A Case Study in Embedded System Design: an Engine Control Unit

Tullio Cuatto, Politecnico di Torino, Italy
Claudio Passerone, Politecnico di Torino – Cadence Europeans Labs, Italy
Luciano Lavagno, Politecnico di Torino – Cadence Europeans Labs, Italy
Attila Jurecska, Magneti Marelli, Venaria Reale, Italy
Antonino Damiano, Magneti Marelli, Venaria Reale, Italy
Claudio Sansoè, Politecnico di Torino, Italy
Alberto Sangiovanni-Vincentelli, Dept. of EECS, University of California at Berkeley, USA

Abstract

A number of techniques and software tools for embedded system design have been recently proposed. However, the current practice in the designer community is heavily based on manual techniques and on past experience rather than on a rigorous approach to design. To advance the state of the art it is important to address a number of relevant design problems and solve them to demonstrate the power of the new approaches.

We chose an industrial example in automotive electronics to validate our design methodology: an existing commercially available Engine Control Unit. We discuss in detail the specification, the implementation philosophy, and the architectural trade-off analysis. We analyze the results obtained with our approach and compare them with the existing design underlining the advantages offered by a systematic approach to embedded system design in terms of performance and design time.

1 Introduction

Hardware/software co-design and embedded system design techniques, such as those presented in [6], advocate a formal design procedure for embedded systems, based on unbiased specification, simulation-based validation, various forms of automated partitioning, and module and interface synthesis.

However, the design methodology followed in practice is far from being at the sophistication level implied by the approaches listed above. The common resistance offered by designers to innovation is made even stronger in the case of embedded system design because of the safety concerns and of the strict constraints on implementation costs. To demonstrate the applicability of the design methodology and tools that our group has proposed ([1]) we tackled an industrially relevant design: an Engine Control Unit for a commercial vehicle. The case study is relevant because it represents an actual product and because of its complexity. In particular, we were able to compare the results obtained with our design methodology with the present implementation showing its advantages both in terms of performance and design time.

The paper is organized as follows. In Section 2 we describe the specifications of our case study. In Section 3 we show how architectural decisions, such as partitioning and processor choice, are made for that example. In Section 4 we discuss some lessons learned in the course of this project, about specification styles and trade-offs. In Section 5 we draw some conclusions.

2 Specification of the ECU

In this paper we present an application of the *POLIS* [1] environment: the formal specification and hardware/software partitioning analysis of a functional subset of an Engine Control Unit (ECU), a typical automotive embedded system. We used as a reference an already existing device from Magneti Marelli, a worldwide supplier of automotive electronic components. Both a functional Structured Analysis specification [4], and a target architecture were available for this device, that was originally designed as a prototype to experiment with On-Board Diagnosis control strategies.

An electronic *Engine Control Unit* (ECU) consists of a set of sensors which periodically measure the engine status, an electronic unit which processes data coming from the sensors and drives the actuators, and the actuators themselves which execute the commands received from the control unit. A control strategy is implemented in the electronic unit to optimize the fuel injection and ignition; in particular it should minimize fuel consumption, minimize emissions of polluting substances and maximize torque and power, when possible. These requirements are usually competing, so the algorithm must find the best compromise for each situation.

The two main tasks of an ECU are the control of injection and ignition. The control specifications for these two tasks are as follows:

Injection : in order to burn completely and correctly the fuel, the ratio between the air and the fuel which go into each piston should be kept constant and close to the value 14.7 (for a gasoline engine). This is achieved by controlling the opening time of each injector.

Ignition : in order to give the fuel enough time to burn completely, the spark should be fired in advance with respect to the instant when the piston is at its highest point. This parameter also affects consumption and emissions, and it is basically computed from the engine RPM.

Both injection and ignition can be adapted dynamically with a very high precision, by processing the inputs signals coming from the sensors.

1-58113-049-x-98/0006/$3.50

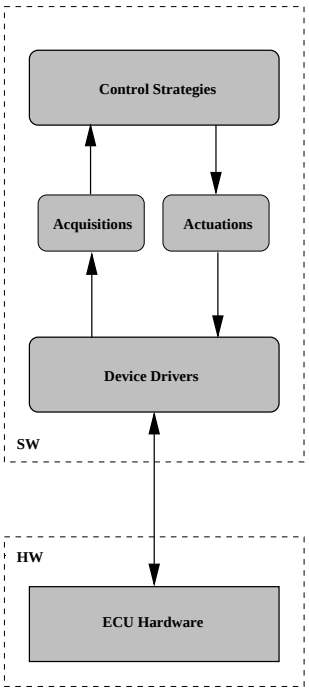

Figure 1: Functional architecture of an ECU

The functional architecture of the ECU is represented in Figure 1. The control layer implements the overall control strategy, aimed at satisfying the normative requirements in an optimal way. The signal acquisition and actuation layer transforms the raw data coming from the sensors (both binary-valued and derived from an A/D conversion) into filtered variables, scaled into the appropriate engineering units (like RPM) used by the control layer, while handling possible sensor errors and failures. The same layer also converts commands for the actuators, measured in engineering units (like seconds) into the appropriate signals (e.g. pulse streams) required by the actuation coils, lights and so on. The device driver layer further separates the signal acquisition and actuation layer, to provide some level of processor independence. The RTOS layer coordinates the overall functionality, by managing time-triggered (e.g., fixed-rate sensor sampling) and event-triggered (e.g., injection, triggered by the piston position) tasks.

The architecture of the embedded controller as implemented by Magneti Marelli contains a Motorola 68332 microprocessor (16 MHz), an ASIC implementing the PWM functions, 33 KBytes of used RAM and 194 KBytes of used ROM. The software written for the microcontroller consists of 35,000 lines written in C. The design effort was 8 man/year.

3 Design of the ECU

3.1 High level functional specification

Starting from the functional description of the ECU mentioned in the previous section, we chose a subset of relevant functionalities to implement using the *POLIS* environment. The ECU we implemented can manage the basic control operation of a 4 cylinder Multi-point Injection and Ignition System. We eliminated secondary tasks such as the interactions with the Catalytic Gas Sensor.

The control algorithm itself was taken from the original specifications untouched. This corresponds to the present organization of Magneti Marelli where two separate divisions handle the control algorithms and the actual implementation in terms of software and hardware modules. This separation favors some degree of architecture independence and the sharing of control algorithms between systems designed for different cars.

The specification has been partitioned into a hierarchical network of functional blocks, each described using the Esterel language. At the top level of the specification there are four main blocks:

1. acquisition of analog signals,
2. acquisition of frequency signals,
3. engine control strategies for the injection and ignition subsystems
4. control of the actuators.

The acquisition blocks can handle several types of sensors, including detection and recovery strategies in case of faulty sensors. The most important ones are the battery voltmeter, the intake manifold air pressure sensor, the throttle position sensor, the air and water temperature sensors, the position sensor of the phonic wheel and the camshaft position sensor.

After measuring and filtering the above external signals, the acquisition block calculates the values of system variables, such as RPM, air flow, derivatives of input signals, and so on. These variables are used by the control strategy layer, whose algorithm was taken from the original specifications almost untouched.

3.2 Functional simulation

Once the functionality of our system has been captured as a set of interacting Extended Finite State Machines, we can use *POLIS* to verify the behavior of the overall algorithm. In this step, timing is not considered since the implementation architecture has not been chosen, but instead a maximally parallel solution is analyzed.

The top level of the simulation model contains the testbench, using special functional blocks that model (in a simplified way, for reasons of performance) the engine and the sensors, thus providing inputs and monitoring outputs of the ECU.

The output of the simulation, as well as the state of several internal variables for debugging purposes, was monitored and compared with the expected data in order to determine the functional correctness of the specification.

3.3 Architecture Selection and Mapping

One of the most important component of our methodology is the architecture selection step. Part of the architecture selection step is the decision on how to partition the functional behavior into hardware and software. Another part is the organization of several software modules into tasks, to be executed together by the RTOS, based on expected input event frequency and priority. One last aspect is the selection of the scheduling algorithm (cyclic or priority-based, pre-emptive or non-pre-emptive, ...).

The performance of the simulation, in terms of simulated clock cycles per second on an unloaded ULTRASparc1, is described in Table 1. The column labeled "no graphics" was obtained by eliminating the graphical displays. The

engine speed (RPM)	cycles/second graphics	no graphics
1000	366K	433K
6666	116K	216K

Table 1: Simulation performance

function	code size manual	synthesized
acquisition	18K	13K
actuation	12K	12K

Table 2: Code size comparison

relatively small difference between the cases with an engine speed of 1000 RPM and 6666 RPM is due to the fact that only some acquisition tasks are triggered by the engine phase, while some are triggered by periodic timer interrupts.

While our system can perform an analysis of different architectural choices involving a number of microcontrollers, in this case, we selected the Motorola 68332 microcontroller, running at 16 MHz, with an on-chip Time Processing Unit (TPU [7]), to be able to compare our results with the already available ECU. Part of the TPU function was specified and simulated at the behavioral level, as described in [5]. The more complex functions implemented using the microprogramming capabilities of the TPU, like recognizing the engine phase from the phonic wheel sensor sample, were, on the other hand, modeled in ESTEREL.

Partitioning was performed by starting from the functional simulation model, in which every block has a delay of 1 clock cycle, and thus corresponds to a maximally parallel hardware implementation. We then moved selected groups of modules to software, based on the timing constraints and feedback from the simulation.

POLIS models functional components as Finite State Machines (extended with integer arithmetic capabilities) communicating via one-place buffers. Any time one of those buffers is overwritten, this is often a sign that a deadline has been violated, because the receiving FSM was too slow. Missed deadlines (in the form of lost events) are logged to a file, and can be used as a guidance for system performance analysis.

Since the system is composed of three main functional blocks, which in turn consist of several modules, we decided to start the partitioning process by changing the implementation of each unit as a whole; only after several experiments we had to refine our strategy by looking more closely at the lower levels of the hierarchy, to identify the critical path.

1. The first experiment was a complete hardware implementation. This is equivalent to the functional verification through simulation described in paragraph 3.1, and yielded the expected results. The system at 16 MHz flawlessly performs its task, but the implementation is very expensive due to the presence of many multipliers and dividers.

2. As a second step, the Analog Acquisition block was entirely implemented as software. The simulation showed that even with a slower CPU (i.e. 4 MHz), the results were correct.

3. The driver blocks were therefore also implemented as software, both for injection and ignition; however, the Frequency Acquisitions unit was left in hardware, and the low level counting task were assigned to the TPU. Even with the engine running at the highest speed (6600 RPM) no event was lost, and the algorithm always managed to correctly control the engine.

4. Except for the 68332 TPU, a full software implementation was then simulated. This time, even with the engine at a moderate speed, a lot of events coming from the crankshaft position sensor were lost. We identified two critical modules, those that filter the signals coming from the engine, since they are scheduled at the highest frequency in the entire system.

5. The next step was to implement one of the critical modules as hardware, and the other as software, but this again did not produce the desired result. By closely examining a plot of the task schedule (in the same form as shown in Figure 2), and comparing it with the missed deadline log, we discovered that the problem was the high latency introduced by the Round Robin scheduler we were using.

6. The solution was therefore to drive the acquisition of the signal coming from the engine through an interrupt, thus stopping the ongoing computation to start the Interrupt Service Routine. In this way, both critical modules can be implemented as software routines, without losing any functionality.

A plot of the scheduling charts with the engine running at full speed for the final partition is shown in Figure 2: the bars which reach the value 1 are the two interrupt routines, and the rest are all the other tasks.

The CPU load, due almost totally to the acquisition and actuation layer, is about 40% in the final partition. This number is close to the estimated load due to that layer in the manual design, thus indicating that the (estimated) performance of the synthesized code is very close to that of hand-written code.

The total code size in bytes (for the 68332 microcontroller) in the final partition is compared with the manual design in Table 2.

The entire partitioning process was performed within two days. The tool lets the designer change the implementation of each single block by just updating a parameter and then running a new simulation. No further compilation is required. Also the scheduling charts and a log of the missed deadlines are automatically generated, and were very helpful to drive the partitioning process.

4 Lessons learned and design guidelines

During this project, we had to choose a method to translate the existing formal functional specification, done with Structured Analysis ([4, 3]) into an executable specification using CFSMs. This required to solve a number of trade-offs about the *granularity* of the CFSMs.

One main aspect of the compilation mechanism used by ESTEREL (and hence by the *POLIS* front-end) is that it *fully abstracts* the user-given specification. This means that the control structure in the ESTEREL specification (e.g., `loop` and `await` statements) is translated into a Finite State Machine. The transition function of each CFSM is then represented as a Binary Decision Diagram [2], and the optimal sequence of `if`, `goto` and assignment statements is computed [1]. This is a very expensive procedure, that yields automatically results that are comparable with hand-optimized code, but is sensitive to module size.

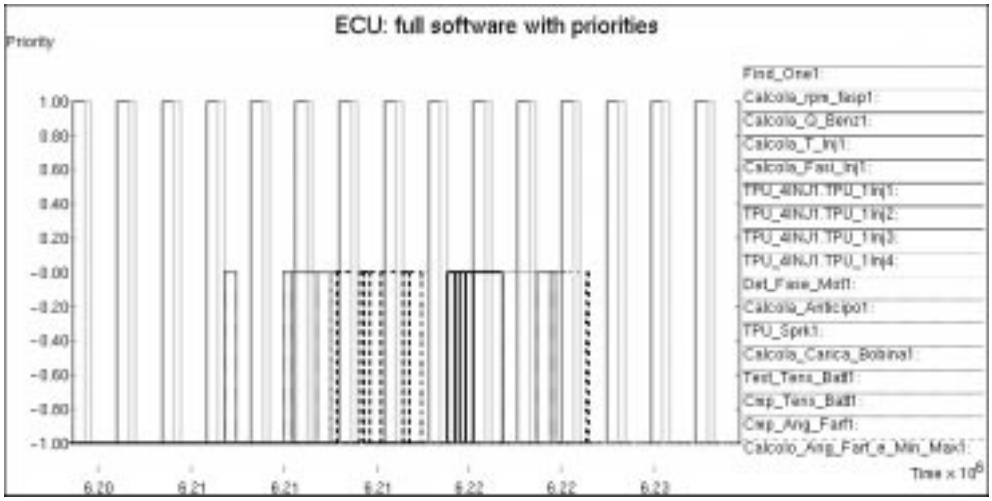

Figure 2: Task scheduling chart for the final implementation

Small modules are easier to understand, and guaranteed not to blow up at compilation time. However, the resulting code, with one procedure per module, can impose an excessive overhead in terms of code size and execution time on the target processor, due to the procedure call and return sequences of short procedures.

On the other hand, large modules offer better opportunities for optimization as described above, but may require too much memory or CPU time during compilation.

POLIS, however, provides the designer with several techniques for exploring these trade-offs in an assisted manner. In particular, it provides methods for

- chaining the execution of several modules, without returning control to the Real Time Operating System,
- merging several CFSMs, by computing their synchronous product, into a single CFSM.

The first technique is relatively cheap, does not cause any potential blowup in code size, and is especially suitable for data-intensive modules. The second technique is especially useful for control-dominated modules with a high degree of interaction, because it abstracts away all the internal communication between the collapsed CFSMs, thus yielding a dramatically simpler implementation in several cases.

For example, we applied the first technique to the submodule Test_Tens_Batt, which consists of two cascaded CFSMs. The first one checks the value of the battery voltage and implements a recovery strategy, while the second one computes the value of an internal variable. In this way we speeded up the code by 11% while keeping the same code size. We estimated that this kind of optimization can be applied to about 20% of the modules of the design.

In another case, we resorted to the opposite transformation, that is splitting a single CFSM, that was updating several variables concurrently, into four cascaded CFSMs, reducing the total code size from 11 Kbytes to 1.5 Kbytes.

5 Conclusions

New methodologies are required in the design of digital electronic systems, due to the increased complexity and reduced time-to-market. Moreover, a product should be flexible to adapt to changes during its lifetime, which is best obtained by using software, and must also meet tight timing constraints, which is most suitable for hardware components. Partitioning between hardware and software is therefore a critical step in the design flow, and CAD tools should help the designer in making the right choices.

In this paper, we have shown how to perform partitioning by using fast co-simulation and software estimation. We have also shown how to optimize the system with respect to code size and running time, by selectively collapsing or dividing modules, and by moving the threshold between control and data-flow in conditional statements. The results obtained are very promising, and were achieved on a real design and in a relative short period of time.

References

[1] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-design of Embedded Systems – The POLIS experience*. Kluwer Academic Publishers, 1997.

[2] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[3] A. Damiano and P. Mortara. Problematiche software nei sistemi elettronici per applicazioni automotive. *Alta Frequenza – Rivista di Elettronica*, 7(3):10–16, May-June 1993.

[4] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, USA, 1988.

[5] H. Hsieh, L. Lavagno, C. Passerone, C. Sansoè, and A. Sangiovanni-Vincentelli. Modeling micro-controller peripherals for high-level co-simulation and synthesis. In *Proceedings of the International Workshop on Hardware-Software Codesign*, March 1997.

[6] G. De Micheli and M. G. Sami, editors. *Nato Advanced Study Insitute on Hardware/Software Codesign*. Kluwer Academic, 1996.

[7] Motorola Inc. *M68300 Family. TPU Time Processor Unit. Reference Manual*, 1990.