Chapter 26
A Model-Based Methodology to Generate Code for Timer Units

Marco Marazza, Francesco Menichelli, Mauro Olivieri, Orlando Ferrante and Alberto Ferrari

Abstract In this paper we present a model-based methodology and a tool-chain supporting pseudo-automated code generation for different Timer Units, which represent a new approach in this field. Programmable Timer Units are timing co-processors used to elaborate complex high-resolution timing functions subject to hard real-time constraints. Verification at the different design stages, as required per safety standards' certification, is becoming a major concern for Timer Units code development life-cycle. Enabling correct-by-construction code generation, our methodology supports code development, integration and testing across all design phases. We show how high-level functional models derived from functional requirements can be mapped onto the target architecture and how architecture-specific code can be generated. Our methodology is then applied to an automotive reference example.

Keywords Embedded Systems · Code generation · Timer unit

M. Marazza (✉) · F. Menichelli · M. Olivieri
Department of Information Engineering, Electronics and Telecommunications (DIET), Sapienza University of Rome, Via Eudossiana, 18, 00184 Roma, Italy
e-mail: marazza@diet.uniroma1.it; marco.marazza@utsce.utc.com

F. Menichelli
e-mail: menichelli@diet.uniroma1.it

M. Olivieri
e-mail: olivieri@diet.uniroma1.it

M. Marazza · O. Ferrante · A. Ferrari
Advanced Laboratory on Embedded Systems (ALES), Via Barberini, 50, 00187 Roma, Italy

O. Ferrante
e-mail: orlando.ferrante@utsce.utc.com

A. Ferrari
e-mail: alberto.ferrari@utsce.utc.com

A. De Gloria (ed.), *Applications in Electronics Pervading Industry, Environment and Society*, Lecture Notes in Electrical Engineering 351,
DOI 10.1007/978-3-319-20227-3_26

26.1 Introduction

An increasing number of today's industrial applications demand accurate control of timing synchronization. Typical examples come from the automotive domain: cylinder spark timing, fuel injection timing and fuel mixture control must be precisely controlled to achieve the highest gain in terms of fuel economy, unwanted emissions and engine performance, while guaranteeing low energy consumption [9]. In these cases even the use of low-latency software interrupts might not achieve the required high time resolution. Moreover, the great number and the concurrent nature of these timing functions exaggeratedly increases the workload of the Electronic Control Unit's (ECU) processor. To help delivering hard real-time functions, programmable Timer Units can be integrated into the ECU architecture. These co-processors are provided with custom hardware and software to reduce the amount of I/O processing on single- or multi-core CPUs [8]. Examples of programmable Timer Units can be found in [4, 6]. Each programmable Timer Unit is provided with its own programming language: while the ETPU [4] is provided with a customized high-level assembly programming model, the GTM [6] is provided with a C-like compiler prototyped in LLVM [7]. Despite high-level languages have been developed, still Timer Units' programming models differ significantly and require a great amount of time to develop and debug the code. In this paper we propose a methodology to add a model-based programming layer. The benefits of such an approach are manifold: (1) the programmer would be provided with a single programming environment for many Timer Units; (2) a model-based environment typically allows simulating the models to check whether they fulfill functional requirements; (3) many tools in the market supporting model-based design provide automatic code generation, thus saving code development time considerably and enable automatic test generation [3]; (4) many formal analysis techniques (e.g. [3, 10]) can be successfully addressed on functional models, thus reducing the risk of generating unspecified or bugged source code. To the best of the author's knowledge this approach has not been addressed yet for Timer Units and represents an opportunity to improve code development and verification for Timer Units.

26.2 Reference Example

As a reference example we consider a function controlling the cylinder ignition timing of an internal combustion engine [5]. This function controls the generation of spark pulses that feed a spark plug actuator. Spark pulses must be synchronized to specific angular positions of the rotating shaft of the engine. Our reference ignition function consists of generating a main spark pulse, followed by a sequence of multi-spark pulses at each complete engine cycle (720°). Generally, the properties of these I/O functions (i.e. the angular value at which the main pulse must start for a specific cylinder, etc.) are parametrized and can be changed at run-time by the appli-

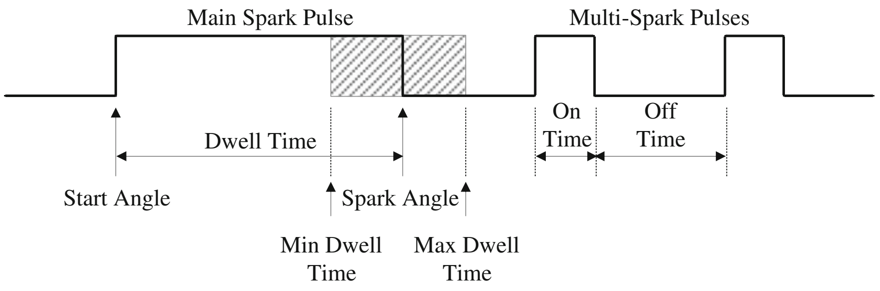

Fig. 26.1 Specification of the ignition function

cation running on the ECU. Figure 26.1 represents the specification of the ignition function. The *Dwell Time* is a function parameter set by the application running on the ECU and indicating the required active time of the main spark pulse. *Start Angle* is the required shaft's angular position at which the main spark pulse must switch to active and *End Angle* is the required angle for the opposite transition of the main spark pulse. Since the ignition function must guarantee that the spark pulse ends at the correct engine angle irrespective to engine acceleration or deceleration, it also has two additional parameters: *Minimum Dwell* (time) and *Maximum Dwell* (time). To ensure that the ignition coil has been charged sufficiently to generate a reliable spark after the pulse ends the spark pulse must remain active for at least a *Minimum Dwell* time length. Conversely, the spark pulse must be shorter than *Maximum Dwell* to ensure that the ignition coil being charged is not damaged by too much current and heat. After the main spark pulse has been generated, a set of equidistant short spark pulses can follow. This sequence is characterized by only three parameters: *Number of Multi-spark Pulses*, *Off Time* and *On Time*, indicating the number, and inactive time and active time lengths respectively.

26.3 Methodology

We propose a model-based methodology to automatically generate code for Timer Units. The main steps are summarized in Fig. 26.2; in this Section we briefly review each of the activities.

26.3.1 Functional Model

In our model-based methodology we start from the functional model *M* which is a formal representation of a function of the system, implementing the specified functional requirements. The benefit of implementing requirements by means of a formal

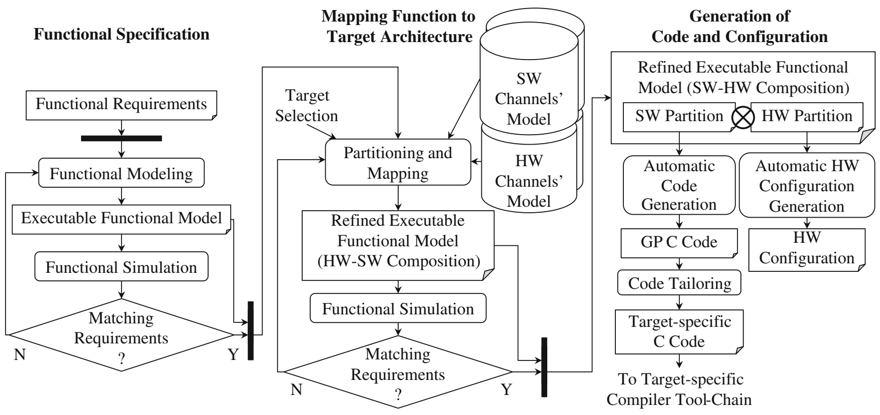

Fig. 26.2 Illustration of the proposed methodology

model rely on the possibility of executing and refining it. The refinement level of the model depends both on the designer's expertise and on the purpose of the model in the work-flow. As shown in the left-hand side of Fig. 26.2, the functional modeling phase is often an iterative process, converging to the needs of the designer. The formal nature of the functional model also opens the way to a set of formal verification activities that can be leveraged (1) to verify the correctness of the model against its (possibly formalized) requirements and (2) to generate test scenarios [3] that can be applied on the design at subsequent phases of the development life-cycle (e.g. for back-to-back testing).

26.3.2 *Mapping Functions on the Target Architecture*

The partitioning and mapping activity (central portion of Fig. 26.2) aims at the *decomposition* of the function into a set of functional components and *allocation* of each functional component to the proper architectural component(s) in the target Timer Unit. The inputs of this activity are: the executable functional model provided by the preceding modeling activity, the selection of the target Timer Unit, a library of hardware channels' models per each Timer Unit, and a library of channels' control software models per each Timer Unit. Timer Units' configurable hardware channels [4, 6], can be thought as a set of hardware-implemented services provided to the software. Each library model is still a functional (possibly hierarchical) model, but enriched with the hardware and software peculiarities of the specific Timer Unit. Model libraries can be designed by the end user or could be provided by third parties, e.g. by the Timer Unit vendor. To match the functional behavior, architectural components are picked up from the hardware and software libraries pertaining to the selected Timer Unit. Different Timer Units provide differing sets of services to their control software. Hence, mapping the same functional model on different architectures can be reduced to a graph covering problem. The result of this activity is a model $C = H \otimes S$ resulting from the composition ($\otimes$) of a hardware library model H and a software library model S and equivalent ($\equiv$) to the input functional model M, ($C \equiv M$).

26.3.3 Automatic Generation of Target Code and Configuration

Referring to the right-hand part of Fig. 26.2, the enriched model $C = H \otimes S$ is used to generate both the configuration and the source code controlling the behavior of each I/O hardware channel. The I/O hardware channel configuration is derived straightforwardly from the hardware partition model H. In facts, H models the function performed by the hardware channel, which only depends on its configuration. On the other side, the software partition model S is used to automatically generate correct-by-construction source code. The generated target source code, along with header files, have to be conform to the various Timer Units programming languages. This code has to be compiled in a later stage of the work-flow in order to be executed in the target Timer Unit. This approach subsumes that a C or high level assembly compiler is available to the developer, so that the automatically generated source code can be effectively translated into the executable machine-code.

26.4 Application to the Reference Example

We applied our methodology by using the Matlab/Simulink/Stateflow tool, along with the Embedded Coder Simulink Toolbox to automatically generate standard C code from our models. Such code has then been modified by hand and compiled with the specific Timer Unit's compiler [1, 7]. This Section gives a short description about how we accomplished the different phases of our methodology. The ignition function has been modeled as a Simulink/Stateflow Extended Finite State Machine (EFSM) by starting from the function specification. The formal model depicted in Fig. 26.3 is a function $F[\mathbf{u}, \mathbf{x}, \mathbf{f}, k]$ that at each discrete time k maps the inputs vector $\mathbf{u}[k]$ and the current state vector $\mathbf{x}[k]$ to a vector of outputs $\mathbf{y}[k]$ and next-state values $\mathbf{x}[k+1]$. All the input parameters are generated by a subsystem external to the EFSM in Fig. 26.3 and can change at every time, as required by the application. Figures 26.4 and 26.5 represent the execution of the ignition function in two corner cases: in Fig. 26.4 *End Angle* arrives before *Max Dwell*, whereas in the example in Fig. 26.5 *Max Dwell* occurs before *End Angle*. The waveform at the bottom of the two figures indicates the time between *Min Dwell* and *Max Dwell*, where the *End Angle* is expected to arrive. The function in Fig. 26.3 has been refined for the two

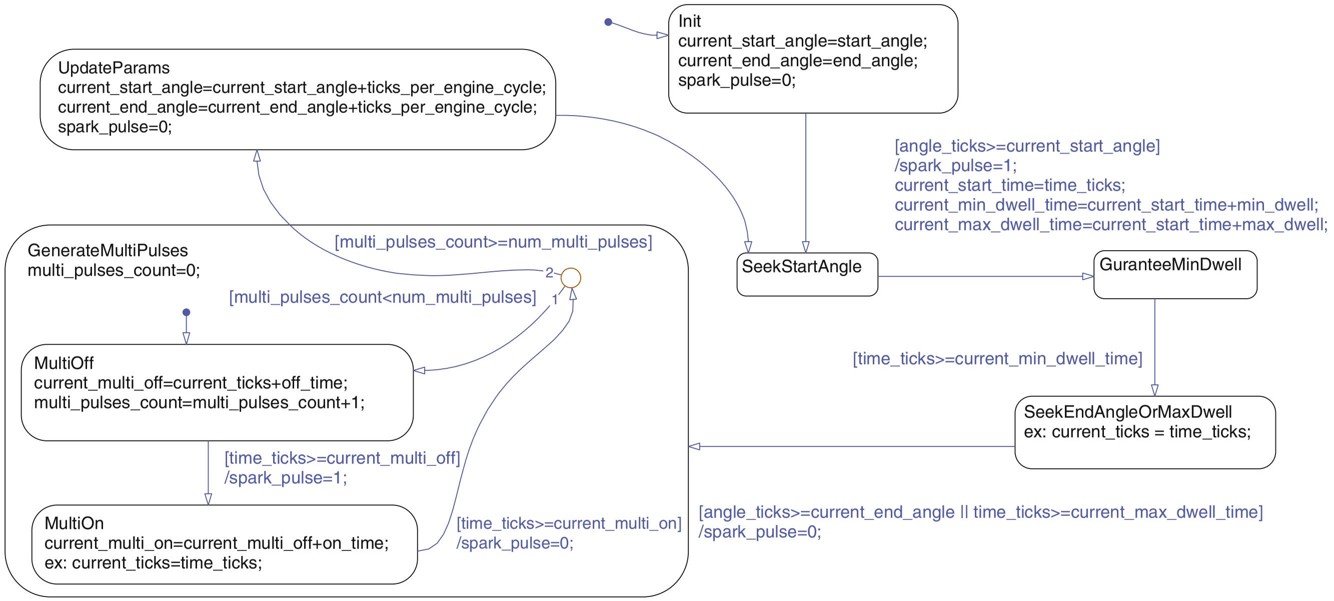
Init
current_start_angle=start_angle;
current_end_angle=end_angle;
spark_pulse=0;
UpdateParams
current_start_angle=current_start_angle+ticks_per_engine_cycle;
current_end_angle=current_end_angle+ticks_per_engine_cycle;
spark_pulse=0;
[angle_ticks>=current_start_angle]
/spark_pulse=1;
current_start_time=time_ticks;
current_min_dwell_time=current_start_time+min_dwell;
current_max_dwell_time=current_start_time+max_dwell;
SeekStartAngle
GuranteeMinDwell
[time_ticks>=current_min_dwell_time]
SeekEndAngleOrMaxDwell
ex: current_ticks = time_ticks;
[angle_ticks>=current_end_angle || time_ticks>=current_max_dwell_time]
/spark_pulse=0;
GenerateMultiPulses
multi_pulses_count=0;
[multi_pulses_count>=num_multi_pulses]
2
[multi_pulses_count<num_multi_pulses]
1
MultiOff
current_multi_off=current_ticks+off_time;
multi_pulses_count=multi_pulses_count+1;
[time_ticks>=current_multi_off]
/spark_pulse=1;
MultiOn
current_multi_on=current_multi_off+on_time;
ex: current_ticks=time_ticks;
[time_ticks>=current_multi_on]
/spark_pulse=0;

Fig. 26.3 Ignition function EFSM modeled in simulink/stateflow

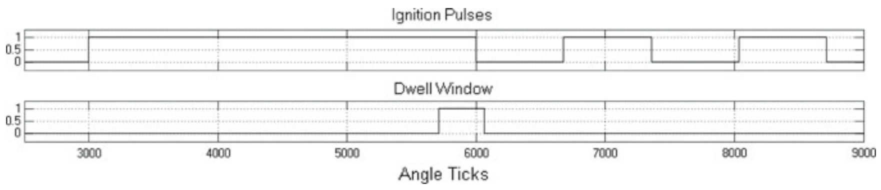

Fig. 26.4 Main spark pulse terminating at *MaxAngle*

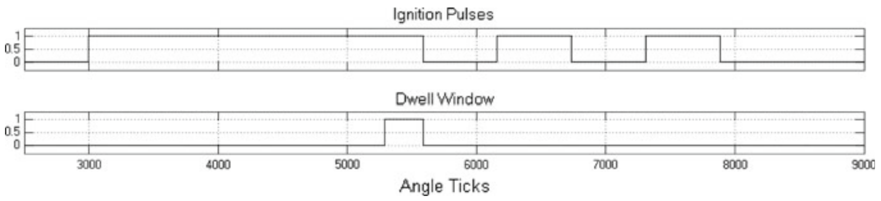

Fig. 26.5 Main spark pulse terminating at *MaxDwell*

architectures in [4, 6]. The hardware channels and the related control software have been modeled through the EFSM formalism. The composition of both the hardware and the software machines represents the ignition function as implemented on the two distinct architectures. The simulations of the refinements for the two architectures give the same results as shown in Figs. 26.4 and 26.5. As defined in Sect. 26.3, we exploited the software partition of the refined EFSM to generate the code specific to the target Timer Unit. In this preliminary work we generated code for a standard x86 platform and then manually tailored the resulting C code to match the programming model of the specific Timer Unit [1, 7]. This step helped us filling a set of tables of correspondences between I/O channel modes of the two Timer Units. The correspondences can be used to map particular "patterns" in the functional model to the corresponding library models of the channel mode or software code specific to the target architecture.

26.5 Conclusion

In this paper we presented a model-based methodology along with the supporting tool-chain for pseudo-automated code generation for different Timer Units, which represents a new approach in this field. The benefits are manifold: code developers can spend their effort on the modeling of the desired function, independently of the target Timer Unit; source code and hardware configuration can be generated automatically from the model and model-based automatic test generation and verification techniques can be exploited to test the design across its development phases. Future work will be devoted to automation of those phases that still require manual intervention.

References

1. ASH WARE: Compiler Reference Manual. v.2.01, 12 2011
2. Ferrante, O., Ferrari, A., Marazza, M.: Automatic Generation of Failure Scenarios for SoC. ERTS, Toulouse, France, 5–7 Feb 2014
3. Ferrante, O., Ferrari, A., Marazza, M.: Model based generation of high coverage test suites for embedded systems. In: Proceedings of the IEEE European Test Symposium, Paderborn, Germany, 26–30 May 2014
4. Freescale: Enhanced Time Processing Unit (eTPU) Reference Manual. 05 2004
5. Freescale: Using the eTPU Spark Function. Application Note. 07 2009. http://www.freescale.com/files/32bit/doc/app_note/AN3771.pdf
6. GTM-IP Specification revision. 06 2013. http://www.bosch-semiconductors.de/media/en/pdf_1/ipmodules_1/timer/GTM-IP_Specification_v1551.pdf
7. Marazza., M., Cremona, F., Ceraolo Spurio, D., Demuth, C., Nastasi, C., Ferrari, A.: Towards a Programming and Analysis Framework for Timer Units. In: JRWRTC, Sophia Antipolis, France, 16–18 October 2013
8. Menichelli, F., Olivieri, M., Benini, L., Donno, M., Bisdounis, L.: A Simulation-Based Power-Aware Architecture Exploration of a Multiprocessor System-on-Chip Design. DATE, pp. 312–317 (2004)
9. Menichelli, F., Olivieri, M.: Static minimization of total energy consumption in memory subsystem for scratchpad-based systems-on-chips. IEEE Trans. VLSI Syst. **17**(2), 161–171 (2009)
10. Rodrigues, C.: A Case Study for Formal Verification of a Timing Coprocessor. IEEE, LATW (2009)