block is suspended, waiting for a delay between t_{min} and t_{max} to expire.
- **Complexity:** $computeFor(t_{min}, t_{max})$. It models a time during which the activity of the block actively executes instructions, before transiting to the next state: that computation may last from t_{min} to t_{max} units of time.

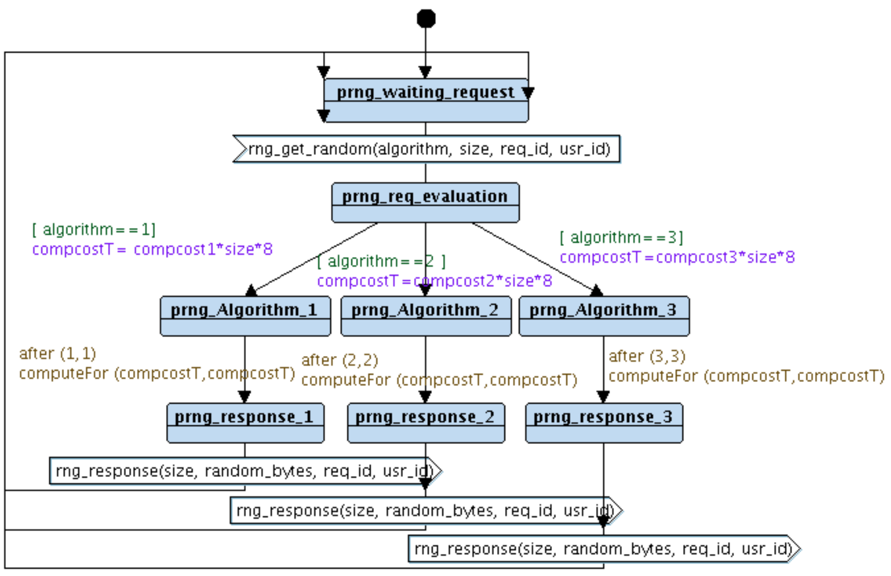

prng_waiting_request
rng_get_random(algorithm, size, req_id, usr_id)
prng_req_evaluation
[algorithm==1]
compcostT= compcost1*size*8
[algorithm==2]
compcostT=compcost2*size*8
[algorithm==3]
compcostT=compcost3*size*8
prng_Algorithm_1
prng_Algorithm_2
prng_Algorithm_3
after (1,1)
computeFor (compcostT,compcostT)
after (2,2)
computeFor (compcostT,compcostT)
after (3,3)
computeFor (compcostT,compcostT)
prng_response_1
prng_response_2
prng_response_3
rng_response(size, random_bytes, req_id, usr_id)
rng_response(size, random_bytes, req_id, usr_id)
rng_response(size, random_bytes, req_id, usr_id)

Figure 3 Overview of a SMD showing logical conditions (in green), operations (in violet), delays and complexities (in brown)

The combination of complexity operators ($computeFor()$), delay operators, as well as the support of hierarchical states - and the possibility to suspend an ongoing activity of a substate - endows AVATAR with main features for supporting real-time system schedulability analysis.

2.2 Formal Verification

The formal semantics of AVATAR is defined by model transformation to timed automaton. TTool implements that transformation, and relies on the UPPAAL model checker [4] to evaluate properties. More precisely, the following properties can be directly evaluated in TTool with a press-button approach:

Deadlock freedom: A deadlock situation arises when no logical progress is possible in the application. The property is satisfied if no deadlock situation is possible.

State Reachability: A state St within a SMD is reachable, if there exists a sequence of traversable transitions starting from the initial state and leading to the state St.

State Liveness: A state St within an SMD satisfies liveness property if St is eventually reached independently of the sequence of transitions that is traversed from the initial state.

More complex properties can also be expressed in TTool using CTL formulae. Further, TTool can automatically verify them with UPPAAL and display the verification result.

3 UML Modeling and Formal Verification

3.1 LLD Requirement Analysis

As proposed in the methodology, an AVATAR model is expected to be made with regard to a set of requirements and the system specification. However, in the scope of LLD models, the coding of EVITA Drivers was made almost “on the fly”, i.e., independently of a specification. Later, when the specification came, an iterative process between specification and code began, i.e., modifications within code pushed changes in specification and conversely. Consequently, the model and properties to be verified was not clear until the specification and code were finally released. Hence, the property analysis could not be performed as expected. However, based upon our previous experience on similar systems, several informal goals were established as general requirements. Those requirements were afterwards modified according to final specification and code (see subsection 3.3). The final requirements for verification are:

1. The Driver must be deadlock free, or differently said, the Driver must not get blocked forever in any circumstance.
2. All Driver interface functions must satisfy given properties stated in the specification, e.g., function re-entrancy.
3. Driver is stateful, that is, calls on driver interfaces results in corresponding modifications on internal driver data structures.
4. Driver phases must be correctly accomplished, e.g., initialization, de-initialization, request, response.
5. The Driver must correctly manage error callbacks from all HW modules used by t he driver, and in particular the HSM Hardware Security Module (HSM).
6. The Driver must process requests according to given directives, e.g., priority ordering.

3.2 LLD Modeling Overview

The LLD modeling task follows a combined top-down/bottom-up approach. The top-down approach takes into consideration the interfaces offered to requesting applications, as specified in EVITA deliverable D3.2 [16]: the EMVY Security Framework and the AUTOSAR stack. Indeed, both frameworks are considered to be the *Middleware* of the system executing right on top of drivers. On the other side, the bottom-up approach takes EVITA hardware components as the resources with which the driver communicate, that is the TC1797 TriCore and the HSM. Those components are further referred to as *LLD Resources*. Of course, resource layers are modelled taking EVITA D4.1.1 [7] as well as D3.2 [16], section 4, as main references. Finally, our LLD model is the tie between Middleware and Resources models (see figure 4).

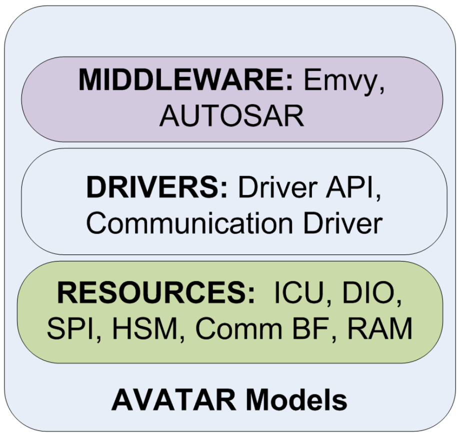

MIDDLEWARE: Emvy, AUTOSAR
DRIVERS: Driver API, Communication Driver
RESOURCES: ICU, DIO, SPI, HSM, Comm BF, RAM
AVATAR Models

Figure 4 Overview of LLD model

Next paragraphs briefly describe LLD model components, represented as UML Blocks (<<block>>). We recall that an AVATAR Block in the model is defined through a list of attributes - that may include user defined data types -, a list of methods - provided as documentation only -, and finally a list of input and output signals. Blocks are linked together via ports allowing signal-based synchronous and asynchronous communications. Along with that, Block's behavior is formally captured in an associated UML State Machine Diagram (SMD), i.e., an automata defined by states and directed transitions. Transitions are amenable for representation of boolean conditions, variable assignations, computational complexity and latencies. In addition, transitions can link input and output signals with states. An outgoing transition is traversed only if its respective conditions are satisfied. In case of transitions with non-mutually exclusive conditions, several transitions could be taken. For model simulation, a branch is randomly taken whilst for formal verification all traversable transitions are considered.

Middleware Model

Middleware Blocks are mostly used to test LLD functionalities. Indeed, test cases are defined within middleware Blocks and used later in simulation and formal verification. Figure 5 shows an overview of Middleware diagram model.

TESTCASE_INIT: Initializes, deinitializes and gets driver status.

TESTCASE_DRVREQS: Interface for sending HSM requests and performing callbacks asynchronously.

Driver Model

Blocks of the LLD model are listed below. This driver model is based upon the final versions of specification D4.2.2 [5] and LLD code made by Fujitsu. An overview of the model is shown in figure 6.

HSM_DRIVER: Receives requests from Middleware and puts requests in Request Queue. For each request, a respective entry is added in the Request List to trace the transac-

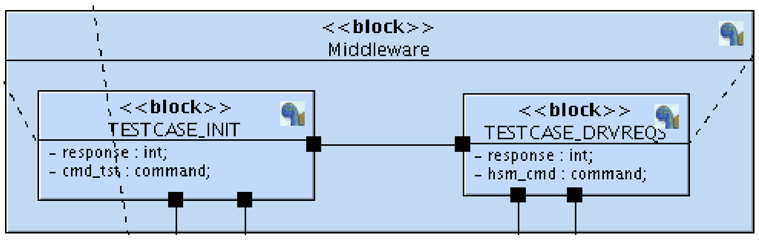


Figure 5 Overview of Middleware block diagram

tion. Callbacks from HSM are mapped to callback functions using the Request List. This Block serves as an interface to transfer signals to/from HSM_Comm_Driver.

HSM_Comm_Driver: Initializes/Deinitializes lower HW modules (ICU, DIO & SPI in Resources model). Models initialization, request and response phases, manages HOST buffer used for HSM data transfers and calls the Request Queue to process requests. This Block controls all communications with Resources model and models the communication handler coded by Fujitsu.

HSM_ QueueDriver: Models a circular buffer able to receive, queue and pop out requests from Middleware. Requests are ordered by priority.

HSM_ListDriver: Models a buffer storing the Request List. Each entry of the list stores a caller application id, the respective callback function and an identification assigned to the transaction.

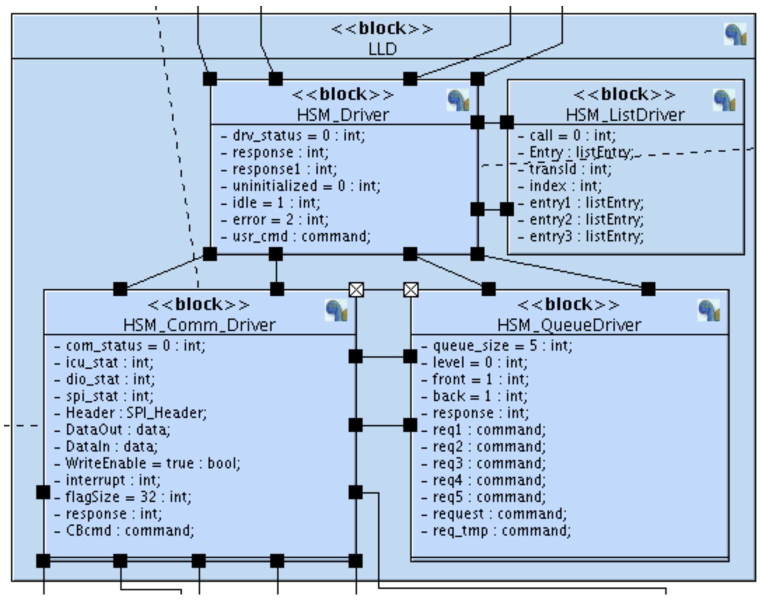


Figure 6 Overview of LLD block diagram

The general behavior of the Driver model is as follows (see figure 7). When the HSM_Driver receives a request from Middleware (point 1), it is stored in the queue HSM_-QueueDriver waiting for processing (point 2). The request may include an entry indicating

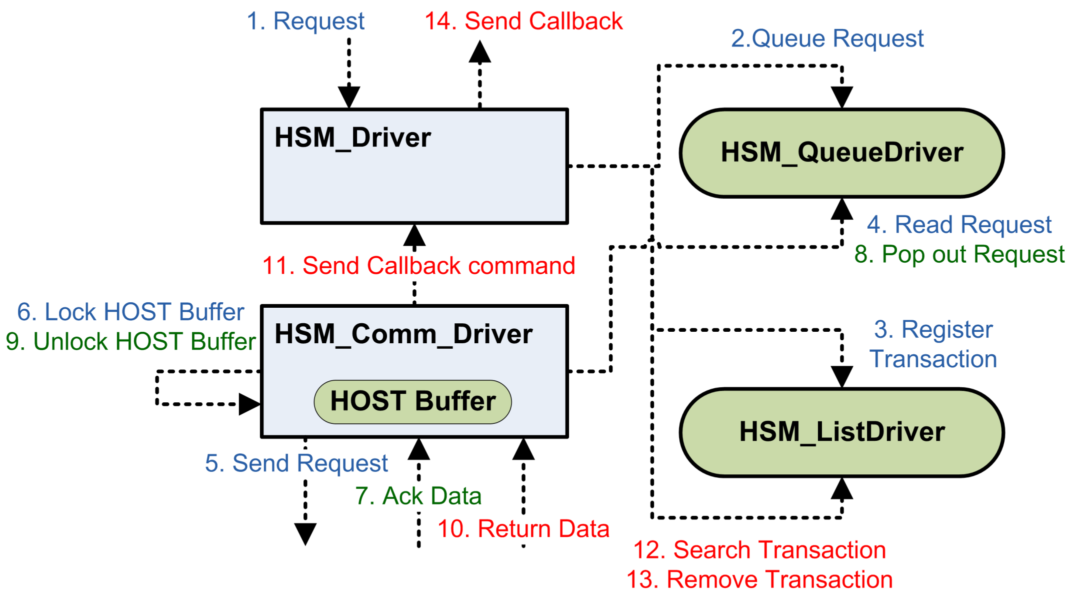


Figure 7 Driver overall behavior

its priority which is used to classify the request in the queue. Along with that, a reference to the transaction is stored in the bufffer named HSM_ListDriver (point 3). To process a request, HSM_Comm_Driver reads the head of HSM_QueueDriver (point 4), serializes the request and sends it to the HSM block via Serial Peripheral Interface (SPI) (point 5). Afterwards, the HSM_Comm_Driver waits for an acknowledgement from HSM block. In the mean time, no other requests can be processed, since the HOST buffer used for SPI communication is locked (point 6). When the acknowledgement is received (point 7), the HSM_Comm_Driver pops out the front of HSM_QueueDriver (point 8) and the HOST buffer is released (point 9). When a response to a previous request is signaled to HSM_Comm_Driver, it reads the response from the HSM block (point 10), de-serializes it, and sends a callback command to HSM_Driver (point 11). Then, HSM_Driver searches for the respective entry stored in HSM_ListDriver (point 12). Once found, the entry is removed (point 13). Finally, the returned data are sent to the original caller using the callback function associated to the request (point 14).

Resources Model

Blocks of this layer are listed just below. The implementation of the Hardware Security Module (HSM) specified in [7] is modeled among the resources. Further modules are considered for establishing a link between LLD and HSM functionalities.

EVITA_ICU: Receives interrupts from HSM, sends HOST-INT line status to HSM_-Comm_Driver Block.

EVITA_DIO: Resets SPI module on HSM, checks Input Capture Unit (ICU) status and controls Master Output Slave Input (MOSI) and Master Input Slave Output (MISO) communication modes, raises/lowers Chip Select line.

EVITA_SPI: Sends command Header, Latency and Data in MOSI mode, receives Latency and Data in MISO mode, models serializing and deserializing operations.

SPI_Connection: Transfers signals and data from slave SPI towards HSM and from HSM towards SPI slave.

SPI_Slave: Receives/sends data from/to TriCore modules (ICU, Digital Input/Output (DIO), SPI). Behaves as an interface for Communication Control and Status module (CCS). Reads and writes data from/to SharedMessageRAM Block, respectively.

CCS: Performs read and write operations on 6 HSM registers that control HSM communication and status. Raises/lowers host interrupts - received by LLD - and HSM interrupts - received by HSM CPU.

SharedMessageRAM: Models input (LLD-to-HSM) and output (HSM-to-LLD) RAM buffers. Stores input and output payload and size.

HSM_Mgmt: Sets CCS registers to allow LLD-HSM communication, transfers Security Building Block (SBB) calls and responses, controls HSM output buffer, reacts to HSM interrupts.

SBB_XYZ: Models a Security Building Block (SBB). Every SBB receives, processes and returns a response to each request.

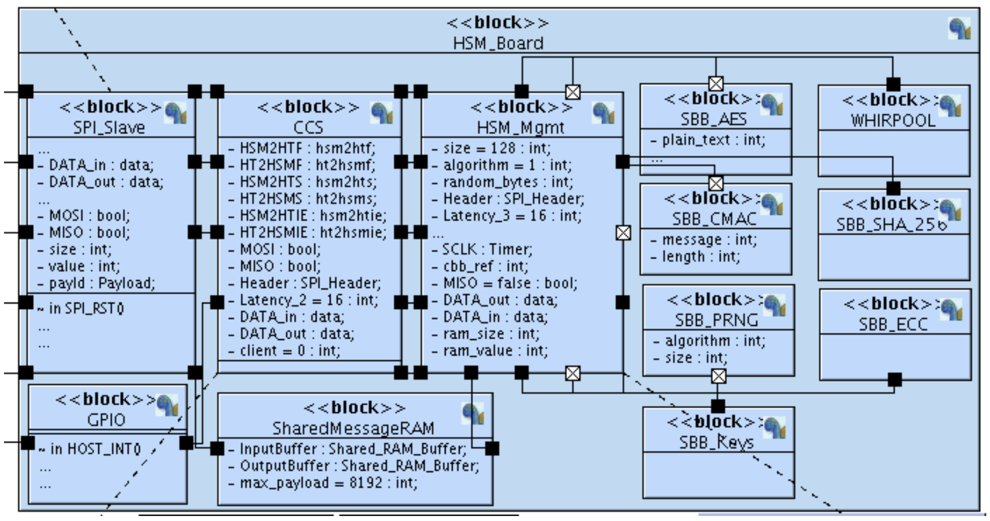


Figure 8 Excerpt of Resources block diagram showing HSM model

3.3 LLD Verification

Three layers have been modeled: Middleware, Driver and Resources. To verify the correctness of the Driver model, first the Resources model was validated against the specification provided in [7]. Afterwards, the Driver model was validated against the specification provided in [5], and taking the code of the Driver as reference. The model was finally presented to Fujitsu. To accomplish this stage, LLD behavior was represented in sequence diagrams and in state flow charts. The LLD constains three phases:

Initialization phase: Corresponds to Driver initialization. Buffers are cleared and HW modules are started. Once initialized, the Driver remains in idle state if the Request Queue is empty, if there is no interrupt from HSM or at last if the Driver waits for an acknowledgement from HSM.

Request phase: A request from the Middleware has been written into the Request Queue. The respective entry is created in the Request List. The Driver reads the front of the Queue, serializes and sends the request to the HSM. Afterwards, a HSM/CCS register is modified to signal the request to the HSM. The Driver can not send another request until the HSM acknowledges the current one.

Response phase: The LLD is called by the Resources layer via an interrupt. The LLD reads a HSM/CCS register to determine whether the instruction is an acknowledgement or a signal for returning data. In case of acknowledgement, the HSM/CCS register is reset and the request in front of the Queue is popped out. In case of returning data, they are read from shared RAM and afterwards the HSM/CCS register is reset. Afterwards, the data are deserialized and mapped into a callback function using the respective entry in the Request List.

The verification approach consists of two stages: **Test Case Simulation** and **Formal Verification**. Simulation stage helps to quickly execute specific test cases. Traces of the model's behavior can thus be analyzed. Amongst others, Simulation is used for model debugging. Formal verification stage proves whether expected properties are satisfied or not, not only in a specific trace but for the whole set of possible sequences thus achieving exhaustive proofs. As shown in figure 9, simulation and verification are carried out relying on a blackbox approach in which the model receives certain stimuli and returns a response. More precisely, stimuli and response are characterized and evaluated with respect to Driver specification and relying upon following notation:

SIS: Stimuli inside of specification

SOS: Stimuli Outside of Specification

MR: Model Response

According to previous statements, next definitions are adopted hereinafter for evaluation of results in model testing and verification.

System Correctness: Both the stimuli provided to the target model and the corresponding response are as specified.

System Inconsistency: The model receives a stimuli inside specification (SIS) but the respective response is not as specified.

System Robustness: When a stimuli outside of specification (SOS) is received, the system continues its nominal operation with other requests.

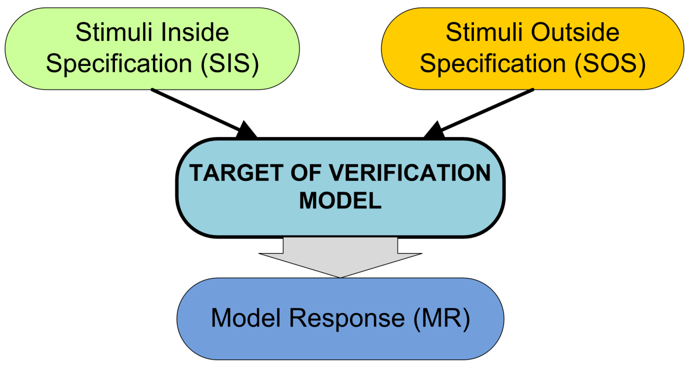

Stimuli Inside Specification (SIS)
Stimuli Outside Specification (SOS)
TARGET OF VERIFICATION MODEL
Model Response (MR)

Figure 9 Stimuli/Response Model verification

Finally, Simulation and Formal Verification of Driver model are conducted as follows (see figure 10):

Test Case Simulation: Several test cases are defined in the Middleware model targetting HSM functionalities or Driver phases. Test cases call one or more SBBs with different parameters and may apply these calls either in parallel or sequentially. The simulation purpose is to compare expected traces with obtained ones. In a nominal scenario, the model is tested with values inside of specification and without considering intentional misbehavior. Conversely, in an altered scenario, the model is tested using values outside of specification or intentionally introducing a misbehavior case, e.g., callbacks with wrong parameters. Thus, through simulation, model correctness and robustness are targetted by identifying possible inconsistencies or weaknesses.

Formal Verification: Several (informal) verification goals are first established in order to provide a first set of properties, and also in order to determine the scope of formal verifications. The goals mainly target deadlock freedom and correct Driver state handling as explained in section 3. Each verification goal is associated with one or more CTL expressions (formulas) representing the goal. Each CTL expression is automatically verified on the Driver model, and a true/false answer is thus obtained[1].

Table 1 presents a list of defined Test Case scenarios and expected Driver model behavior. Indeed, Test Cases define the values that should be returned by the HSM. Moreover, the model includes specific states to explicitly characterize Driver behavior. This way of modeling is particularly useful for simplifying model analysis and verification tasks. Thus, for instance, Driver behavior is considered as “correct” or “accomplished” if a specific state - or set of states - can be reached and the there is a match between expected and actually returned values.

[1]Combinatory explosion may be encountered. In that case, the property cannot be proved, and so, the only option is to further abstract the model

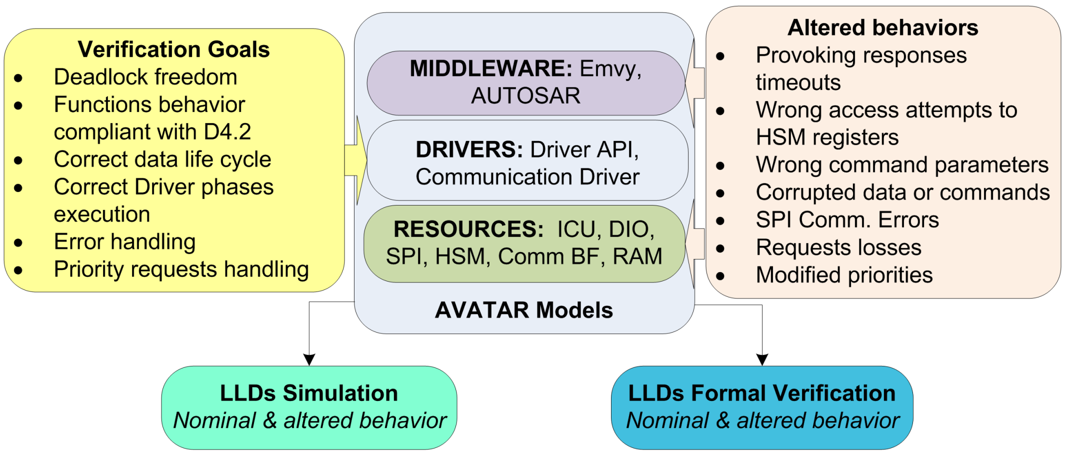

Verification Goals
Deadlock freedom
Functions behavior compliant with D4.2
Correct data life cycle
Correct Driver phases execution
Error handling
Priority requests handling
MIDDLEWARE: Emvy, AUTOSAR
DRIVERS: Driver API, Communication Driver
RESOURCES: ICU, DIO, SPI, HSM, Comm BF, RAM
AVATAR Models
Altered behaviors
Provoking responses timeouts
Wrong access attempts to HSM registers
Wrong command parameters
Corrupted data or commands
SPI Comm. Errors
Requests losses
Modified priorities
LLDs Simulation
Nominal & altered behavior
LLDs Formal Verification
Nominal & altered behavior

Figure 10 LLD verification approach

Table 1 Test Cases for Simulation

Test Case Description	**Expected Driver Behavior**
ST1: Single request sent to HSM	Driver model should be initialized and request and response phases accomplished
ST2: Sequence of non-continuous requests, i.e., Middleware waits for response before sending a new request to the same SBB	Initialization, request and response phases should be correctly performed. The HSM should be able to process all the requests
ST3: Continuous requests: the same SBB is called until the Request Queue is full	Initialization, request and response phases should be correctly accomplished. Once Request Queue is full, the Driver is unable to process additional requests
ST4: Continuous requests, several SBBs are called until the Request Queue is full	Initialization, request and response phases should be correctly accomplished. Once Request Queue is full, the Driver is unable to process additional requests
ST5: Continuous requests, several SBBs are called by several Middleware users until the Request Queue is full	Initialization, request and response phases should be correctly accomplished. Once Request Queue is full, the Driver is unable to process additional requests. Callbacks should be addressed to the original caller
ST6: Continuous requests, several SBBs are called by several Middleware users until the Request Queue is full. Request priorities are introduced	Initialization, request and response phases should be correctly accomplished. Once Request Queue is full, the Driver is unable to process additional requests. Callbacks should be addressed to the original caller according to request priorities

Table 2 shows a list of verification goals with associated CTL formulas and respective semantics and rationale. CTL formulas are automatically proved by injecting them in the TTool front-end.

Table 2 Verification goals and CTL formulas

Verification Goal 1 Deadlock freedom	**Rationale:** As a mandatory property, the Driver must not get blocked forever in any circumstance **CTL Semantics:** Apart from final states, no deadlock situations must appear
CTL1: *AG[not(deadlock) or FinalState]*	
Verification Goal 2 Correct HW modules initialization	**Rationale:** It proves that the initialization phase is always correctly accomplished by HSM_Comm_Driver. Initialization (Init phase) is mandatory for Driver operation **CTL Semantics:** If the HSM_Comm_Driver receives an init signal from the stop status, it must always finally initialize HW modules
CTL2: *AG[HSM_Comm_Driver.InitCommDriver ⇒ AF[HSM_Comm_Driver.ComDrvInitialized]]*	
Verification Goal 3 Correct HSM Driver initialization	**Rationale:** It proves that the initialization phase (Init phase) is always correctly accomplished by HSM_Driver. Initialization is mandatory for Driver operation **CTL Semantics:** If the HSM_Driver receives an init signal from the stop status, it must always initialize HSM_Comm_Driver and clear HSM_QueueDriver and HSM_ListDriver
CTL3: *AG[HSM_Driver.InitDriver ⇒ AF[HSM_Driver.DriverInitialized]]*	
Verification Goal 4 Correct HW modules de-initialization	**Rationale:** It proves that HSM_Comm_Driver can eventually de-initialize lower HW modules and go to stop status. De-initialization is a functional property of de-init phase **CTL Semantics:** If the HSM_Comm_Driver receives a de-init signal from the idle status, it must always de-initialize HW modules
CTL4: *AG[HSM_Comm_Driver.Deinit_HW ⇒ AF[HSM_Comm_Driver.ComDrvStop]]*	
Verification Goal 5 Correct HSM Driver de-initialization	**Rationale:** It proves that HSM_Driver can eventually de-initialize HSM_Comm_Driver and go to stop status. De-initialization is a functional property of de-init phase **CTL Semantics:** If the HSM_Driver receives a de-init signal from the idle status, it must always de-initialize HSM_Comm_Driver
CTL5: *AG[HSM_Driver.DeinitDrv ⇒ AF[HSM_Driver.DrvStop]]*	
Verification Goal 6 Correct Queuing of requests	**Rationale:** This property is mandatory to ensure that, unless the Queue is full, requests are always accepted by the Driver to be processed (request phase) **CTL Semantics:** If the HSM_Driver is in the idle status and receives a request from the Middleware, it must always queue the request unless Request Queue is full
CTL6: *AG[HSM_Driver.QueuingRequest and HSM_QueueDriver.level<max ⇒ AF[HSM_Driver.RequestInQueue]]*	

Continued on next page

Table 2 Verification goals and CTL formulas – continued from previous page

Verification Goal 7 Adequate HOST buffer protection	**Rationale:** The property is mandatory to ensure that requests are not overwritten in the HOST buffer (request phase) **CTL Semantics:** The HSM_Comm_Driver must not process any request unless the variable controlling HOST buffer access is set to true
CTL7: *AG[HSM_Comm_Driver.CheckQueue ⇒ HSM_Comm_Driver.WriteEnable==true]*	
Verification Goal 8 Correct processing of requests	**Rationale:** The property ensures that if a request in the Queue is read, it is always sent to the HSM what makes the Driver unable to process another request (request phase) **CTL Semantics:** Whenever there is a request in the Queue, the request phase must be accomplished and the HOST buffer locked
CTL8: *AG[HSM_Comm_Driver.ProcessRequest ⇒ AF[HSM_Comm_Driver.LockHostBuffer and HSM_Comm_Driver.WriteEnable==false]]*	
Verification Goal 9 Correct management of HSM interrupts	**Rationale:** The property ensures that HSM interrupts are correctly managed. The HOST_INT line should be raised before the Driver reads data from HSM shared RAM (response phase) **CTL Semantics:** Whenever an interrupt is signaled to the Driver, the HOST_INT line must be raised and the respective data must be finally read from the HSM shared RAM
CTL9: *AG[HSM_Comm_Driver.CheckHOSTINTline ⇒ AF[HSM_Comm_Driver.WaitHSM2HTF_ack]]*	
Verification Goal 10 Driver processes HSM acknowledgement	**Rationale:** The property ensures that HSM acknowledgements (response phase) enable HSM_Comm_Driver for processing more requests. It implies that requests accepted by the HSM are removed from the Queue (data life cycle) **CTL Semantics:** Whenever the HSM acknowledges a request to HSM_Comm_Driver, the front of the Queue is pop out and the HOST buffer is unlocked
CTL10: *AG[HSM_Comm_Driver.ResetHSM2HTF_ack ⇒ AF[HSM_Comm_Driver.PopFrontQueue and HSM_Comm_Driver.WriteEnable==true]]*	
Verification Goal 11 Driver processes SBB return code	**Rationale:** It proves that HSM_Comm_Driver properly manages returned data from HSM until respective command is sent to higher layers (response phase) **CTL Semantics:** Whenever the HSM signals returned data to HSM_Comm_Driver (callback), the data is read from shared RAM and deserialized. The resulting command must be finally sent to HSM_Driver
CTL11: *AG[HSM_Comm_Driver.ReadHSM2HTF_provide ⇒ AF[HSM_Comm_Driver.SendCBcommand]]*	

Continued on next page

Table 2 Verification goals and CTL formulas – continued from previous page

Verification Goal 12 Request Queue/List correspondence	**Rationale:** The property proves returned-sent data correspondence. Assuming that every request is processed by the HSM, it also ensures that entries in the Request List will be eventually removed (data life cycle) **CTL Semantics:** Whenever a callback command is received by HSM_Driver, a respective entry must be found in the Request List
CTL12: *AG[HSM_Driver.OperateOnList ⇒ AF[HSM_Driver.ReadEntry]]*	
Verification Goal 13 Correct callback function mapping	**Rationale:** The property proves a correspondence between callbacks and callers: each callback must be associated to a single caller what concludes the request-response cycle **CTL Semantics:** Whenever an entry is found in the Request List, the callback function must send the HSM response to origin caller
CTL13: *AG[HSM_Driver.ReadEntry ⇒ AF[HSM_Driver.BackIdle4 and HSM_Driver.response==1]]*	
Verification Goal 14 Correct test case execution	**Rationale:** It is proved that the state of the Test Case showing correct termination is always reached. Moreover, it implicitly proves that request-response correspondence is respected and correct overall Driver behavior **CTL Semantics:** Whenever a test case is executed the “end” state of the test case is reached
CTL14: *AF[TESTCASE_DRVREQS.InitTestCase ⇒ AF[TESTCASE_DRVREQS.EndTestCase]]*	

3.3.1 Abnormal Scenario: Race Condition in Queuing

The methodology upon which verification relies (see section 2), leads to an iterative process until targeted properties are satisfied. During that process, several issues were identified in the EVITA Driver. Thus, a relevant one is used to exemplify formal verification of an abnormal situation. Indeed, the EVITA Driver contains race conditions when accessing shared buffers like the Request Queue. Amongst others, the scenario:

1. Initializes the Driver, sends two requests in parallel and waits for the respective responses.
2. Executes the main phases of Driver behavior (init, request and response).
3. Models a non-exclusive access to the Request Queue.
4. Contains a race condition that may lead to request over-writting.
5. Targets model correctness and also robustness by exploring specific conditions.
6. Helps to formally prove Driver weaknesses or inconsistencies.
7. Is compliant with the behavior of the final Driver version used in the EVITA prototype.

The verification results for this scenario are presented as a traceability matrix in table 3. Proofs were carried out over an early model of EVITA Driver.

Table 3 Verification results for the abnormal scenario

Verification Goal	Result
Verification Goal 1 Deadlock freedom	Not Satisfied
Verification Goal 2 Correct HW modules initialization	Satisfied
Verification Goal 3 Correct HSM Driver initialization	Satisfied
Verification Goal 4 Correct HW modules de-initialization	Satisfied
Verification Goal 5 Correct HSM Driver de-initialization	Satisfied
Verification Goal 6 Correct Queuing of requests	Not Satisfied
Verification Goal 7 Adequate HOST buffer protection	Satisfied
Verification Goal 8 Correct processing of requests	Satisfied
Verification Goal 9 Correct management of HSM interrupts	Satisfied
Verification Goal 10 Driver processes HSM acknowledgement	Satisfied
Verification Goal 11 Driver processes SBB return code	Satisfied
Verification Goal 12 Request Queue/List correspondence	Satisfied
Verification Goal 13 Correct callback function mapping	Satisfied
Verification Goal 14 Correct test case execution	Not Satisfied

3.3.2 Nominal Scenario: Queuing Parallel Requests

The Driver model was improved to cope with identified issues. To make a comparison with respect to previous scenario, the model is also proved with two requests in parallel. However, the conditions become part of a nominal case which the Driver should be able to deal with. Amongst others, the scenario:

1. Initializes the Driver, sends two requests and waits for the respective responses.

2. Executes the main phases of Driver behavior (init, request and response).

3. Targets model correctness and possible Driver inconsistencies.

4. Helps to prove model improvements, e.g., mutually exclusive access to shared buffers.

5. Copes with state explosion problem during verification and decreases the time spent in verification tasks.

6. Is partially compliant with the behavior of the final Driver version used in the EVITA prototype.

The results from verification are presented as a traceability matrix in table 4. The verification was carried out over the improved model of EVITA Driver. More precisely, the improved model:

- Grants mutually exclusive access to shared buffers.

- Supports parallel and sequential execution of requests.

- Introduces and manages priorities in requests (used for queuing requests according to its relevance).

3.3.3 Formal Verification with Observers

In previous subsections, several verification goals were described and proved based on CTL formulas. That approach requires skills on the CTL language and on formal verification in general, which might be a limitation for non-experimented users. To overcome such issue, Observers can be introduced in the model using AVATAR blocks. Indeed, an Observer is a non-intrusive element that gets signals from the model, and uses them to determine whether the system is in a specified state, or not. In that latter case, an error state is usually defined: it suffices to prove the non-reachability of that error state to prove that the property modeled by the Observer is satisfied. To exemplify our approach, an instance of an Observer is described in next paragraphs.

The defined Observer targets the Verification Goal 6 defined in table 2. A nominal sequence of signals indicating a correct processing of a request is as follows:

1. A request is received by the Driver.

2. The request is sent by the Driver to the Request Queue and List.

3. After responses evaluation, the Driver signals correct storing of request.

Also, the Observer may have to simultaneously handle several requests sent to the driver at the same time. The observer is presented in figure 11 and its SMD diagram is given in Figure 12. The following signals are used by the observer to evaluate the property:

ARequestWasReceived(): Indicates that a request has been received by the Driver.

QueueLevel(level): Informs the response from the Request Queue.

Table 4 Verification results for the nominal scenario

Verification Goal	**Result**
Verification Goal 1 Deadlock freedom	Satisfied
Verification Goal 2 Correct HW modules initialization	Satisfied
Verification Goal 3 Correct HSM Driver initialization	Satisfied
Verification Goal 4 Correct HW modules de-initialization	Satisfied
Verification Goal 5 Correct HSM Driver de-initialization	Satisfied
Verification Goal 6 Correct Queuing of requests	Satisfied
Verification Goal 7 Adequate HOST buffer protection	Satisfied
Verification Goal 8 Correct processing of requests	Satisfied
Verification Goal 9 Correct management of HSM interrupts	Satisfied
Verification Goal 10 Driver processes HSM acknowledgement	Satisfied
Verification Goal 11 Driver processes SBB return code	Satisfied
Verification Goal 12 Request Queue/List correspondence	Satisfied
Verification Goal 13 Correct callback function mapping	Satisfied
Verification Goal 14 Correct test case execution	Satisfied

ARequestWasQueued(): Signals a correct storing of the request in the Queue.

The observer is able to sequentially receive signals corresponding to different requests. This explains why a second input signal "ARequestWasReceived()" is used in the Observer (see figure 12). Also, the Observer is able to detect if a request is overwritten: the input signal "RequestOverWritten()" is made for that purpose. And so, receiving that event may lead to an error state, depending whether the signal contains a 0 or 1 attribute. Finally, the use of Observers eases formal verification of properties because properties are directly described with the formalism of the model, and directly within the design.

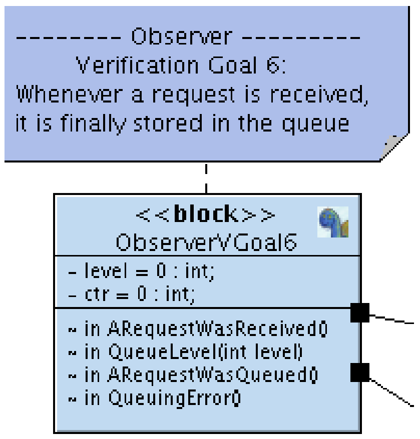

-------- Observer ---------
Verification Goal 6:
Whenever a request is received,
it is finally stored in the queue
<<block>>
ObserverVGoal6
- level = 0 : int;
- ctr = 0 : int;
~ in ARequestWasReceived()
~ in QueueLevel(int level)
~ in ARequestWasQueued()
~ in QueuingError()

Figure 11 Overview of a block Observer

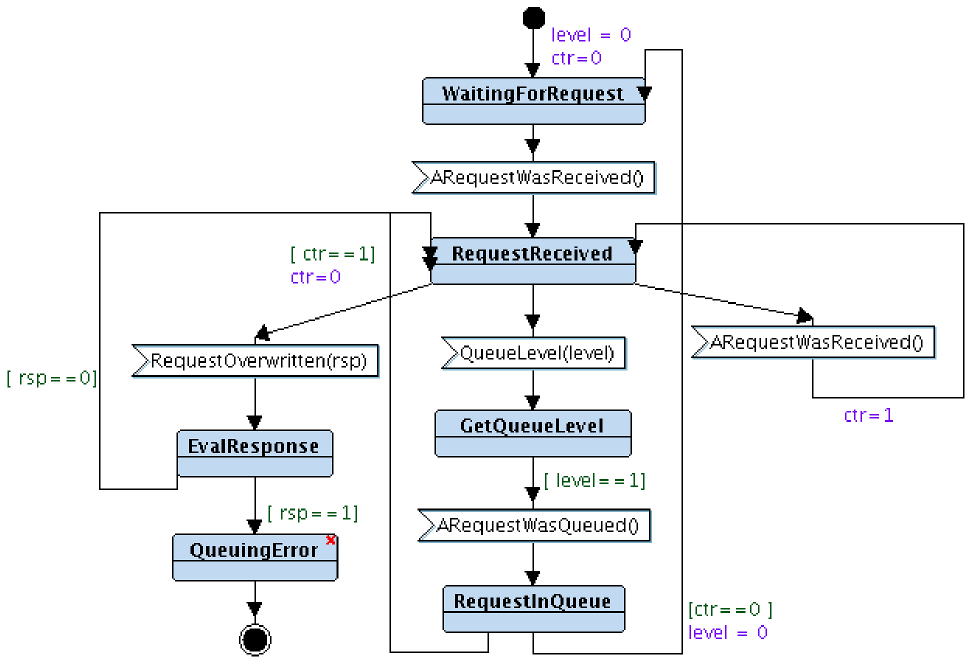


Figure 12 SMD of an Observer verifying correct queuing of requests

3.4 Conclusions

LLD is a critical component of the EVITA architecture and the tie between middleware applications and HW resources layer. Indeed, LLD provides an interface to directly interact with HSM resources. Consequently, Driver reliability is a major requirement in the scope of the EVITA prototype architecture. This section described a methodology to verify the correctness of LLD, and thus improve reliability of their implementation.

More precisely, our approach relies on a SysML/UML modeling environment named AVATAR. AVATAR natively includes formal verification features. During the verification process applied on several driver versions, several weaknesses were identified, leading to make improvement proposals. More particularly, a race condition problem was identified when accessing shared buffers. The race condition was identified with deadlock searches and analyzing incorrect queuing of requests. After respective improvements, the model of drivers grants mutually exclusive access to shared buffers, supports parallel and sequential execution of requests and introduces and manages priorities in requests. On that last model, verification results demonstrate that all evaluated properties are satisfied. However, even if several tests and proofs were made, more complex scenarios should still be analyzed in order to better explore Driver behavior, e.g., for managing errors and values outside of specification.

4 Automatic C-Code Generation

This section presents the work achieved in the scope of the code generation for LLD. The objective of code generation is to allow to generate an executable code from a UML or SysML model in order to improve code quality. Code quality refers to software engineering quality criteria, and more particularly *robustness* in the scope of critical software. Indeed, UML or SysML models are expected to be intensively verified. Combined with a model-to-code generator, the overall approach is expected to improve the final code quality. The overall process could also be enhanced using certified formal and executable code generators. Certification is not addressed in this deliverable.
More specifically, when dealing with the EVITA project, code generation is expected to improve the quality - and therefore the security - of LLD. Modeling at a high level abstraction is commonly applied to controlling parts of applications. Previous sections have already presented the modeling of the controlling parts of LLD. This modeling activity was useful to identify LLD weaknesses, which have been addressed in a second model. Then, from that second model, three different approaches could be used:

- **Ad-hoc coding**. Recoding the LLD controller by hand, simulating and comparing traces with the ones of the reference model and then, integrating missing elements, that is, the data manipulation part of drivers.
- **Code generation**. Generating the LLD controller automatically, and then enhancing the generated code with data manipulation.
- **Enhanced model and full code generation**. Enhancing models with data manipulation, and generating a fully functional version of LLD.

The section interest is clearly on the second and third options. They are both detailed hereafter.

4.1 Code Generation

The late availability of drivers conducted us to implement a generic code generator from AVATAR models. By "generic", we mean that the generated code is not specific to a given class of applications (e.g., drivers), but simply relies on a main entry point - a *main*() function - that relies on the POSIX interface to implement the model. This code is meant to be linked with an *AVATAR library* (also called *AVATAR Runtime*).

4.1.1 AVATAR to C/POSIX

Basically, the C/POSIX code generator of TTool works as follows:

- One *.c* and one *.h* files are generated for each block: they contain a representation of the State Machine Diagram (SMD) of the corresponding block. The translation of operations on variables, method calls and tests is quite straightforward. On the contrary, synchronous data exchange, asynchronous data exchange, and time manipulation are more complex and are thus handled by the AVATAR library (i.e., the AVATAR runtime)

- The main file (*main.c*) is in charge of defining one thread per block, setting the attributes of those threads (e.g., on which CPU each thread must be executed, which scheduling policy to use, etc.), starting all threads, and finally waiting for their termination

4.1.2 AVATAR Runtime

The AVATAR runtime is a set of libraries that handle all synchronous and asynchronous communications between blocks. Basically, it relies on data structures to store requests from blocks, and on mutex and condition variables to achieve necessary synchronization between threads of blocks. Its implementation is lightweight (about 2000 lines of C code). The AVATAR runtime is automatically linked against the generated code when compiling the latter.

4.1.3 Use of the Generated Code

The generated code can be used in three ways:

- **On the local platform**. The code is generated, compiled for the local platform, and executed. This step is particularly useful for fast debugging purpose. This is the only option when the target platform has not yet been defined.
- **On a prototyping platform**. If the target platform is defined but not yet available, a prototyping platform can be used to evaluate the performance of the overall system (code + OS + HW target).
- **On the target platform**. In the scope of LLD, it means executing the Driver on the Tricore environment.

All these usages of the code generator are intended to be applied on refined models. A refined model is a model in which some abstractions of a more abstract model have been resolved. For example, AVATAR designs make it possible to abstract algorithms with their estimated durations: a *computeFor(minDuration, maxDuration)* can be added to state machines transitions. Another example of abstraction is to let branches of choices undetermined, that is, at a high level of abstraction, all branches of choices may be considered. At formal verification level, this means that all branches have to be explored. But on a more refined model, branches of a choice are not randomly taken, but they are usually rather selected according to the result of previous computations. Finally, abstractions shall be resolved before generating code.

To do so, an AVATAR user could use the AVATAR state machines to put more information in its model. Unfortunately, when coming to complex algorithms - e.g., in our case, cryptographic algorithms - , a graphical model based on state machines is not practical. Therefore, the best option is probably to directly replace given elements of an AVATAR design with its corresponding implementation code, e.g. replacing a *computeFor(minDuration, maxDuration)* by the C algorithm it models. If this C code included into the model is actually ignored by the integrated simulation and formal verification capabilities of TTool, this code can be automatically included in the POSIX/C code generated by TTool.

Finally, the most abstract AVATAR models performed with TTool generally represent the control part of applications, and are thus often amenable to simulation and formal verification. On the contrary, more refined models resolve non determinism behaviors with low-level representations (e.g. in C) of data and algorithms. Those refined models are not amenable to simulation and formal verification, but are definitely useful for prototyping purpose.

4.1.4 Implementation and Results

The AVATAR-to-C code generator we implemented in the scope of EVITA is already integrated in the latest beta release of TTool [10]. This code generator has been successfully tested on several AVATAR models. Moreover, this work has been published and presented at SAME'2011 [1], and will be published and presented at ERTSS'2012 [2]. In the scope of those research works, the code generator has been applied to an academic case study (an e-reader application) and to the emergency braking system of EVITA described in D2.1 [8], respectively.
For the second application, it has also been prototyped on the virtual prototyping platform SocLib [15] which is now linked to TTool. This prototyping phase, as implemented in TTool (see Figure 13), works as follows:

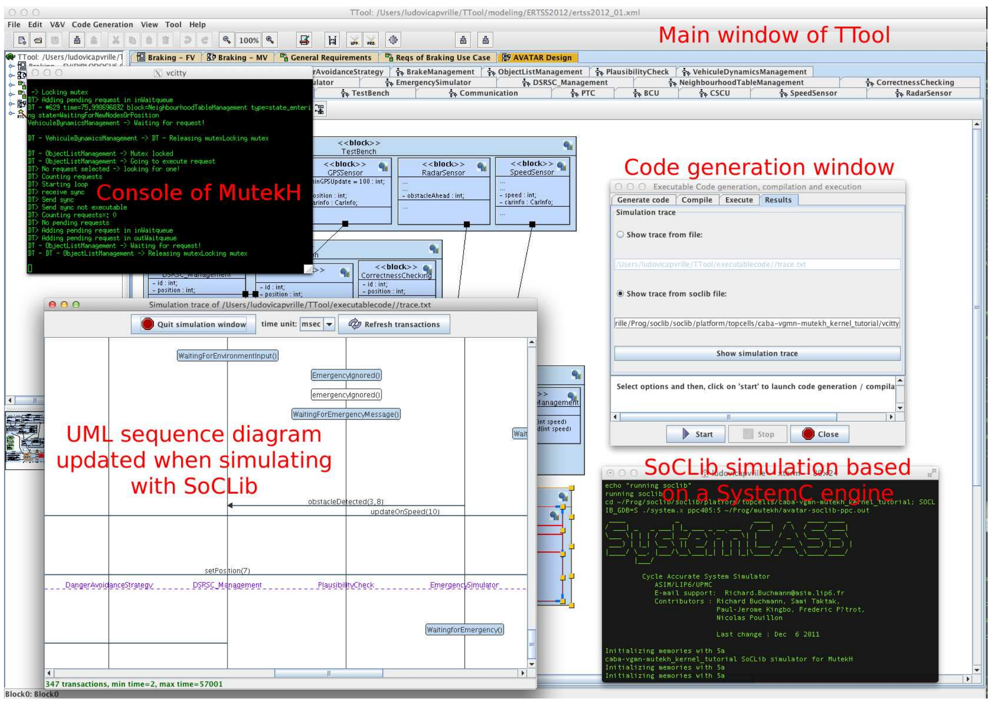
1. **Generation of the cross-compiler**. A cross-compiler for the target platform must be generated. In our example, we have used *gcc*-based cross-compilers. In the scope of our example, we have prototyped the active braking application on various 32-bit processor architectures, including PowerPC, Arm, Mips and Sparc. The Tricore environment is not available in SoCLib.

2. **Generation of the C/POSIX code**. From an AVATAR model in which non deterministic behaviors have been resolved (ideally), TTool generates a set of *.c* and *.h* files, as previously explained. The main file describes how threads are mapped on the different CPUs of the hardware architecture. In the scope of LLD, only one CPU would be used.

3. **Compilation of the code**. The generated C code, the AVATAR runtime, and MutekH [11] are compiled with the cross-compiler, and linked together as one executable file. The executable file could obviously be run on the real hardware or a virtual prototyping platform built using SoCLib. Another operating system could obviously be used instead of MutekH.

4. **Prototyping with SoCLib**. The SoCLib simulator is started with the desired hardware configuration which runs the executable file generated at previous step. The active braking application has been tested on several processors: *PowerPC*, which are commonly used in automotive systems, but also *Mips*, *Arm* and *Sparc* processors

5. **Results analysis**. Results of the prototyping simulation can be visualized either in the console, or directly in TTool as a UML sequence diagram (see Figure 13). Indeed,



```
 -> Locking mutex
DT> Adding pending request in inWaitqueue
DT - #629 time=75.998696832 block=NeighbourhoodTableManagement type=state_enteri
ng state=WaitingForNewNodesOrPosition
VehiculeDynamicsManagement -> Waiting for request!

DT - VehiculeDynamicsManagement -> DT - Releasing mutexLocking mutex

DT - ObjectListManagement -> Mutex locked
DT - ObjectListManagement -> Going to execute request
DT> No request selected -> looking for one!
DT> Counting requests
DT> Starting loop
DT> receive sync
DT> Send sync
DT> Send sync not executable
DT> Counting requests=: 0
DT> No pending requests
DT> Adding pending request in inWaitqueue
DT> Adding pending request in outWaitqueue
DT - ObjectListManagement -> Waiting for request!
DT - DT - ObjectListManagement -> Releasing mutexLocking mutex
```

```
echo "running soclib"
running soclib
cd ~/Prog/soclib/soclib/platform/topcells/caba-vgmn-mutekh_kernel_tutorial; SOCL
IB_GDB=S ./system.x ppc405:5 ~/Prog/mutekh/avatar-soclib-ppc.out

        Cycle Accurate System Simulator
           ASIM/LIP6/UPMC
           E-mail support:  Richard.Buchmann@asim.lip6.fr
           Contributors : Richard Buchmann, Sami Taktak,
                          Paul-Jerome Kingbo, Frederic P?trot,
                          Nicolas Pouillon

                          Last change : Dec  6 2011

Initializing memories with 5a
caba-vgmn-mutekh_kernel_tutorial SoCLib simulator for MutekH
Initializing memories with 5a
Initializing memories with 5a
```

Figure 13 Prototyping environment based on TTool, SoCLib and MutekH

the generated code might be instrumented so as to contain model references. The GNU debugger *gdb* can also be used to have more information about the execution of the code, e.g., about memory allocations, to perform step-by-step execution, to monitor which threads are currently executing, etc. From simulation traces, important prototyping information can be obtained. For instance, in the active braking application of EVITA, the latency between the receiving of an emergency message and the corresponding braking action can be clearly evaluated for each processor type. The same evaluation could be applied for the LLD.

4.1.5 Applying the Code Generator to LLD

Once again, the implemented code generator generates a stand-alone application. Currently, a stand-alone application can thus be generated from the LLD model presented at previous section, and execution traces can be analyzed at UML level. The code has been generated from the whole model, that is, it contains a representation in C of layers communicating with the driver: the driver test bench (i.e., blocks named "TESTCASE" in the model), and the underlying layers with which the LLD interacts, that is the SPI and the HSM.

Figures 14 and 15 present an excerpt of traces we obtained when executing the code generated from the driver models on a Linux machine. The trace in the first Figure 14 corresponds to a request sent from a testcase block to the LLD. The request is subsequently added to the list of requests of the driver. The second trace in Figure 15 corresponds to the communication between the core of the driver with the underlying SPI layer. Those two traces were obtained using an instrumentation of the code, and converting traces

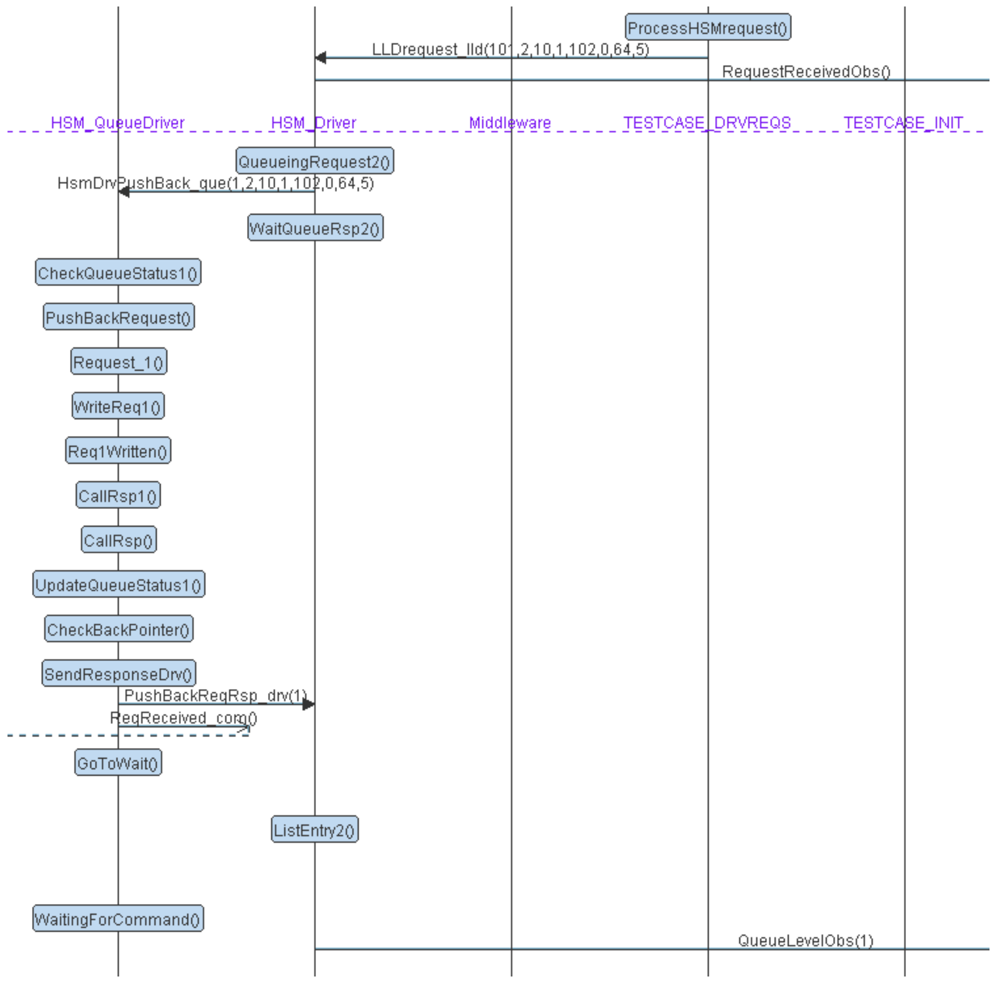


Figure 14 Excerpt of execution traces of LLD: request to the LLD

produced by the code into a format corresponding to the one used at modeling level, thus allowing the debugging directly at model-level. Thus, TTool now implements the code instrumentation and the displaying of code execution directly at UML level.

4.1.6 Adaptation of code Generation for Drivers

As explained in the previous subsection, a few adaptations are still necessary to make this automatically generated code directly executable as a driver, since currently, an application is generated, and definitely not a driver.
A driver generally handles communications with either the upper software layer or with the controlled hardware.

- The upper layer usually relies on an interface to communicate with the driver. In UNIX-like systems, this interface relies on a set of standard functions, the main of which are: *init*(), *open*(), *read*(), *write*(), *ioctl*(), *close*(). This Application Programming Interface (API) applies to most Linux drivers.
- The driver is informed by the controlled device of the completion of an operation or

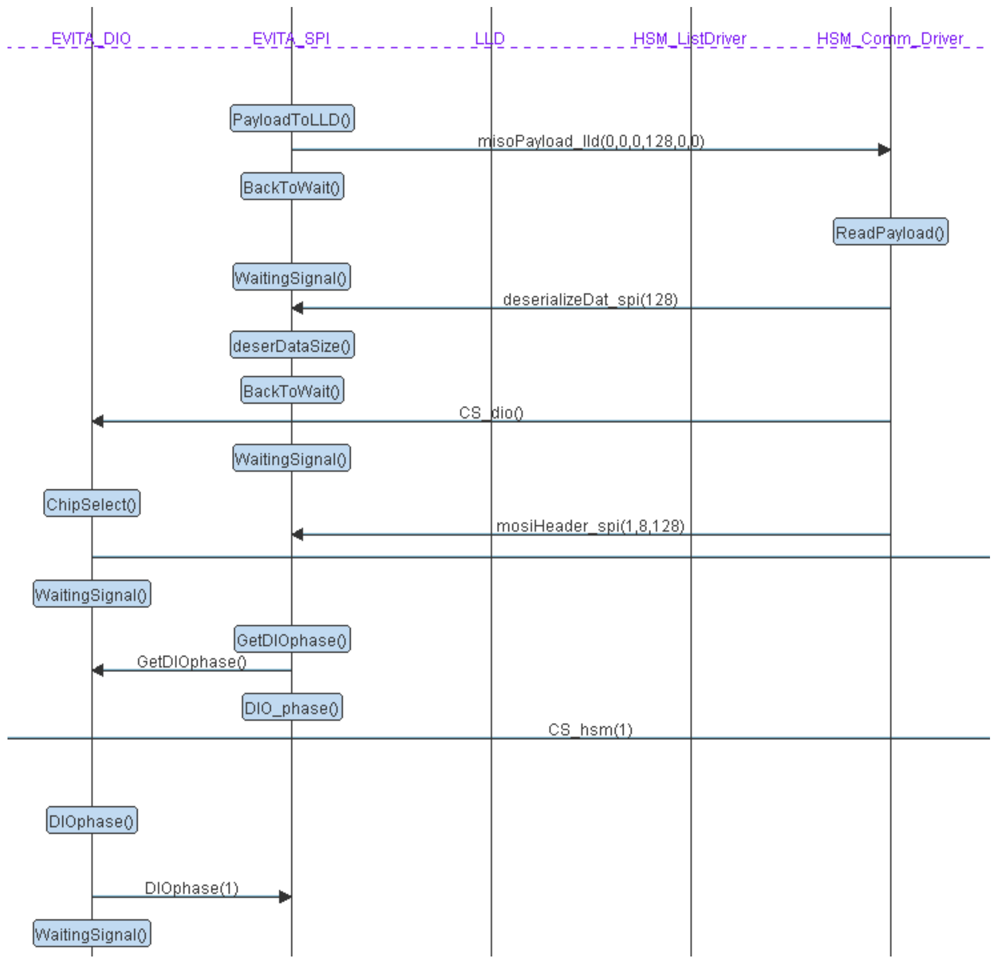


Figure 15 Excerpt of execution traces of LLD: communication between LLD and SPI

of a data availability either with an interrupt, or reading regularly a specific register of the controlled hardware (polling technique).

The EVITA LLD is informed by the underlying device with interrupts. From the upper layer, a specific API can be used to send requests to the HSM. The result is obtained using a callback approach. Due to the late release of drivers and execution environment (e.g., Autosar), we were not able to adapt the implemented AVATAR code generator, but we do provide a few hints on how it could be adapted to suit the EVITA needs.

An AVATAR model is built upon a set of blocks. Currently, each AVATAR block is translated as a (POSIX) thread, and can wait for the receiving of events. However, the AVATAR system is self-contained, and therefore it cannot wait for external calls, e.g. *function calls* or *interrupts*: this is the main facility we would need to add to AVATAR models. The code generator would obviously need to be adapted as well. We are assuming that drivers implementation can rely on the thread interface so as to reduce modifications on the code generator. Finally, here are the main modifications to perform on AVATAR and its code generator:

- **Enhancing models with functions and external signals**. The code that needs to be generated for the interface of the driver must support the call of functions. In AVATAR, only internal signals are currently supported, that is, when a signal is received by an AVATAR block, the latter executes in its thread the corresponding actions. A driver does not work that way: when a call to a driver function is performed, actions corresponding to that call are realized inside of the caller's thread: that scheme is not supported in AVATAR. Therefore, the code generated for the interface of a driver with the upper layer shall not be generated as a thread, but as a collection of functions.

  This generation of functions could be done as follows. An AVATAR signal might be defined as *external*, and does not need to be connected to another block. Each block that does not correspond to a thread - but to a collection of function, i.e., an interface - shall be tagged with the keyword "interface". Then, all blocks tagged with "interface" can have specific state machines states also tagged with "interface". All external signals that can be received from those states are considered as functions of the drivers. For example, if a state tagged "interface" and called "WaitingforUserInput" waits for two external signals *open* and *close*, it means that this system has two functions *open*() and *close*() that can be called by external threads. The same approach could be used for interrupt using external signals, and blocks and states tagged with "interrupt".

- **Enhancing the code generator with functions and interrupts**. All blocks are generated as threads, except the ones with tagged "interface" or "interrupt". For the latter, only a set of C functions are derived. The prototype of those functions is the one of external signals that can be received from "interface" and "interrupt" states. Code generated for interrupts will have to be adapted to the local platform since that kind of code is very specific to the hardware platform. However, the controlling part of interrupt code will be generated from the AVATAR model.

4.2 Full Code Generation

The code generation approach presented at previous section takes the assumption that the code is first generated from the model, and then the latter is enhanced by hand, more probably with definitions and manipulations of data. Indeed, the model mostly contains controlling parts of applications, and so, of drivers. A way to avoid this "by hand" step would be to put all information directly in the model. In that case, models would have to be annotated with additional information. We think that one main enhancement could resolve that issue: putting references to functions on state transitions:

- Functions are first declared and implemented in external C and H files. This function are most likely to to declare complex data structures not present in the model, and various functions to initialize and manipulate them
- Calls to functions are added at state machine level. On the transition exiting the "start" state of state machines, there will probably be a call to a function that initializes data not present in the model. Anyway, all state transitions can be enriched with functions calls. The formal verification step totally ignores those additional information
- The code generator works the same way as before, apart from the fact that calls to those new state transition functions are added in the code
- The code can then be compiled and further linked with the implementation of external functions defined in the first step

4.3 Conclusion

In the scope of the EVITA project, a model-to-C code generator has been specified, implemented and tested. In particular, the code generator has been tested for automotive applications, and for complex models. Its implementation is now part of the latest releases of TTool.

Yet, the defined and implemented code generator shall be enhanced to support explicitly the particularities of drivers, in particular, requests to drivers and interrupts. A first proposal for handling these features has been provided, but is not yet implemented.

The environment we settled targets in particular the improvement of code quality. This point has still to be compared with regards to the manual approach.

5 Conclusions and Future Work

To complete the EVITA Description of Work and Annex I, the following technical work was achieved by *Institut Télécom* as part of T4200:

- An adequate environment has been proposed for the modeling and formal verification of drivers. The environment we develop offers a high-level language, SysML, and integrates formal verification techniques that can be applied directly from the model, without any specific knowledge. EVITA LLD was successfully modeled and verified. An important weakness has been identified and accordingly corrected. The overall approach has been implemented in TTool [10].
- A code generation scheme has been proposed to be able to generate part of the LLD code from a high-level environment. A first code generator has been defined and implemented in TTool. The code generator has been experimented for several applications, including the LLD. Currently, it still needs to be enhanced since the code generator generates a stand-alone application that can therefore not interact with its environment. In particular, the interrupt issue should be addressed in the scope of drivers.

Thus, the overall approach is expected to have a final version of LLD better respecting its specification while being more reliable.

References

[1] L. Apvrille and A. Becoulet. Fast and multi-platform prototyping of embedded systems from uml/sysml models. In *The 14th edition of the Sophia Antipolis MicroElectronics Forum (SAME'2011)*, Sophia-Antipolis, France, October 2011.

[2] L. Apvrille and A. Becoulet. Prototyping an embedded automotive system from its uml/sysml models. In *ERTSS'2012*, Toulouse, France, February 2012.

[3] L. Apvrille, J.-P. Courtiat, C. Lohr, and P. de Saqui-Sannes. TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit. In *IEEE transactions on Software Engineering*, volume 30, pages 473–487, Jul 2004.

[4] 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.

[5] C. Fischer, F. Pirklbauer, and M. Mittendorfer-Holzer. HSM Low Level Driver Specification. Technical Report Deliverable D4.2.2, EVITA Project, 2011.

[6] A. Fuchs, S. Gürgens, L. Apvrille, and G. Pedroza. On-Board Architecture and Protocols Verification. Technical Report Deliverable D3.4.3, EVITA Project, 2010.

[7] T. Gendrullis, M. Wolf, and H. Platzdasch. Hardware Implementation Specification. Technical Report Deliverable D4.1.1, EVITA Project, 2011.

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

[9] D. Knorreck, L. Apvrille, and P. De Saqui-Sannes. TEPE: A SysML language for timed-constrained property modeling and formal verification. In *Proceedings of the UML&Formal Methods Workshop (UML&FM)*, Shanghai, China, November 2010.

[10] LabSoc. TTool. In *http://ttool.telecom-paristech.fr*.

[11] LIP6. Mutekh. http://www.mutekh.org.

[12] OMG. UML 2.0 Superstructure Specification. In *http://www.omg.org/docs/ptc/03-08-02.pdf*, Geneva, 2003.

[13] G. Pedroza, L. Apvrille, and D. Knorreck. AVATAR: A SysML Environment for the Formal Verification of Safety and Security Properties. In *11éme Conférence Internationale sur les Nouvelles Technologies de la Repartition, NOTERE-2011*, Paris, France, May 2011. IEEE.

[14] G. Pedroza, M.S. Idrees, L. Apvrille, and Y. Roudier. A Formal Methodology Applied to Secure Over-the-Air Automotive Applications. In *74th Vehicular Technology Conference: VTC2011-Fall*, San Francisco, USA, September 2011. IEEE.

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

[16] B. Weyl, M. Wolf, F. Zweers, T. Gendrullis, M. S. Idrees, Y. Roudier, H. Schweppe, H. Platzdasch, R. El Khayari, O. Henniger, D. Scheuermann, L. Apvrille, G. Pedroza, H. Seudié, J. Shokrollahi, and A. Keil. Secure On-board Architecture Specification. Technical Report Deliverable D3.2, EVITA Project, 2010.