Generating Hardware Specific Code at Different Abstraction Levels using Averest

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

ABSTRACT
In general, embedded systems can be designed at different levels of abstraction, e.g., as pure hardware circuit designs, as bare-iron level programs (without an operating system), as programs based on a real-time operating system, and as models of a model-driven development. This paper focuses on a synchronous model-driven development tool called Averest. Using Averest, we describe how we consider and combine system descriptions at the mentioned four levels of abstraction. We discuss a case study targeting a distributed embedded system where these different levels have been used.

Categories and Subject Descriptors
B.5.2 [Register-Transfer-Level Implementation]: Design Aids—Automatic synthesis, Hardware description languages, Optimization, Simulation; C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems; C.0 [General]: Hardware/software interfaces; D.2.10 [Software]: Design—Methodologies

General Terms
Design

Keywords
ACM proceedings, hardware specific code, abstraction levels, real-time embedded systems, synchronous languages, Averest

1. INTRODUCTION
Embedded systems, ranging from very simple systems up to very complex aircraft engine controllers, may nowadays have quite challenging (real-time) computational and design requirements. These increased demands have lead to the development of heterogeneous and distributed designs that sometimes require optimizations at low levels of abstraction. Moreover, with the advent of such complex systems, the development of embedded systems has called for new design methodologies.

Embedded systems can be modeled and implemented at different abstraction levels, in particular, as an application-specific hardware circuit, as bare-iron level software (runs without operating system directly on the hardware), as tasks of a real-time operating system, and also as realization-independent behaviors in a model-driven design flow. This paper considers such a model-based design flow for embedded systems using the Averest [4, 1] tool. Systems are thereby modeled independently of a later realization using a synchronous programming language (called Quartz). Using a case study, we demonstrate how to translate a single Quartz module into code at different levels of abstraction.

2. MODEL-BASED DESIGN BY AVEREST
This section describes how Averest can be used to target the different abstraction levels of embedded systems mentioned in the introduction. To this end, we first give a brief introduction to the Averest toolkit.

2.1 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 generated), 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 or 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 code or hardware code from the same Quartz module without any modification of the Quartz module (by applying different transformations for synthesis).
2.2 Code Generation at Different Abstraction Levels

We now present the different abstraction levels from the perspective of how the Averest framework approaches these levels.

2.2.1 Pure Hardware Development

The first step to generate a hardware description from a Quartz file consists of the construction of an equation system of an AIF file. To this end, compound data types like tuples are reduced to scalar data types, and if the hardware description is wanted at the bit-level, all data types are finally reduced to booleans only. Then, an equation system describing the behavior is obtained by collecting for each boolean variable all actions on that particular variable. Afterwards, these equations are used to generate Verilog code that can be used by state of the art tools to configure an FPGA or to generate a hardware circuit.

2.2.2 Bare-Iron Level Implementation

Programming languages like C can be used for modeling a desired system behavior, specifically for the application-specific hardware. Depending on the hardware used (for instance, a micro-controller), this method typically makes use of memory-mapped I/O to access hardware peripherals.

In addition to the implicit transformation steps, the user needs to define during compilation that the generated code targets the bare-iron level. This is done by an AIF transformation ToBareIron that includes the renaming of the interface variables to target (the bare-iron part) of a simple hardware abstraction layer (HAL). This part of the HAL is very simple and only maps the interface variables to the corresponding hardware interface.

2.2.3 OS-Based Design

OS-based design can be considered as an extension of the previous layer where the execution of the software makes use of a real-time operating system (RTOS). Using RTOS, different tasks of a complex system can be better managed, organized, and executed with a virtual parallelism.

The usage of a RTOS with Quartz is easily done by extending the HAL for communicating with such an operating system. Besides the implicit transformations to generate C code, the user must declare during compilation that the generated code targets the OS-based level. This is done by an AIF transformation ToOS that includes the renaming of the interface variables to target the OS-based part of the HAL.

2.2.4 Model-Driven Development

It is easy to change the hardware/software partitioning of an existing Quartz code. Replacing an existing component by a Quartz module only requires a new Quartz module with a compatible interface. The same techniques are usable to exchange hardware by software and vice versa. A module in Quartz must be written for (or excluded for) compilation to use the Quartz implementation (or a predefined component). All these different methods do not restrict the re-use of already existing components to assemble new systems. These components are rarely physical objects. Instead, code bases exist for hardware parts (e.g., processor cores), for software parts (e.g., software libraries) as well as combinations of them (e.g., CAN bus controllers). They are usually distributed as so-called intellectual property (IP) cores. In [3], an adequate infrastructure for their integration in Averest is explained. Hence, a model-based design approach is easily realizable by maintaining a HAL connecting these IP cores with the generated code. In that way, the HAL must contain the complete hardware-specific code.

2.3 The Code Generation Work Flow

The work flow to translate a Quartz description to different abstraction levels, is shown in Figure 1. The design process starts with the Quartz implementation. First, using Averest, the Quartz implementation can be directly translated at the pure hardware level (gate level or HDL representation), without striving further for any additional step. For the remaining three levels of abstraction, the Quartz description is processed with the common procedure as: The code generator of Averest, is fed with the Quartz code. The Averest toolkit generates automatically the C code translation of the original Quartz implementation. Depending on the Quartz implementation and the transformations applied in the Averest toolkit (explained in the previous section), this translated C code can represent any of the three levels of abstraction. As a requirement of the model-driven programming, a HAL acts as a bridge between the software design (translated C code) and the hardware (typically the instruction set of the embedded controller). The HAL, merely a software library, maps the translated C code into a form that is compatible with the targeted hardware.

![Figure 1: The Code Generation Work Flow](image)

3. 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 have become very practical and realizable. The best evidence is the intensive research in the field of driver assistance systems. The ConceptCar [2] (designed and developed by our group together with Fraunhofer IESE) is an experimental embedded system (remotely controlled) with the objective of testing and verifying modern future car features by deploying different classes of applications.
The ConceptCar (as shown in Figure 2), although not as big as a conventional car, has been built and engineered as close as possible to a modern car (concerning its embedded systems). The ConceptCar is a research platform remotely operated via a standard 2-channel (throttle and steering) 27MHz radio transmitter system. It is set up with a pair of sensors (wheel speed, gyro/accelerometer, distance etc.) for interacting with the environment and surroundings. It uses an air-cooled sensorless brushless electrical motor for throttle, and a servo motor for steering. The power train of the ConceptCar consists of two independent power sources, one for the heavy load electric system (motors/actuators), and the other for powering up all the ECUs. Depending on the ground conditions, it is capable of driving at a speed up to 50 km/h.

Although it is not incorporated with as many ECUs as a modern car can carry (up to hundred), the ConceptCar still features 7 different ECUs (as shown in Figure 3), where each ECU is responsible for a specific operation. These ECUs are organized in three processing units: The SensorBoard ECUs, as incorporated with different sensors, form the input processing unit, responsible for interacting with the environment and surroundings. The ARMBoard is used as a data processing unit and only comes into play when complex mathematical computations are required. The ActorBoard ECU forms the output processing unit, responsible for creating the PWM signals to drive the actuators (DC motor and servo). Likewise a modern car, all ECUs interact with each other via a centralized CAN bus architecture. Since the ConceptCar incorporates two separate power sources, a special ECU called EmergencyBoard is added, which serves two purposes: (1) Isolates the actuators from the other boards (Galvanic Isolation), thereby removing any possibility of damaging the other ECUs due to any power consumption issue related to the motor and the servo. (2) Uses an additional transceiver module for invoking the emergency stop. This additional transceiver module can reach up to 1.5 km. Pushing any of the three buttons on the emergency remote control sets the car in a stop mode. All the ECUs (except EmergencyBoard) communicate with each other via centralized CAN bus, with the maximum achievable data transfer rate of 1 Mbps.

Practically, different ECUs of the ConceptCar have been tested and implemented with three levels of software design (except pure hardware development). For instance, the SensorBoard_Steering and SensorBoard_Throttle are implemented using bare-iron level. The SensorBoard_Inertial has been implemented with a real-time operating system, namely FreeRTOS. The ActorBoard is modeled with the combination of model-based and OS based design methods, using Matlab and FreeRTOS, respectively. Using the presented Averest approach, we generated codes at different levels of abstractions for different ECUs, essentially from a single source description for each ECU. As an outcome, the same behavior has been observed with different codes, translated at different abstraction levels, as achieved with the original implementation at the specific abstraction level.

4. CONCLUSION

We presented an approach to target different levels of abstraction, namely pure hardware circuit level, bare-iron level programs, programs on top of a real-time operating system, and behavioral descriptions of a model-driven development, with a single source description using Averest. For this purpose, a distributed real-time embedded system, namely the ConceptCar is presented as a case study. Using Averest, the codes are generated out of the single source description, at different levels of abstraction. Collaboratively, the HAL held responsible for mapping the generated codes on the hardware interface. Using translated codes at different abstraction levels, we achieved the same functionality with different ECUs, as achieved with the original implementation at the specific abstraction level.

5. REFERENCES