System Level Design of Embedded Controllers: Knock Detection, a Case Study in the Automotive Domain

Leonardo Mangeruca*, Alberto Ferrari*, Alberto Sangiovanni-Vincentelli*,†,
Andrea Pierantoni‡, Michele Pennese‡

* PARADES, Via San Pantaleo, 66, 00186 Roma, Italy
‡ Magneti Marelli Powertrain S.p.A,Via Timavo, Bologna, Italy
† EECS Dept., University of California at Berkeley, CA 94720

Abstract

We present a case study in the design of automotive engine controllers: the development of a knock detection algorithm and its implementation in an optimized platform. The design problem is complicated by the need of using heterogeneous models of computation and different design environments. The use of different design environments, one for functional design and one for architectural design space exploration, requires to transform a model of computation into another. We describe how we solved this problem and we present the final design with the trade-offs explored.

1 Introduction

Knock detection is an important, compute intensive task in modern automotive engine control. The implementation of controllers that include knock detection often requires a DSP as a co-processor in a standard micro-controller architecture thus adding cost and design complexity. Since knock detection is becoming an integral part of engine controllers, it makes sense to explore alternative implementations. To do so, we need to evaluate a number of architectures that are based on different selections of components such as micro-processors, memories and interconnection schemes.

The evaluation has to be carried out by mapping computation to processing elements while making sure that timing and other physical constraints are satisfied. This design problem is typical of embedded systems in other industrial segments. Methodologies have been proposed for embedded system design including architecture design space exploration. However, the flow from algorithm analysis to implementation is not supported today by a unified tool environment thus forcing advanced designers to work with a patchwork of tools and formats.

The most serious problem for designers is actually the mismatch of the models of computation used by different tools. There is no guarantee that what has been modeled and simulated at a certain level of abstraction is consistent with other levels of abstractions. Consistency can only be verified by careful mapping of a model of computation into another. This is still an open problem in general in absence of an encompassing theory of models. We discuss in details this and other aspects of system level design while presenting our solution to the design of the knock detection function in engine controllers.

The paper is structured as follows: in Section 2, the knock detection problem is presented. In Section 3, the design methodology is presented. In Section 4 and Section 5, the general aspects of transforming the model of computation used in Simulink for hybrid system simulation into the model of computation supported by the VCC design environment, are described. Experimental results for the design exploration phase are presented in Section 6 and conclusions are presented in Section 7.

2 The Design Problem

In an internal combustion engine, an air-fuel mix is first injected into the combustion chamber of a cylinder. After the mix is compressed by the piston, a spark is generated to ignite the mix. At a first glance, it seems that maximum efficiency is achieved when the spark is given exactly at the end of the piston run (Top Dead Center, TDC). However, it is actually best from fuel efficiency point of view to give the spark *before* the TDC. Spark advance is measured in negative angles corresponding to the position of the piston with respect to the TDC. When the spark is given, the combustion wave propagates from the top of the cylinder towards the bottom generating the force needed to push back the piston and generate torque for the motion of the car. Knocking refers to an unwanted secondary combustion process that is not controlled by the spark but by pressure and temperature at the periphery of the combustion chamber. Knocking exercises forces on the piston and the combustion chamber that may cause engine failure and, as such, should be carefully avoided. The probability of knocking is proportional to spark advance. In standard engines, knocking is then minimized open-loop by setting an *a priori* limit on spark advance. In turbo-charged engines, an open-loop strategy is too risky given the high temperatures and pressions that develop in the combustion chamber. Hence, measurements are taken that detect knocking as it ensues, thus allowing a closed-loop control of the spark advance to eliminate knocking. Because of the relentless pursue of reductions in fuel consumption and pollution, knock detection is becoming a requirement for any type of engine since greater efficiency can be achieved by spark advances that are on the edge of causing knocking. Engine control algorithms actually use knocking sensors to regulate spark advance.

Knocking is not trivial to detect. Adding sensors to the combustion chambers is obviously out of the question. The effects of knocking can be sensed as forces acting on the transmission

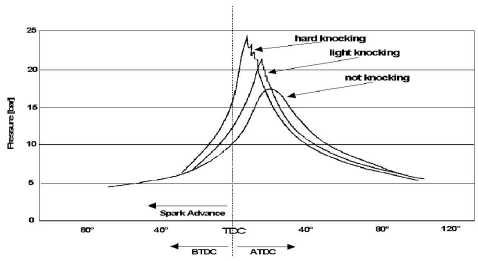

Figure 1. Cylinder Pressure Cycle with and without knock phenomen

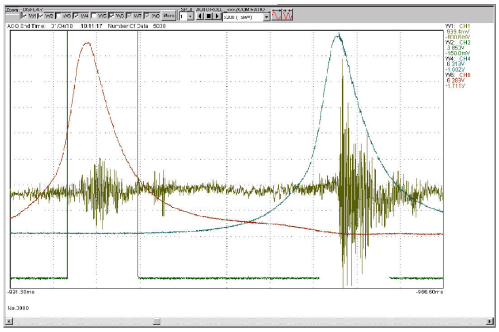

Figure 2. Signal of accelerometer and in-cylinder pressure sensor

due to high frequency components added by knocking to the primary pressure cycle between the TDC (Top Dead Center) and the following $40° \div 60°$ Fig. 1.

The vibrations generated in the combustion chamber are transmitted to the engine mechanical structure and can be measured via a piezoelectric or quartz accelerometer placed on the cylinder block. The maximum measurable frequencies due to knocking on the engine block are limited to 20KHz. However, the accelerometer collects all the vibrations components generated by the entire mechanical system. The measured signal includes: *white noise* components (due to random noises associated to friction and rotation) and *pink noise* of low variance related to mechanical impulsive effects other than knocking. The relation between knock vibration on the in-cylinder pressure cycle and the accelerometer signal is shown in Fig. 2.

A statistical analysis of the accelerometer signal and of the input vibrations shows that a $50KHz$ sampling frequency for the accelerometer output signal is necessary. The sampled signal is then filtered through a 6^{th} order digital elliptic filter (implemented with 15 multiplication and 12 accumulation for each sample) to estimate the accelerometer power spectrum.

To estimate the overall computation load that the knock detection algorithm implies, note that:

- The accelerometer signal is significant only in an angular window of $40° \div 60°$ after TDC. The time window T_w corresponding to the angular window A_w is given by $T_w = (60 * A_w)/(360 * rpm) = A_w/(6 * rpm)$[1], and the consequent number of samples at the sampling frequency F_s is $N_s = T_w * F_s = (A_w * F_s)/(6 * rpm)$.

[1] rpm (round per minute) lies between $2000 \div 5000$.

- At $50KHz$ of sampling rate and $2000\ rpm$, the number of samples in the angular window is between $N_{s,min} = (40 * 50000)/(6 * 2000) = 166$ and $N_{s,max} = (60 * 50000)/(6 * 2000) = 250$.

- Due to the constant angular window of integration, the sample population decreases as the engine speed increases. The ratio of the computation time (the product of number of samples and the elaboration time per sample) to engine period is constant.

3 The Design Methodology

The design flow we follow is related to both platform-based design [9] and model-based design [11]. Platform-based design is a meet-in-the-middle approach where successive refinements from a high level of abstraction carry the design to an implementation platform that is built bottom-up using elements from an appropriate library. The platform performance parameters, such as timing, cost and power consumption, are estimated to guide the selection of the platform instance that implements the design. Model-based design indicates the top-down part of the design. The description of the functionality of the design is done in formal (or semi-formal) language (model). The language is used to simulate and/or analyze the functional design. When the designer is satisfied with his/her solution, the implementation on the platform of choice either as software running on a micro-processor or hardware is automatically (or semi-automatically) extracted from the mathematical model.

The design flow calls for three basic steps: functional design and analysis, platform instance selection, implementation of the functionality using the components of the platform instance.

For engine controllers, functional design consists of the derivation of appropriate control laws whose mathematical properties are to be assessed on the closed-loop system. Stability, controllability and observability are typical properties of interest. These properties are in general proven manually, ensured by rigorous synthesis techniques or heuristically analyzed by means of extensive simulations. The plant (engine and transmission chain) is modeled either as a hybrid system, i.e., a composition of continuous-time components (whose behavior is captured with differential equations), discrete-time and discrete-event components [4], or with a continuous-time averaging model. The controller is the result of the implementation of mathematical control laws. Because of the implementation platforms available today, the control algorithms are mostly implemented in software. For this reason, controllers are mostly described in terms of discrete time or discrete event models that often result from a refinement process of continuous-time control algorithms. The closed-loop system is a complex hybrid system. The capture of the behavior and its simulation is today monopolized by the Matlab and Simulink language and simulation tools [1], from The Mathworks. Simulink uses continuous-time as the basis for hybrid system simulation, thus resulting in rather long computing times since a large number of samples is necessary to represent continuous waveforms.

Platform selection requires a number of experiments that are designed to explore a constrained design space. For each experiment, we need to map the functionality of the design to the components of the platform instance considered and estimate the overall performance. This step is supported by only a few tools (see for example [8]). In this case, the estimation process must be supported by a simulation mechanism that is blindingly fast to be accurate enough. Hence, embedding this system in continuous-time is not feasible. Event-driven simulation of abstract, simplified models is appropriate here [7]. However,

if we do not wish to have inconsistent results, the model used for functional design must be compatible with the model used for performance estimation. Unfortunately, this is not the case today. This phenomenon is only the tip of the iceberg! The problem of heterogeneous system design and analysis requires a deep understanding of *Models of computation* (MoCs) that define how the system specification executes, and of their flat and hierarchical combinations [2] . The problem is exacerbated by the lack of precise documentation about the semantics of the languages used by different tools.

Following platform selection, the functionality has to be mapped into the components of the platform and an implementation derived. Today, this step is, in general, entirely manual. Model-based methods consists in partially automating this process by using tools such as RealTimeWorkshop by The Mathworks and Target Link by dSpace, which generate C code from the Simulink intermediate format directly. Code generation is seen as a strategic response to increasingly longer development time and coding errors.

In our methodology, the crucial step is then transforming from the Simulink simulation model to the VCC internal model so that the behavior is the same in both domains.

4 From continuous time to discrete event

It is readily shown that transforming the hybrid model, called the *D/C model* in the sequel, into a discrete event specification, that we term the *DE model*, is not trivial, in the sense that careful attention need to be placed on the adaptation of the models of computation in the two environments. Consider a discrete time logic operator. In the discrete time domain, the logic operator executes at each sampling time instant and is provided with both input values at each sampling time instant, i.e. input values are synchronized. Instead, in the discrete event simulation, events correspond to signal value changes, so that input values are not synchronized (two inputs need not change at the same time) and the logic operator's functionality need to be modified to keep this into account. For example, in SystemC it is sufficient to locally store old input values, while in Cierto VCC this is not sufficient, for events are asynchronous by definition, and signals might need to be explicitly synchronized by means of additional synchronizing components.

Every simulation in Matlab/Simulink is carried out by embedding continuous time, discrete time and discrete event components in the continuous time domain. The main consequence of this simulation policy is that all signals are synchronized, in the sense that every component "reads" its input values at each simulation time instant, and as a result the simulation is highly inefficient, for discrete time and discrete event components are simulated in continuous time. These points are specific to the Matlab/Simulink environment, for in general mixed-signal simulators [10] gain in efficiency by avoiding continuous time simulation for discrete components.

Hence, the first step to correctly translate the specification is to partition the starting model into three subsystems, each one comprising only homogeneous components (i.e. either continuous time or discrete time or discrete event). Moreover, for reasons of efficiency, continuous signals must be avoided in discrete event simulations, being therefore essential that interacting continuous components be hierachically composed in a single continuous model that will be transformed into a *black-box* component, so that continuous signals are not seen by the discrete event simulator.

[2] Many MoCs have been devised and used. They differ because of characteristics such as ease of modeling, efficency of analysis (e.g. formal verification, simulation, etc.), expressiveness, synthesizability, compositionality [12, 5].

Let us assume that the original specification be made up of l continuous time components, m discrete time components and n discrete event components, possibly intensively interacting with one another. We further assume, as discussed above, that the l continuous components do not directly interact with one another. The interface signals between continuous and discrete components are adapted by means of samplers, interpolators (e.g. zero-order hold) and event generators (e.g. differentiators), globally referred to as *adapters*.

The transformation into the discrete event domain is carried out by encapsulating each of the l continuous components together with its adapters in a *black-box* component, that is thus discrete at its interface (the computation inside the black-box is still continuous). The l black-box components are triggered by p ($p \leq m$) different sampling clocks. Discrete components are easily embedded in the discrete event domain, though a transformation might be needed to adapt to the discrete event simulator's MoC. We refer to this problem as the *MoC compatibility issue*. Transformed discrete components are referred to as *white-box* components, in contrast to black-box ones.

Among the continuous time components there are some that do not interact with discrete event components. We call these *PS (purely sampled) components*. All other continuous components are termed *NPS (non-purely sampled) components*. PS components can be simulated in the discrete event simulator as any other white-box component, i.e. their current local state is "never in the future" (i.e. ahead of the current simulation time). Instead, the local state of NPS components need to be "never in the past" (i.e. behind the current simulation time), for they can generate unpredictable events in the simulation. For these components it is necessary to implement techniques such as simulation back-tracking and separation of posting and commitment of events.

These techniques can be realized locally inside the component whenever the discrete event simulator provides a means of component's self-scheduling. This is provided for example in *Cierto VCC* [8] by the *async-event* and in *SystemC* [2] by a self connection. The back-tracking technique can be implemented by locally copying the current state of the component before taking the transition to the next event in the future, while the separation of event posting and commitment can be done by self-scheduling (i.e. by committing an event on the self-connection) at the future event time and by local (at the component level) event queue handling. If new input events are received before the self-activation, then back-tracking can be used to remove the current self-scheduled events, recompute the future events and commit the new events on the self-connection.

In the discrete event domain the simulation follows the discrete event simulation paradigm [7], according to the following simple iterative steps. Note that sampling clocks might be implemented both as timers, if provided by the simulator, or as NPS black-box components.

1. integration of the NPS black-box components up to the next unpredictable event;
2. integration of the PS black-box components up to the current time;
3. if no current event is in the event queue, the simulation is continued up to the next event in the queue or the next sampling event from any of the sampling clocks;
4. the next instantaneous events from the queue are consumed according to the model of computation embedded in the discrete event simulator;
5. go to point 1.

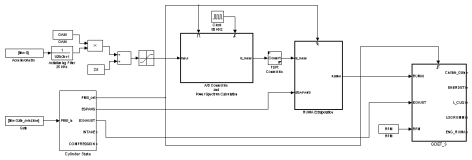

Figure 3. Top view of the D/C model in Simulink

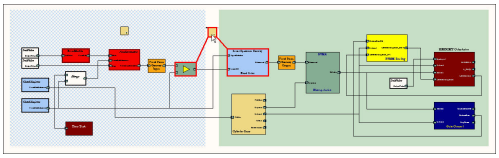

Figure 4. Flat view of the DE model in Cierto VCC

The transformation induced by these rules and simulation algorithm has some limitations. For instance, the described discrete event simulation supports only strictly causal systems, hence it does not support algebraic closed loops, frequently used in continuous/discrete time domain. This limitation is not strongly affecting the design process. Indeed, we want to address control algorithms that can be actually implemented, so that algebraic loops must be resolved before the transformation.

Other important limitations need to be considered. In general, any model transformation alters the basic model properties, hence mathematical properties verified on the original model might not hold on the transformed one. For all these cases, computational constraints, under which the properties have been validated, must be propagated to the discrete event domain. While a more formal treatment of the transformation is necessary for a general understanding of the limitations, it has been considered out of the scope of this paper.

The Simulink specification of the knock detection algorithm was translated into a VCC behavioral description through the following steps:

1. the specification was initially reorganized into a number of hierarchical blocks inside Simulink;

2. for each leaf block in the hierarchy, C code for the block was automatically generated using or Mathworks/Real-Time-Workshop or DSPACE/TargetLink [3];

3. each C block was encapsulated into a WhiteBoxC[3] shell suitable for integration into the VCC environment;

4. the WhiteBoxC blocks in VCC were interconnected in the VCC environment to reconstruct the original specification in the new semantic domain provided by VCC (Fig. 4).

Step 3 is crucial in the adaptation of the original specification to the new semantic domain defined by the target tool, Cierto VCC in this case. This is generally due to the MoC compatibility issue. For example, digital blocks in Simulink are defined through the presence of *triggers*, which activate the corresponding blocks. In VCC behavioral descriptions, every input to each block is considered as a trigger, therefore block executions in the two environments are not in one-to-one correspondence. Moreover, the specific firing rule in VCC (simultaneous inputs to a block are computed one at time in unspecified order) introduces other sources of difficulties in the translation

[3]WhiteBoxC are leaf block components specified in a subset of C language.

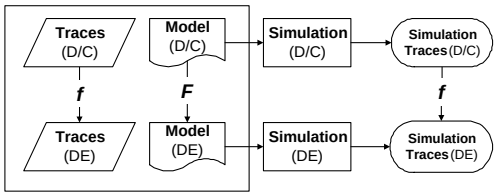

Figure 5. Relations between models, simulations and traces

process, if functional equivalence is desired. In contrast, SystemC does not suffer from this firing rule problem (input values can be locally synchronized), but this is only because we started from a Matlab/Simulink specification, whose model of computation well adapts to that of SystemC.

5 Functional validation

When the MoC compatibility issue applies, the problem of the validation of the transformed model against the original model need to be addressed. Indeed, the need of ensuring that the two models are equivalent stems from the requisite of correctly propagating the system properties from the original model down to implementation.

To address this problem two approaches are possible: correct-by-construction transformations and equivalence verification. The former approach is fairly complex, especially when dealing with different time domains and MoCs, so that only simple properties can be rigorously treated. The latter approach on the other hand is computationally intractable and is usually replaced by functional validation, which is based on simulation, thus being a heuristic non-exact technique. This is the current approach in industrial design and requires a great deal of designer's interaction. Equivalence by simulation is defined by means of *equivalence* of simulation traces. This semi-formal definition of equivalence is based on the comparison of simulation results, requiring a deep understanding of the execution semantics of both simulators and of the transformation between the models. In general these simulation traces are heterogeneous, for they are drawn from different time domains.

Fig. 5 shows the relationship between models, simulations and simulation traces. The choice for a transformation between the models, called $\mathbf{F}$ in Fig. 5, is paired with a choice of a transformation $\mathbf{f}$ on the set of all traces. In more intuitive terms, the transformation $\mathbf{f}$ corresponds to a set of rules for converting continuous and sampled signals (or more in general traces) into sequences of events called *event signals* (or more in general discrete event traces). For example, a square waveform can be transformed into an event signal by defining an event at each change of value.

Simulations define time interpretations of traces, called simulation traces, according to the specific MoC of the corresponding simulators. For example, while discrete event traces only define a partial order between events, their corresponding simulation traces are made up of a total ordered set of events, according to the simulator's MoC. In other words, the discrete event simulator defines a mapping from partial ordered traces to total ordered traces defined on the underlying simulation time.

It is clear that in order to compare simulation traces, we need to abstract from the underlying simulation time. The relation between the simulation traces can then be established in terms of the transformation $\mathbf{f}$ applied to the set of simulation traces:

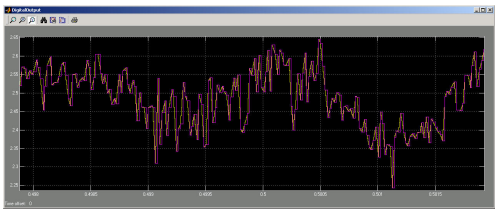

Figure 6. Accelerometric input validation

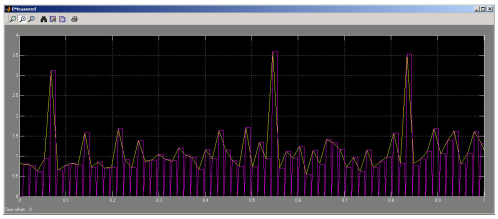

Figure 7. Energy computation validation

two simulation traces $T_{D/C}$ and T_{DE} are said to be equivalent if $\mathbf{f}(T_{D/C}) = T_{DE}$.

The automation of the transformation process can be addressed by defining a class of *functionally invariant transformations* between the models, i.e. such that the equivalence condition, namely $\mathbf{f}(T_{D/C}) = T_{DE}$, be satisfied by construction. The way to the definition of such transformations can be paved by introducing specific design styles at the continuous/discrete time level of specification, which the system model must comply with. Adapting a legacy specification to a given design style is in general an easier problem, for it requires an homogeneous (in terms of specification environment, time domain and model of computation) transformation of the model specification. Note that automated transformations may ensure exact equivalence of closed systems (i.e. where the environment is comprised in the model), for it does not rely on equivalence by simulation traces.

From the discussion above it is clear that functional equivalence is far from being straightforward and, even after automatic translation, the viability of functional validation still remains of inestimable value. This has been made possible in VCC by the creation of *matlab-probes* that allow to provide VCC simulation data in **.mat* format, suitable for integration into Simulink functional simulations. The comparison of the simulation data within the Simulink environment is shown in Fig. 6 and Fig. 7.

6 Design exploration

The application described in Section 2 presents interesting implementation trade-offs that must be accurately investigated especially as regards its coexistence on the same hardware platform with other important engine control functionalities. In particular, real time aspects (due to filtering functions) of the knock detection functionality require early assessments of overall computational loads in order to carefully define the hardware partition of the platform so to minimize cost without compromising performance. In the sequel, an example of architectural exploration is presented, showing how early decisions in the design process can be taken by taking advantage of early design space exploration.

Behavior adaptation and validation: once the behavioral description was made available in VCC, it has been adapted

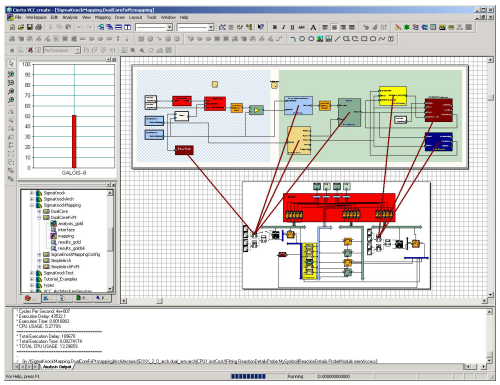

Figure 8. Performance simulation: target system

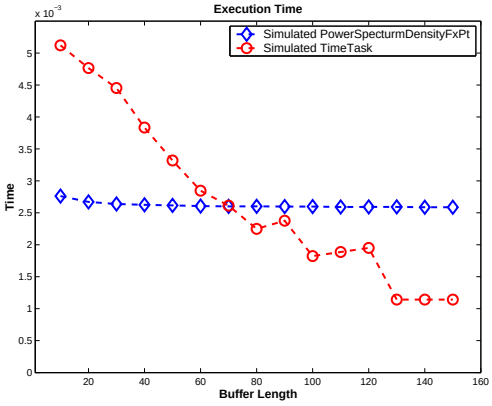

Figure 9. Results of performance simulation

to perform architectural exploration. In particular, the *Power Spectrum Estimation* (PSE) has been encapsulated into a parameterized *for-loop* in order to perform the computation using a bufferization of the samples. The bufferization is necessary to migrate the computationally intensive PSE task from hardware to software, allowing to trade off computational power for silicon area. The parameterization of the loop enables the simulation of several buffer lengths to look for an optimum value. The modified specification has been then behaviorally validated against the original specification through functional simulation in VCC and Simulink (Fig. 6 and Fig. 7).

Mapping of behavior onto architecture: before architectural exploration we need to define the complete system model by mappping the behavioral specification onto an architectural model, in this case the JANUS [6] (a dual-processor micro-controller based on ARM7TDMI) VCC architectural performance model (Fig. 8). The PSE and most of the knock detection functionality has been mapped to software (on the two processors). The buffer between the *A/D converter* and the PSE filter has been simulated using a *behavioral memory* [8], i.e. a reference to a set of memory elements (e.g. RAM, registers, etc.) of an architectural model.

Architectural exploration: the main purpose of architectural exploration in this application has been established to be the estimation of the computational load required by the specification and the analysis of the interactions of the knock detection with other engine control functionalities. To accomplish this, we simulated the interaction of the knock detection functionality with a task, that we will call *TimeTask*, of a fixed ex-

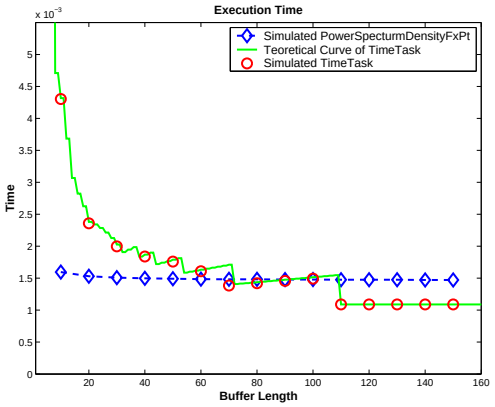

Figure 10. Comparison with theoretical curve

ecution time of $1ms$ (when not preempted), activated on every TDC. The *TimeTask* is given lower priority than the PSE task, so that its actual exectution time depends on the number of interruptions due to preemption by the higher priority task. After the mapping phase, a set of buffer length values has been considered for simulation (Fig. 8).

Simulation results: the computational load of the knock detection algorithm is dominated by the PSE task. The computational load per sample has been measured to be 310 clock cycles on the ARM7TDMI processor. If N_s is the number of samples per acquisition, the pure CPU time for this task per acquisition is given by $T_{PSE} = (N_s * 310)/F_{ck}$. We used acquisitions of about 200 samples at $2000rpm$, so that $T_{PSE} = (200*310)/(40MHz) = 1.55ms$. Each acquisition need to be carried out between two TDC occurrences, i.e. the allotted time is $T_{tot} = 60/(N_{TDC} * rpm) = 60/(2*2000) = 15ms$. Hence, the pure CPU load per acquisition for the PSE functionality is $1.55/15 = 10\%$. To this the operating system overload need to be added. Since the operating system load is more than significant, we considered sampling bufferization to reduce context switching activity. We explored several buffer dimensions by taking the buffer length as a parameter in the simulations. As expected the *TimeTask*'s actual execution time decreases as the buffer length increases (Fig. 9), showing that the smallest buffer length at which the task execution time is minimum is 130 (65 samples, for the buffer is divided in two segments to avoid sample overwriting).

Validation of the simulation process: the performance results where validated using a simplified mapping, where specifically the interactions between the *TimeTask* and the PSE task are carefully investigated. For such a mapping we were able to compute by paper and pencil the exact execution time curve for the *TimeTask* (Fig. 10), showing that the smallest buffer length for which the task execution time is minimum is 110 (55 samples). Fig. 10 shows that in the simplified mapping the execution time curve obtained by performance simulation in VCC closely approximate the theoretic curve, providing the same result for the smallest buffer length ensuring the minimum task execution time.

7 Conclusions

The design of a knock detection algorithm and of its implementation has been addressed. In our flow, a mathematical model of the system functionality (e.g. a control algorithm) is defined by the designer, who elaborates this functionality until the desired mathematical properties are met. The mathematical specification obtained is then translated, possibly automatically, into a behavioral specification within a discrete event simulation environment and a set of platforms are considered for implementation. At this step behavioral simulation of the new specification is used to obtain data for the functional validation agaist the original specification. Several mappings of the behavioral specification on the architectural models are then tried out and merit figure analysis is used to chose heuristically an optimized system mapping.

The design flow required the use of different tools at different levels of abstraction. The most challenging problem in this exercise was to relate the different models of computation used by the two tools. This is an essential problem in system-level design where a unified design environment is still in its infancy. We indicated the problems to solve and described how we mapped a model of computation into another. Some experimental results on platform selection were given.

References

[1] http://www.mathworks.com/.

[2] http://www.systemc.org.

[3] http://www.dspaceinc.com/en/Products/PCGS.htm.

[4] A. Balluchi, L. Benvenuti, M. D. Di Benedetto, C. Pinello, and A. L. Sangiovanni-Vincentelli. Automotive engine control and hybrid systems: Challenges and opportunities. *Proceedings of the IEEE*, 88, "Special Issue on Hybrid Systems" (invited paper)(7):888–912, July 2000.

[5] S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[6] A. Ferrari, S. Garue, M Peri, S. Pezzini, L.Valsecchi, F. Andretta, and W. Nesci. The design and implmentation of a dual-core platform for power-train systems. In *Convergence 2000*, October 2000.

[7] S. Ghosh and E. Debenedictis. An asynchronous distributed discrete event simulation algorithm for cyclic circuits using a data-flow network. In *Proceeding of the International Conference on Systems, Man and Cybernetics*, October 1991.

[8] Cadence Design Systems Inc. Cierto vcc user guide, 1998.

[9] K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, 19(12):1523–1543, December 2000.

[10] S. Mayes. Mixed-signal simulation issues for top-down design. In *Electronics Engineer, Design Corner II*, October 1997. http://www.jat.co.kr/eda/saber/topdown.pdf.

[11] Scott Ranville. Practical application of model-based software design for automotive. In *Society of Automotive Engineers (SAE) Conference*, May 2002.

[12] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal models for embedded system design. *IEEE Design and Test of Computers*, 17(2):14–27, April 2000.