Model Interpretation for an AUTOSAR compliant Engine Control Function

Sakthivel Manikandan Sundharam
University of Luxembourg
FSTC/Lassy
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg
sakthivel.sundharam@uni.lu

Sebastian Altmeyer
University of Amsterdam
CSA Group
Science Park 904
1098XH Amsterdam
altmeyer@uva.nl

Nicolas Navet
University of Luxembourg
FSTC/Lassy
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg
nicolas.navet@uni.lu

***Abstract*—Model-Based Development (MBD) is a common practice in the automotive industry to develop complex software, for instance, the control software for automotive engines, which are deployed on modern multi-core hardware architectures. Such an engine control system consists of different sub-systems, ranging from air system to the exhaust system. Each of these sub-systems, again, consists of software functions which are necessary to read from the sensors and write to the actuators. In this setting MBD provides indispensable means to model and implement the desired functionality, and to validate the functional, the non-functional, and in particular the real-time behavior against the requirements. Current industrial practice in model-based development completely relies on generative MBD, i.e., code generation to bridge the gap between model and implementation. An alternative approach, although not yet used in the automotive domain is model interpretation, the direct interpretation of the design models using interpretation engine running on top of the hardware. In this paper, we present a case study to investigate the applicability of model interpretation, in contrast to code generation, for the development of engine control systems. To this end, we model an engine cooling system, specifically the calculation of the engine-coolant temperature, using interpreted model based development, and discuss the benefits and low-lights compared to the existing code-generation practice.**

I. Introduction

Model-Based Development (MBD), also frequently referred to as Model-Driven Engineering (MDE), denotes the use of models as the main artifacts to drive the development of systems. It has been profoundly reshaping and improving the design of software-intensive embedded systems specifically. Traditionally, model-driven development (based on code generation) is deployed in the automotive industry. Code generation is used to generate code from a higher level model and create a working application.

As mentioned in [4], Model-Based Development is being used for series development by a majority of the automotive companies. Especially in development phases i.e., system design and coding, the model-based design is used extensively. As mentioned in [7], this kind of MBD used by automotive suppliers and car manufacturers is called *generative MBD*, since code and other artifacts are automatically generated from the model.

The other fundamental approach to achieve applications from models is *interpreted MBD*. Interpreted MBD can be seen as a set of platform independent models that are directly interpreted by an execution engine running on top of the hardware, with or without an operating system.

The fact that models can be directly executable helps a great deal as the development cycle time can be shortened; and there is no distortion between the model and what is executed. Though, to the best of our knowledge, the technique of model interpretation remains unexplored in the automotive domain, it can facilitate and speed up the development, deployment and timing verification of applications with real-time constraints running on potentially complex hardware platforms. Verification also can be done more easily as defects will be caught earlier in the process since there is no difference between the model and the executable program. In this paper, we present a case-study to evaluate how *interpreted MBD* can be applied to an automotive software development scenario.

This paper is structured as follows. In Section II, we explain the state of the industrial practice of automotive function development. Section III describes an AUTOSAR-compliant engine-coolant temperature calculation function used as case-study. In Section IV, we discuss our modeling approach, and Section V presents the case study. Finally, Section VI summarizes the results and discusses the case study. Section VII concludes the paper.

II. Automotive Function development - state of the practice

We explain the state-of-the-art of the development of an automotive function using an automotive engine management software system, which are commonly developed using a Model Based Development (MBD). The engine is controlled by an Electronic Control Unit (ECU) that contains engine functions for different sub-systems.

The requirements of the engine functions are specified in one of the Application Life-cycle Management (ALM) suites and traced until its realization as ECU. In ALM, different tools are integrated to develop and maintain the software. For example, IBM has an ALM suite called IBM Rational Team Concert (RTC) where Rational DOORS is the requirements management tool that captures all the functional and non-functional requirements. These requirements are analyzed further to design the engine function. Popular Model Based

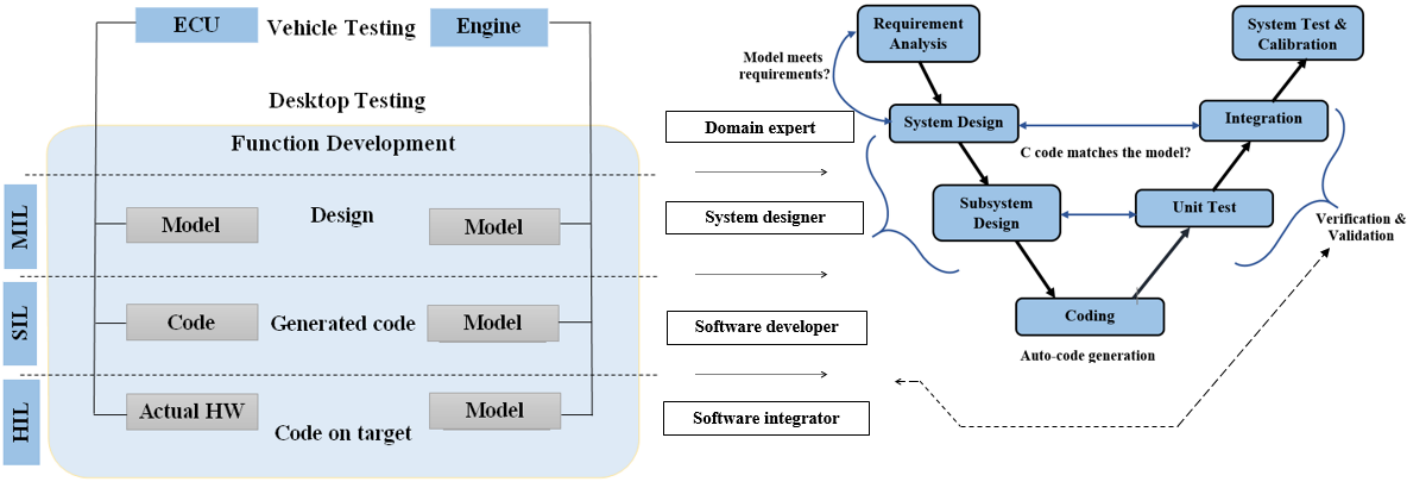

Fig. 1. Engine function development flow - Illustration of verification techniques, involved stakeholders and development phases.

Design (MBD) tools are MATLAB/Simulink (MLSL) from Mathworks, ASCET-MD from ETAS, and SCADE Suite from ANSYS. These industrial MBD tools further generate code for engine functions using code generators. Each engine control function is further (unit-) tested and integrated into the ECU.

Figure 1 shows the software function development flow practiced in the automotive industry. The system model of the engine captures the ideas and requirements. The model is an executable specification and can be simulated and rapid-prototyped to explore different design options. In the existing approach, the modeling environment is primarily used to describe the domain problem, in this case the engine function to be developed against the functional requirements. *Domain experts* and *software designers* are involved in this phase. The controller model is tested in a simulation environment (which includes the plant model, *i.e.* the engine) and this testing is called *Model-in-the-Loop* (MiL) testing to ensure that the model meets the requirements.

In the next step, the code is generated from the model using a code generator. Then, the code is verified under an engine model. This phase is referred to as *Software-in-the-Loop* (SiL) testing. *Software developers* are involved to test each engine function individually using unit testing. Next, the function is integrated with other existing engine functions in the integration phase by the *Software integrators*, typically a tier-one supplier. The complete engine software is then ported to the ECU hardware, which can be verified using a *Hardware-in-the-Loop* (HiL) testing system, such as PT-LABCAR, which realistically emulates vehicles I/Os.

In the current practice [3], the execution environment on the target is different from the execution within the modeling environment in terms of I/Os, scheduling and even in terms of generated code. Indeed, the target-generated code will be optimized towards the platform and thus be as efficient as possible. On the negative side, the build tool-chain must be available, and it takes a substantial amount of time to produce an executable program from the designed model (build time can require several 10s of minutes). Simulink and its block sets (like Simscape, Stateflow etc.) are examples for modeling environment and Embedded Coder is an example of the code generator for production code generation on a specific target processor. The generated code can be further customized to meet the requirements (e.g., with respect to safety). In the automotive software development, there is a high probability for mixed-mode development, where generated code is integrated with manually-developed functions.

III. AUTOSAR-compliant Engine function

The engine cooling system is an important part of the vehicle. It is responsible for maintaining optimum operating temperature. The coolant is circulated through the engine block with the help of an electric water pump. The coolant will reduce the temperature of the engine block and then will run through the radiator equipped with a fan to remove waste heat.

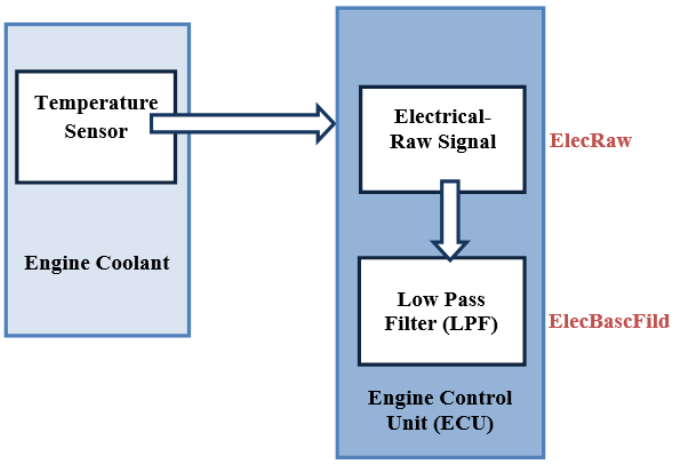

Fig. 2. Physical layout of an AUTOSAR compliant engine-coolant system function - Engine coolant temperature sensor connected to an ECU

Figure 2 shows the physical layout of the engine-coolant temperature calculation which is considered as the use case to

present our modeling approach. The engine-coolant temperature sensor plays an indispensable role in the engine cooling system. Precise information about the temperature is essential due to various reasons: the data are used by the engine control unit to adjust the fuel injection and ignition timing. Further, the temperature value is used to control the cold starting of the engine, to control the calculation of the fuel quantity, and to control the fan speed of the electric cooling radiator. This data is also used to provide readings of the coolant temperature gauge to the dashboard to protect the engine from over-heating.

The engine-coolant temperature sensor is connected to the engine ECU through an analog to digital pin. The electrical output is obtained from the sensor that monitors the temperature of the engine-coolant. As per AUTOSAR design pattern [2] catalogue for standard sensors, the overall system consists of 3 modules as depicted in Figure 3. *Sensor/Actuator Components* are special AUTOSAR software components which encapsulate the dependencies of the application on specific sensors or actuators. The AUTOSAR architecture takes care of hiding the specifics of the micro-controller (this is done in the micro-controller abstraction layer, MCAL, part of the AUTOSAR infrastructure running on the ECU) and the ECU electronics (handled by the ECU-Abstraction layer, also part of the AUTOSAR Basic Software).

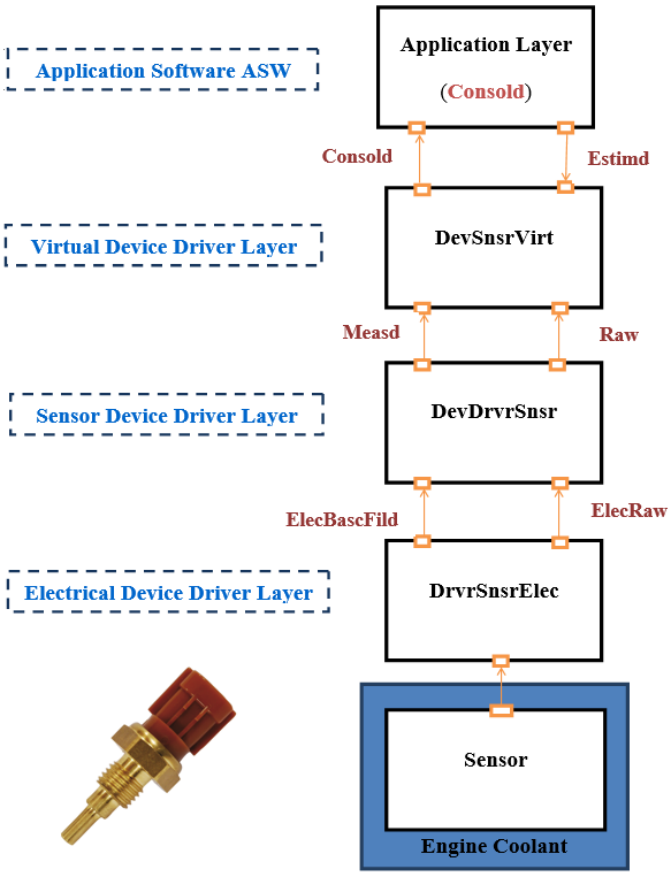

Fig. 3. AUTOSAR design pattern for a standard sensor

The architecture of the engine-coolant temperature calculation function involves 3 AUTOSAR software components:

Electrical Device Driver Layer (DrvrSnsrElec):
The electrical value from the temperature sensor is read through the input pin and stored in the variable *ElecRaw*. The raw electrical signal (*ElecRaw*) is rugged against signal faults using the Low Pass Filter (*LPF*) and the filtered raw electrical signal (*ElecBascFild*) is obtained.

Sensor Device Driver Layer (DevDrvrSnsr):
At this stage, the raw electrical signal is converted into its physical temperature value (*Raw*) using a lookup-table, where the corresponding value is provided. The temperature value of the filtered electrical signal (*ElecBascFild*) is also obtained from the lookup-table and is provided to the next layer.

Virtual Device Driver Layer (DevSnsrVirt):
In this layer, the possible signal range check, electrical errors, cable interruption and sensor faults that may occur are identified. This is done in order that incorrect values from the sensor are not taken into account for the calculation in case of sensor malfunctioning. Other errors such as a cable interruption, short circuit to battery or sensor voltage saturation can also be detected and appropriate flags will be set:

- *ElecBascFildbit* - The electrical validity bit shows that the sensor raw value is electrical valid.
- *ElecBascFildbitCommon* - The common validity bit shows that the engine-coolant temperature as a whole is valid and can be transfered to the application Layer. Based on the temperature values calculated in this layer, the obtained temperature value (*Measd*) is compared with the estimated value (*Estimd*) from the application layer. This comparison determines the validity of the calculated value. If valid, the final temperature value (*Consld*) is sent to the application layer.

IV. Function development - Proposed approach

To the best of our knowledge, model interpretation for automotive function development has not been explored and experimented in the past. In case of model interpretation, a generic model-interpretation engine is implemented which executes the model of the engine function. As shown in Figure 4, the modeling environment includes the execution environment. Hence, the executable artifacts (*i.e.*, model and execution engine) are available within this environment. The model interpretation can be launched within the development environment or on a target platform. In the latter case, the interpretation can run on top of an OS or directly on the hardware. There are two possible interpretation modes: simulation and real-time. Simulation mode is suited for the use in the design phase, where execution should be as fast as possible, which implies that the activation frequencies of the processes

are not respected and they execute (conceptually) in zero time. Typically, executing in simulation mode is several orders of magnitude faster than in the real-time mode. Real-time mode is for the execution of the program with the actual desired temporal behavior of the application.

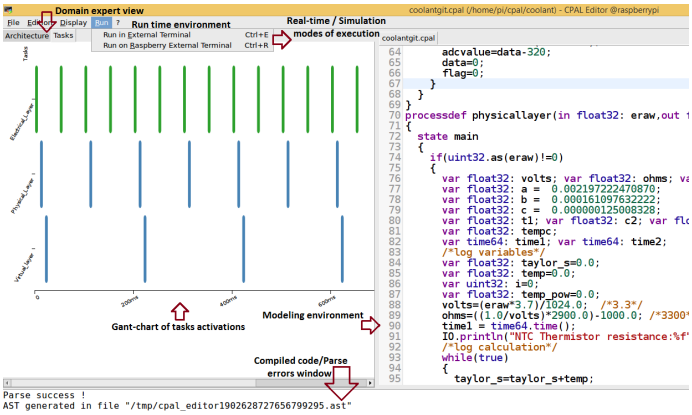

Fig. 4. An integrated environment, here the CPAL-Editor, with the code of the model, the Gantt chart of the processes activations and the possibility to execute the models in simulation and real-time mode both locally or on a target.

To ensure that simulation reflects the real-time behavior on the target platform, timing annotations (e.g., execution time latencies, jitters, etc) can be introduced in simulation mode. Those timing annotations can be derived from measurements on the target architecture, from WCET analysis and, possibly, by schedulability analysis if other software components can interfere with the function under development. Timing accurate simulation thus provides benefits to identify faults in design phase itself, earlier, thus than with the traditional design process.

As the model itself can be executed, no additional artifacts are needed, and, unlike in the traditional generative MBD, no target specific code is generated. Instead, the specifics of the platform are taken care by the interpretation engine. Further steps of the application development, such as compilation of source code to object code and the linking stage to produce the executable program, are not required.

V. A Case Study - Engine-coolant temperature calculation

The model of the engine-coolant system is developed in the CPAL (Cyber Physical Action Language, see [1, 6]), which is a new language to model, simulate, verify and program Cyber Physical Systems. CPAL[1] is a language jointly developed by our research group at the University of Luxembourg and the company RTaW. Many industrial use-cases are demonstrated [5] using CPAL in the past.

The model-based environment of CPAL consists of a single integrated development environment, i.e., the CPAL-Editor. The CPAL editor, combines the design, simulation, execution (both locally and on a target), visualization of the functional architecture and execution chronogram in one integrated environment. The model-interpretation engine is specific to the target platform. This interpretation engine can be executed on top of an operating system or without an operating system, the latter being called Bare-Metal Model Interpretation (BMMI). CPAL BMMI is available on the NXP Semiconductors Freedom-K64F, a low-cost development platform which is form-factor compatible with the Arduino R3 pin layout. The experiments in this study are performed on a Raspberry Pi equipped with a multi-core ARM Cortex-A7 processor operating at 900 MHz running Raspbian OS.

A typical engine-coolant temperature sensor can measure in the range −40°C to +150°C. In our case study, we have considered a Negative Temperature Coefficient (NTC) type sensor with an operating voltage as 3.3V. Figure 5 shows the experimental setup which aims to mimic the engine cooling system. The MCP3008 is an external ADC interface which is connected to the sensor. Since the sensor operates with the thermistor principle, a voltage divider circuit with 3.3V reference is added. ADC data from MCP3008 is communicated to the processor using the Serial Peripheral Interface (SPI). The sensor software component is modeled according to the AUTOSAR design catalog described in Section III. The speed of the electric fan is controlled based on the measured temperature.

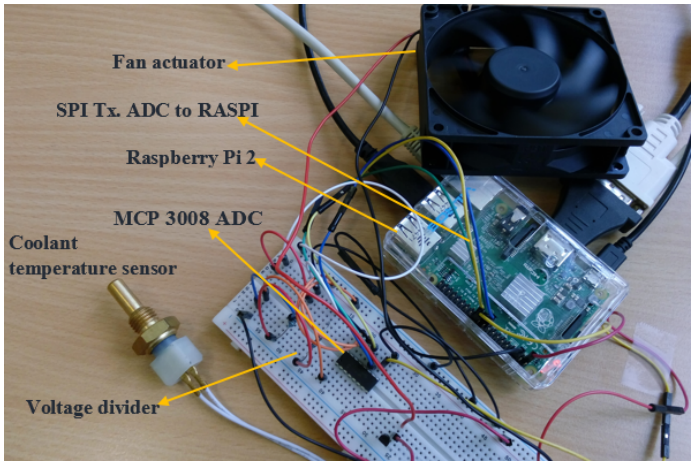

Fig. 5. Experimental set-up - Sensor interfacing to hardware

Out of the two possible CPAL execution environments (*i.e.*, bare-metal or hosted by an OS), we use the interpretation engine on top of an OS (Raspbian on Raspberry Pi) which can also execute in real-time, although with a lesser real-time predictability than the bare-metal implementation. The engine-coolant temperature is calculated by the sensor software component modeled in CPAL. Figure 6 shows the sample run-time environment where simulation and real-time execution are performed. Both interactive and non-interactive executions are possible. The interactive mode of execution is useful in program analysis and debugging. In interactive mode, the user has different execution options, such as a step-by-

[1]The CPAL documentation, graphical editor and the execution engine for Windows, Linux and Raspberry Pi platforms are freely available from http://www.designcps.com.

step execution, or uninterrupted execution for a pre-defined duration.

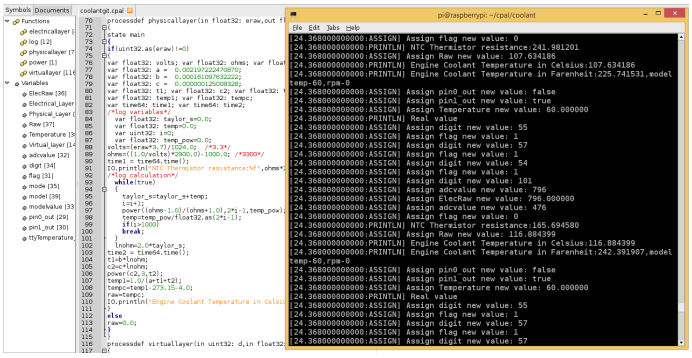

Fig. 6. CPAL model and execution environment under real-time mode

Since it is an interpretation-based execution environment, the user can list and change the values of global variables at run-time, as well as execute additional code statements. In non-interactive mode, the program is executed indefinitely or for a specified duration without requiring additional user inputs.

VI. Results and Discussions

From the case-study experience, we present our proposed development flow for function development. Figure 7 shows the development flow of model interpreted approach to develop an engine function.

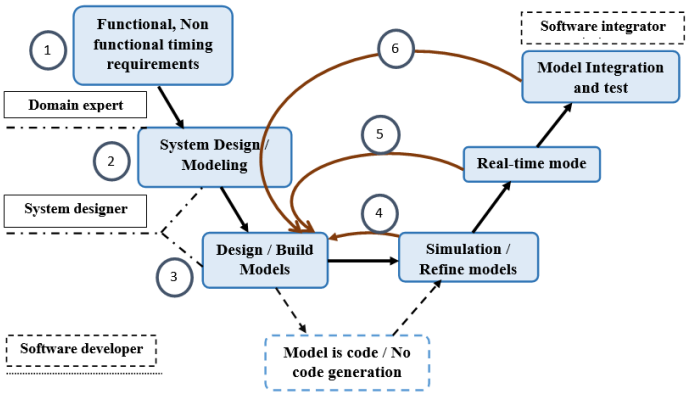

Fig. 7. Model interpreted engine function development flow - steps and stakeholders involved

a) Model interpreted development steps: In the first step all functional, non-functional including timing requirements of the engine function are collected. These are further analyzed by domain experts. The specifications are implemented in CPAL (step 2 - system design in Figure 7). During the development, as soon as the function model is updated the functional architecture, and other views created out of the model such as execution Gantt charts, are automatically updated too (step 3) which is done in the background along with the modifications. This allows the designer to immediately visualize and understand the effects of the changes made, without the need for building the executable and running it in debug mode. The latest version of the model is always available to execute, be it in simulation mode or real-time mode, locally or on a target. Typically performed once the simulation is satisfactory (step 4), the execution in real-time mode (step 5) helps the designer to assess the performances on the target, enabling rapid-prototyping. If simulation or execution in real-time mode highlights faults, the model is refined in an iterative process. From the development of the engine-coolant temperature calculation function, we here summarize the benefits and differences against the existing generative MBD approach.

b) Adapting to requirement changes is faster: The most important benefit of model interpretation is that changes in the model do not require an explicit regeneration/rebuild/retest/redeploy step. This shortens significantly the turnaround time and, in some scenarios, the overall change management process (how changes in the requirements are implemented). Although it is not available in CPAL yet, it would be possible for models to be updated at run-time, without the need to stop the running application, hence improving productivity. Also, since no artifacts are generated, the build times can be also reduced. Depending on the specific use case, an interpreter combined with model can even require less memory than generated code.

c) Finding failures in model is easier: Failures during the testing phase, after all modules have been integrated, expose problems that are clearly in the model, since the model itself is executed. Unlike with code generation, there is no need to trace back from the generated artifacts where the failure occurred in the model, which is often hard. On the other hand, debugging models at run time is possible. Since the model is available at run-time, it is possible to debug function models by stepping through them at run-time (e.g., we can add breakpoints at the model level). When debugging at model level is possible, domain experts can debug their own models (e.g., step-by-step) and adapt the functional behavior of an application based on this debugging. This can be very helpful when, for example, complex control or data-flows are involved.

d) Portability and hardware independence: Portability is another advantage of model interpretation. An interpreter in principle creates a platform independent target to execute the model. By rewriting only the hardware-specific components, it is possible to develop an interpreter which runs on multiple platforms, as it is the case for CPAL. In case of code generation, we need to make sure we generate code that is specific to the platform. In case of model interpretation, the interpreter handles the platform-specific adaptation.

A notable advantage of the model interpretation is that it hides the complexity of the hardware platform away from the programmer making it easier to configure the run-time environment and deploy the application. Indeed, easier deployment is an important difference. When code generation is used, we often see that we need to open the generated source code in an Integrated Development Environment (IDE) to analyze the program and build it from there to create the final application.

In case of BMMI, we just have to upload the model and reset the target, or, when the interpreter is hosted by an OS, execute it within the development environment or in command-line (possibly on a target through a script). Hence, it is much easier for domain experts to deploy and test an application, instead of only modeling it.

e) Benefits of single integrated environment: The important difference between interpreted approach and generative is that domain experts and software developers can work together around a single integrated environment and on a single model. As shown in Figure 8, the integrated modeling environment provides a graphical view of the architecture of the designed function model. This model can be used by domain experts for functional analysis and verification, and by software engineers to do function development and testing from day one on.

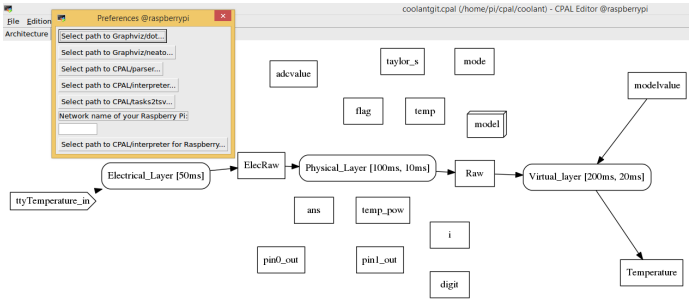

Fig. 8. Software architecture of the coolant temperature calculation

VII. Conclusions

Code generation is the standard practice in the industry for MBD of embedded systems, and this holds true in particular for engine function development. In this paper, we discuss a model-interpretation development flow that is exemplified with the development of an engine coolant temperature calculation by an AUTOSAR compliant software architecture. By comparison with the usual development chains relying on code-generation and based on the case-study, we discuss the benefits of model interpretation which includes simplicity, productivity and early-stage verification possibility, specifically in the time dimension. For instance, CPAL, the model-based development environment that we have chosen for our case study, already provides the basic mechanisms to offer timing-realistic simulation early in the design process. Our ongoing work is on a method to automate the derivation of the temporal quality-of-service required by a software module and, leveraging on model-interpretation, enforce it at run-time.

Although model-interpretation brings advantages, it is not going to cover all use-cases. The main reason is that model interpretation is intrinsically slower than compiled code. There are ways to mitigate this drawback in production code such as calling binary code from interpreted code (e.g., legacy code or specialized functions) or, possibly, selectively generating code for the computation-intensive portions of the model. Interpretation and code generation are often seen as two alternatives, not as a continuum. However, one may also imagine relying on model-interpretation, and benefits from the associated productivity gains, until the function/ECU meets all functional requirements, and then switch to code-generation for production code. This remains to be investigated in the future works.

Acknowledgment

This research is supported by FNR (Fonds National de la Recherche), the Luxembourg National Research Fund (AFR Grant n°10053122).

References

[1] S. Altmeyer, N. Navet, and L. Fejoz. Using CPAL to model and validate the timing behaviour of embedded systems. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Lund, Sweden, July 2015.

[2] AUTOSAR consortium. AUTOSAR design catalogue. http://www.autosar.org/fileadmin/files/releases/4-2/application-interfaces/general/auxiliary/AUTOSAR_TR_AIDesignPatternsCatalogue.pdf.

[3] M. Broy, M. Feilkas, M. Herrmannsdoerfer, S. Merenda, and D. Ratiu. Seamless model-based development: From isolated tools to integrated model engineering environments. *Proceedings of the IEEE*, 98(4):526–545, 2010.

[4] M. Broy, S. Kirstan, H. Krcmar, B. Schätz, and J. Zimmermann. What is the benefit of a model-based design of embedded software systems in the car industry? *Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*, page 310, 2013.

[5] L. Fejoz, N. Navet, S. M. Sundharam, and S. Altmeyer. Applications of the CPAL language to model, simulate and program cyber-physical systems. In *Demo Session of 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016)*, 2016.

[6] N. Navet, L. Fejoz, L. Havet, and S. Altmeyer. Lean model-driven development through model-interpretation: the CPAL design flow. In *Embedded Real-Time Software and Systems (ERTSS2016)*, January 2016.

[7] N. Tankovic, D. Vukotic, and M. Zagar. Rethinking model driven development: analysis and opportunities. In *Information Technology Interfaces (ITI), Proceedings of the ITI 2012 34th International Conference on*, pages 505–510. IEEE, 2012.