Targeting Different Abstraction Layers by Model-Based Design Methods for Embedded Systems: A Case Study

Omair Rafique, Manuel Gesell, and Klaus Schneider
Department of Computer Science
University of Kaiserslautern
Kaiserslautern, Germany
Email: (gesell,schneider)@cs.uni-kl.de

***Abstract*—In this paper, we show how code can be generated at different levels of abstraction from a single source description. To this end, we use a model-driven development tool called Averest that is based on a synchronous programming language. We illustrate our approach by means of a case study from the domain of distributed real-time automotive embedded systems. This paper focuses thereby mainly on the use of the Averest toolkit to generate code at different levels of abstraction.**

I. Introduction

In general, embedded systems can be modeled, developed and programmed at four different levels of abstraction, namely as pure hardware designs, as bare-iron level software (managing resources without an operating system), as real-time operating system based software (using an operating system to manage resources), and as abstract software models (not considering resources at all). These levels differ with respect to the considered details of the real embedded system [1], [2]. In general, the level of abstraction of a model is defined by the amount of information provided by that model: A low-level abstraction model provides more information, and reveals better correspondence with the hardware than a high-level abstraction model.

This paper considers a model-based design flow for embedded systems using the *Averest*[1] [5], [6] tool. Systems are thereby modeled independent of a later realization using a synchronous programming language (called Quartz). Using Averest, we generate code at different levels of abstraction from a single source description in Quartz. The complete description and the work flow on how Averest targets each level of abstraction is presented in [4]. This paper rather focuses on the application of the approach to a case study to present and demonstrate the translation of a single Quartz module to system descriptions at different levels of abstraction.

II. Related Work

This section is intended to present, review and emphasize some of the widely used model-driven development tools with respect to the features offered by these tools. However, due to lack of space, it is not possible to present a detailed survey.

Nowadays, many model-based programming tools stand firm in the market competition. Some well-known examples are: Matlab[2], IBM Rational Rose[3], Object Management Group's (OMG[4]) Model-Driven Architecture (MDA) and Systems Modeling Language (SysML[5]) etc. Matlab's model-based design supports system-level design, simulation, automatic code generation, and continuous test and verification of embedded systems. Formal verification can be performed on models designed in Matlab and Simulink using the Simulink design verifier and Polyspace software products. Similarly, the HDL coder allows one to generate code in hardware description languages (HDL) from Matlab functions, Simulink models, and Stateflow charts. The generated code can be used for prototyping and the design of field-programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs). Rational Rose supports component-based development and controlled iterative development. Models created with Rose can be visualized with several UML diagrams. It also supports round-trip engineering with several languages like C++. Similarly, OMG is promoting a model-driven approach for software development through its MDA initiative. MDA provides an open, vendor-neutral approach to the challenge of interoperability, building upon and leveraging the value of OMG's established modeling standards: UML, Meta-Object Facility (MOF), and Common Warehouse Meta-model (CWM). Platform-independent application descriptions built using these modeling standards can be realized using any major open or proprietary platform, including Java, .NET and web-based platforms. OMG's SysML is a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities.

This list could be continued with a number of further tools, each providing various distinct features. However, this paper focuses on a synchronous model-driven development tool Averest. To this end, we provide a brief introduction to the *Averest* toolkit in the next section (more information on the synchronous input language is given in [5]).

[1] http://www.averest.org

[2] http://www.mathworks.com/matlabcentral

[3] http://www-01.ibm.com

[4] http://www.omg.org

[5] http://www.omgsysml.org

Averest

As a long-term project, our group developed the *Averest* tool for HW/SW-codesign and verification of synchronous Quartz programs. It contains a compiler that computes for a given Quartz program an equivalent set of synchronous guarded actions that are stored in an *Averest Interchange Format* (AIF) file. There are several transformations available to modify a generated AIF system description. For example, the reduction of compound data types like tuples and arrays to scalar types, reduction to boolean types for hardware synthesis, the aggregation of all guarded actions on one variable into a single guarded action (so that equations are obtained), dead code elimination, the generation of an extended finite state machine (EFSM), and many more are available.

After suitable transformations, AIF systems can be converted to various target languages. For example, there are code generators for software synthesis (producing C, Java and SystemC) or hardware synthesis (producing SystemC, VHDL and Verilog files). Moreover, a simulator and a code generator for the well-known model checker SMV are also available in the *Averest* framework. As typical for a model-based design, it is to be noted here that it is possible to generate software or hardware code from the same Quartz module without any modification of the Quartz module (instead, one simply has to apply different transformations for synthesis).

III. Case Study: The ConceptCar

With the advent of programmable Electronic Control Units (ECUs) in modern cars, the idea of testing, experimenting and implementing innovative methods for the development of these embedded systems has become practical and realizable. The best evidence is the intensive research in the field of driver assistance systems. The *ConceptCar*[6] (designed and developed by our group together with Fraunhofer IESE[7]) is an experimental vehicle (remotely controlled) with the objective of testing and verifying modern future car features by deploying different classes of applications.

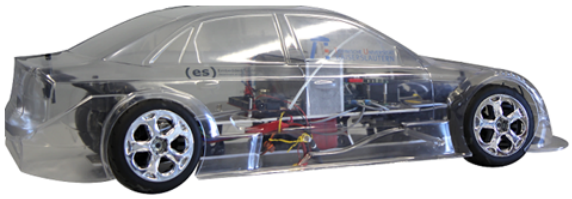

Fig. 1. The ConceptCar

Embedded systems in the ConceptCar (shown in Figure 1) have been built and engineered as close as possible to a modern car. Even though the ConceptCar is only a small radio-controlled vehicle, we can extend it with application-specific embedded systems to control various functionalities that control e.g. braking and steering. While a modern car may have up to hundred ECUs, the ConceptCar currently has 7 different ECUs (shown in Figure 2), where each ECU is responsible for a specific operation. To this end, we now present how each ECU contributes to driving and steering the ConceptCar.

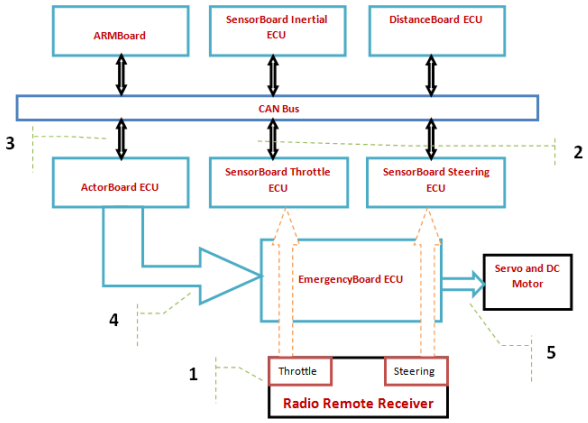

Fig. 2. Architecture of the ConceptCar

The Driving Mechanism

The two channel radio transmitter system, responsible for generating throttle and steering signals, generates the PWM signals for each channel, with 20 ms cycle length and 1-2 ms duty cycle, where 1.5 ms duty cycle represents the idle position. As shown in Figure 2, these signals are fed into the SensorBoard_Steering and SensorBoard_Throttle via the EmergencyBoard (1), where the PWM signals are calculated and normalized. The normalized data is then finally placed on the CAN interface (2). In case complex mathematical computations are required, the ARMBoard receives the normalized data, performs the desired calculations, and places the processed data on the CAN bus with a different id. The selector switch on the ActorBoard chooses the source of data, either receiving processed data from the ARMBoard or normalized data from the sensor boards (3). Independent of the source, the ActorBoard, based on these signals, finally produces the desired PWM signals (4), and passes these signals to the actuators (servo motor and DC motor) via the EmergencyBoard (5).

It is important to understand the services offered by the EmergencyBoard. Having no access to the CAN bus, this board only accepts the input from the radio receiver and the ActorBoard, and bypasses it to the sensor boards and the actuators (servo and DC motor), respectively. The EmergencyBoard only bypasses the PWM signals as provided by the radio remote receiver and the ActorBoard, if there is no emergency signal being sent from the emergency transceiver module, thereby bringing in the feature of emergency stop and the galvanic isolation (isolating the actuators from the rest of the ECUs).

IV. Targeting Different Abstraction Layers using Averest

This section presents how system descriptions are generated at different abstraction levels using *Averest* for ActorBoard ECU of the ConceptCar: This ECU has been generated at three different levels of abstraction.

[6]http://es.cs.uni-kl.de/research/applications/concept-car
[7]http://www.iese.fraunhofer.de

```
macro CAN_RECEIVE_OK = 4;
macro ERRORID_PWM_OUTOFBOUNDS = 1;
macro ERRORID_PWM_FREQUENCY = 2;
macro ERRORID_RECEIVER_NOSIGNAL = 3;

module ActorBoard (
      nat{exp(2,32)} ?STEERINGDATA_IN, //received from the CAN bus
            ?STEERINGDATA_IN_ARM7,
            ?THROTTLEDATA_IN,
            ?THROTTLEDATA_IN_ARM7,
            ?CAN_ERROR,
      bv{8} ?SWITCH_STATE_IN,
            ?PULSESTEERING_TIMEOVER,
            ?PULSETHROTTLE_TIMEOVER,
      nat{exp(2,16)} !PWM_STEERING_OUT,//out signals for driving the actors
            !PWM_THROTTLE_OUT,
      bv{8} !SWITCH_STATE_OUT,
      bool  SOURCE1_LED,
            SOURCE2_LED,
            RUNS_LED){
   //***place the switch status (either '0' or '1') on the CAN bus***
   loop{
      HANDLE_LEDS (SWITCH_STATE_IN,SOURCE1_LED,SOURCE2_LED);
   }
   ||
   //***keep track of CAN error***
   loop {
      if((CAN_ERROR == ERRORID_PWM_OUTOFBOUNDS) |
               (CAN_ERROR == ERRORID_PWM_FREQUENCY) |
               (CAN_ERROR == ERRORID_RECEIVER_NOSIGNAL))
      {
         DISABLE_LED(RUNS_LED);
      }
      else {
         ENABLE_LED(RUNS_LED);
         //if source 1 from switch is selected, use unprocessed value (ignore ARM7)
         if(SWITCH_STATE_IN{0}) {
               PROGRAM_PWM_STEERING(PULSESTEERING_TIMEOVER,STEERINGDATA_IN, PWM_STEERING_OUT);
               ||
               PROGRAM_PWM_THROTTLE(PULSETHROTTLE_TIMEOVER,THROTTLEDATA_IN, PWM_THROTTLE_OUT);
            }
            //if source 2 from switch is selected, use processed reading from ARM7
            else  {
               PROGRAM_PWM_STEERING(PULSESTEERING_TIMEOVER,STEERINGDATA_IN_ARM7, PWM_STEERING_OUT);
               ||
               PROGRAM_PWM_THROTTLE(PULSETHROTTLE_TIMEOVER,THROTTLEDATA_IN_ARM7, PWM_THROTTLE_OUT);
            }
      }
      runs: pause;
   }
}
```

Fig. 3. Quartz Description for the ActorBoard

A. Modeling the ConceptCar's Behavior

Since the ConceptCar features programmable ECUs, there are four different approaches to model and program each individual unit. These four different approaches are characterized by the mentioned different levels of abstraction. Practically, different ECUs of the ConceptCar have been tested and implemented with three levels of software design (except pure hardware development). For instance, the sensor boards for steering and throttling are implemented using bare-iron level. The SensorBoard_Inertial has been implemented with a real-time operating system, namely FreeRTOS. Initially, the ActorBoard was implemented using bare-iron programming, and was later extended with the combination of model-based and OS-based design methods, using Matlab and FreeRTOS, respectively.

Using Averest, we generate code for all the ECUs, at different levels of abstraction, by using a single description in Quartz. This allows the programmer/developer to program and access different ECUs of the ConceptCar, simply by using a single model-based language, namely Quartz, without having any proficiency in the corresponding abstraction level. In other words, it bypasses the requirement of having knowledge and understanding of the instruction set of the embedded processor in case of bare-iron level, device drivers or scheduling algorithms in case of RTOS programming, and models written with other model-based design tools. One step further, Averest also allows one to produce HDL code directly from the Quartz description, in the absence of any targeted embedded processor.

1) The Target ECU: The ActorBoard: Due to lack of space, it is not possible to present the Quartz description and the corresponding code produced for different abstraction levels and for each ECU. As mentioned, the ActorBoard was previously implemented with the bare-iron method, and is now extended with the model-based and the OS-based design methodologies. This ECU (with all three implementations) better explains the approach of realizing each Quartz implementation at different levels of abstraction.

The basic functionality of the ActorBoard is to drive the actuators (DC motor and servo) for throttling and steering the ConceptCar. This is done by generating the PWM signals for throttling and steering, based on the data received from the remote controller. In general, the ActorBoard receives the throttling and the steering input directly from the SensorBoards via the CAN bus. Alternatively, if some intensive processing is desired, the ARMBoard receives the signals first, processes them, and finally places the processed signals to the CAN bus.

The complete Quartz description for the ActorBoard is shown in Figure 3. This description mainly involves two separate loops (running in parallel), each confined to a specific task. The first loop using sub-module HANDLE_LEDS, updates the source LEDs (represented by interface variables SOURCE1_LED and SOURCE2_LED in the description). The second loop continuously monitors the CAN bus. Since the data loaded on the CAN bus can possibly come from two different sources (from the SensorBoard or the ARMBoard), a hardware switch is placed as an indicator to the ActorBoard for selecting the source for throttling and steering signals (represented by SWITCH_STATE_IN in the Quartz description). When the SWITCH_STATE_IN is at active-high state, the ActorBoard fetches the unprocessed data from the CAN bus, thus ignoring the processed data from the ARMBoard, indicated by the interface variables STEERINGDATA_IN and THROTTLEDATA_IN, passed to the sub-modules PROGRAM_PWM_STEERING and PROGRAM_PWM_THROTTLE, respectively. Conversely, when the SWITCH_STATE_IN is at active-low state, the ActorBoard receives the data from the CAN bus with the id representing the processed data from the ARMBoard, indicated by the interface variables STEERINGDATA_IN_ARM7 and THROTTLEDATA_IN_ARM7 in the description. Independent of the source, it then generates the corresponding PWM signals.

B. Code Generation at Different Abstraction Levels

Taking into consideration the Quartz description of the ActorBoard, we present in this section the generated code at different levels of abstraction. Since the complete generated programs are quite lengthy, we present the code snippet that performs the operations to receive the data from the CAN bus, and to generate the corresponding PWM signals with respect to the status of the switch.

1) The Bare-Iron Level: A code snippet from the original bare-iron level implementation is as follows:

```
if (source) {
    // source is high --> signal passthrough
    id_throttle = CANID_THROTTLE;
    outpin_disable(&source2_led);
}
else {
    // source is low --> data from ARMBoard
    id_throttle = CANID_THROTTLE_ARM7;
    outpin_disable(&source1_led);
}
// Receive the messages with the decided id
if (CAN_receive(id_throttle, &message) {
    // Generate PWM signals based on received message
    programPWM(&pwmThrottle, message);
}
```

This code snippet presents the scenario of deciding the source of a received message (throttle data in this case), depending on the status of the hardware switch. If the switch is placed at the active-high position, the unprocessed value is fetched from the CAN bus. In contrast, if the switch is placed at the active-low position, the ActorBoard receives the ARMBoard's processed data from the CAN bus. Once the data is received, independent of the source, the corresponding throttle signal (PWM signal) is generated.

2) OS-Based and Model-Based Design: The same behavior as implemented using Matlab as model-based design, and FreeRTOS as the RTOS, is shown as follows:

```
portTASK_FUNCTION(vQueryCAN, param) {
    while (1) {
       timeStamp = xTaskGetTickCount();
       if (CAN_rcv(srcSel?CANID_THR:CANID_THR_ARM, &msg)) {
          nor = (float)message / 1000.0f;
          pwmThr.pwmPul = getpwmPul(nor, MXREV, IDL, MXTHR);
          pwmThr.timeStamp = timeStamp;
       }
    }
}
```

The above code snippet stems from a C program obtained by a Matlab model of the ActorBoard, for deciding the source of received messages. The hardware switch is represented by srcSel in the code. Since it uses the FreeRTOS, the task of generating the PWM signal is scheduled separately, as a separate task (portTASK_FUNCTION() in the code).

In the same way, using the Averest Toolkit, the code can be generated at all levels of abstraction for each ECU of the ConceptCar.

V. Conclusion

We considered the ConceptCar as a case study for code generation at different abstraction levels. In particular, the Quartz description for the ActorBoard ECU of the ConceptCar is considered for this purpose. Using Averest, software at different levels of abstraction is generated from the single source description. As a result, the same behavior has been observed and recorded with different software levels in practice (compared with the original implementation at the specific abstraction level).

References

[1] P. Benjamin, M. Erraguntla, D. Delen, and R. Mayer, "Simulation modeling at multiple levels of abstraction," in *Winter Simulation Conference*, D. Medeiros, E. Watson, J. Carson, and M. Manivannan, Eds., vol. 1. Washington D.C, USA: IEEE Computer Society, 1998, pp. 391–398.

[2] E. Lee and S. Seshia, *Introduction to Embedded Systems – A Cyber-Physical Systems Approach*. http://leeseshia.org, 2011, ISBN 978-0-557-70857-4.

[3] O. Rafique, "Design, development, and integration of a wireless communication unit in ConceptCar," Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany, February 2013.

[4] O. Rafique, M. Gesell, and K. Schneider, "Generating hardware-specific code at different abstraction levels using Averest," in *Software and Compilers for Embedded Systems (SCOPES)*. ACM, 2013.

[5] K. Schneider, "The synchronous programming language Quartz," Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, Internal Report 375, December 2009.

[6] K. Schneider and T. Schüle, "A framework for verifying and implementing embedded systems," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, B. Straube and M. Freibothe, Eds. Dresden, Germany: Fraunhofer Institut für Integrierte Schaltungen, 2006, pp. 242–247.