Embedded System Engineering Using C/C++ Based Design Methodologies

Claudio Talarico[†], Aseem Gupta[‡], Ebenezer Peter[†], and Jerzy W. Rozenblit[†]

[†]Department of Electrical and Computer Engineering
The University of Arizona
Tucson, AZ 85721-0104, USA

[‡]Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA 92697, USA

Abstract

This paper analyzes and compares the effectiveness of various system level design methodologies in assessing performance of embedded computing systems from the earliest stages of the design flow. The different methodologies are illustrated and evaluated by applying them to the design of an aircraft pressurization system (APS). The APS is mapped on a heterogeneous hardware/software platform consisting of two ASICs and a microcontroller. The results demonstrate the high impact of computer aided design (CAD) tools on design time and quality.

1. Introduction

In recent years, growing system complexity and shrinking time-to-market requirements have resulted in a strong need for new design methods and tools. In order to keep pace with the increased system complexity, designers must work at a higher level of abstraction [1]. Depending on the abstraction level (namely, the number of details used to model the system) different concerns can be addressed and solved. At each step of the design process, the key to cope with complexity is to model the systems, only with the minimum number of details needed. Tools support is needed throughout all steps of the design flow, from the formal specification of the system to its physical implementation.

Traditionally, the design of embedded systems has been carried out by decomposing and allocating the system to hardware and software, then allowing separate hardware and software design teams to design their respective parts, and finally integrating hardware and software. This separation of design tasks leads to the potential for initial design mistakes to be carried until the integration phase, where they are much more difficult and costly to correct. We address this issue by using the same high level language, i.e. C/C++, for describing both hardware and software. The idea is to keep the hardware and software design activities tightly coupled [2].

Given system functionality the goal is to find the best architecture and the best partitioning of functionality into the architectural components. Here, the term architecture is used to mean not only the set of hardware and software components forming the system but also their topology. Starting from the same specification, many different architectures and functionality-architecture mapping may be produced. The exploration of all these alternatives requires the ability to rapidly estimate the performance resulting from a particular partitioning. In order to evaluate performance, we cannot afford to synthesize and simulate at the cycle level, every possible design alternative. The use of a C/C++ based methodology simplifies the system modeling task and maintains computation time within feasible ranges [3]. As a result: 1) design assessment can be done much earlier in the design cycle, and 2) execution time to explore different design tradeoffs is much shorter.

The rest of this paper is organized as follows. First we propose a top-down C/C++ based system design process that can be used from executable specification of the system to silicon. Then, we illustrate the viability of our approach by introducing an APS design example and applying different C/C++ design based methodologies on it. Last we compare the results obtained and provide conclusions.

Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)
0-7695-2308-0/05 $20.00 © 2005 **IEEE**

2. Design Flow

Designing an embedded system requires many capabilities: 1) describing the interaction between the system and the external environment, 2) describing the system architecture, 3) modeling the behavior of hardware and software components forming the system, 4) describing system constraints and requirements, 5) describing the test scenarios used to simulate the system, and 6) defining a set of gauges to measure various performance metrics during simulation execution. As a consequence, the complexity of the design process is determined by the semantic and syntax of the system level design language (SLDL) adopted as implementation vehicle.

System level design approaches can be broadly classified into three groups: system-level synthesis, platform-based design, and component-based design [4]. In *system-level synthesis* the design starts by describing system behavior. Then system architecture is generated by the behavior and finally a register transfer level (RTL) model or an instruction set simulation (ISS) model are generated depending whether the behavior is going to be mapped on hardware or software. In *platform based design* the system behavior is mapped to a predefined system architecture, instead of being generated from the behavior as in the system level synthesis approach. In *component-based design* the task of selecting components and combining them in a proper architecture is not defined a priori. Compared with platform based design this solution provides a higher flexibility, however it requires a well developed database of components (also known as intellectual properties or virtual components) before it can be effectively implemented.

In general a SLDL requires two essential attributes: 1) it should support modeling at all levels of abstraction, from purely behavioral un-timed models to cycle accurate RTL/ISS models, and 2) the models should be executable and simulatable, so that functionality and constraints can be validated. The two most commonly used SLDL in embedded system engineering are: SystemC [5] and SpecC [6].

The objective of system level design is to generate system implementation from behavior. To that end, we propose a design process based on the use of finite state machines as mathematical model of computation to describe behavior, and either SystemC or SpecC as implementation vehicle. In order to reduce the complexity of system design a number of intermediate models are built. Each intermediate model describes specific design tasks and objectives and can be independently executed and simulated.

The design process we propose belongs to the system-level synthesis group and is illustrated in Figure 1. Here, we decompose the design process in four main steps: 1) specification modeling, 2) architecture modeling, 3) communication modeling and 4) implementation modeling. The *specification model* is a formal description of the system functionality, but does not carry any implementation details, and it is un-timed in terms of both computation and communication. After the specification model is analyzed and validated the system functionality is partitioned and the various partitions are mapped to different components. The *architecture model* defines the final set of components into which the functionality is mapped and its topology. The execution delays of the processes assigned to the components are modeled by means of unit delta delays, while communication among components is modeled via message passing. Hence at this level, computation is approximate-timed, while communication is un-timed. The *communication model* defines the protocol and the accurate timing followed by the various components to exchange information. The *implementation model* represents the hardware components in terms of resister transfers and the software components in terms of instruction set architecture. At this level, computation components as well as communication components are refined down to individual clock cycles.

3. A Design Example

To illustrate the details of our approach we apply it to the design of an Aircraft Pressurization System (APS). As altitude increases it becomes increasingly difficult for humans to handle the air pressure. Atmospheric pressure decreases as altitude increases (Figure 2), so in aircrafts flying at high altitudes, the cabin pressure has to be controlled. The APS is an automated control system that controls the pressure in an aircraft cabin within comfortable limits. If pressure control is not provided many physiological problems may occur, for instance, ear-ache and gastro-intestinal problems.

The inputs to the APS are ambient pressure, cabin pressure and a signal that indicates emergency.

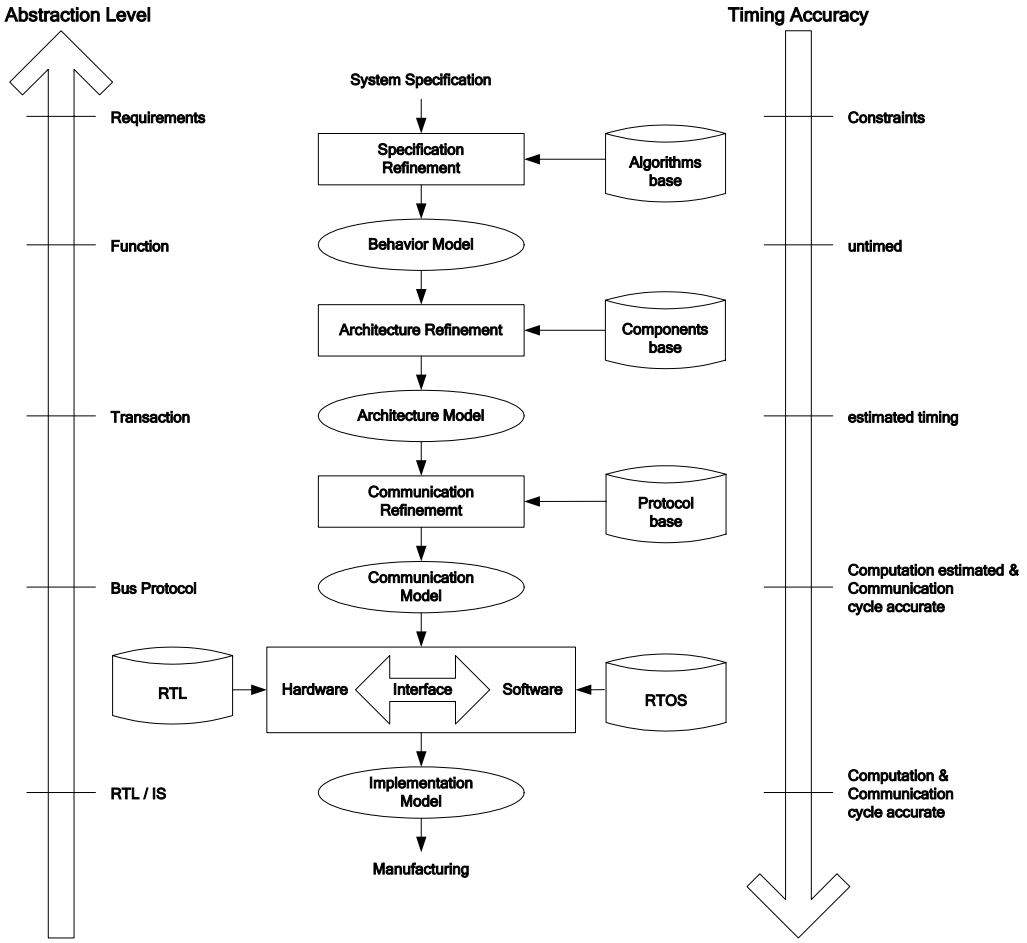

Figure 1. Design Flow

The outputs of the APS are signals that control three different valves namely pressure valve, release valve and dump valve. A high signal to any of these valves causes them to open while a low signal closes them. The pressure valve is used to pump air into the cabin to raise the cabin pressure. The release valve causes slow and steady release of cabin pressure. The dump valve is opened in case of emergency for a quick decay in cabin pressure. The inputs and outputs of the aircraft pressurization system are shown in Figure 3. The operation of the APS is illustrated by the detailed block diagram shown in Figure 4, and the finite state machine diagram in Figure 5.

If the aircraft is flying at altitudes of less than 5000 ft the cabin can be comfortably maintained at ambient pressure and the APS is said to be in *normal* state of operation. In *normal* state the pressure valve is closed and the release valve is kept open to equalize the cabin pressure to ambient pressure.

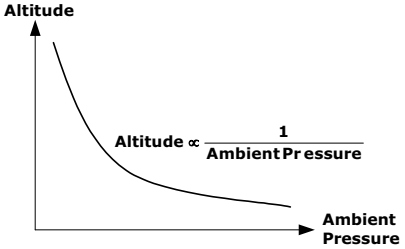

Figure 2. Altitude vs. Ambient Pressure

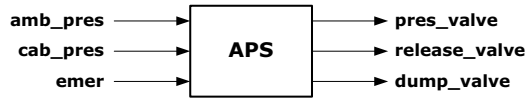

Figure 3. Basic block diagram of APS

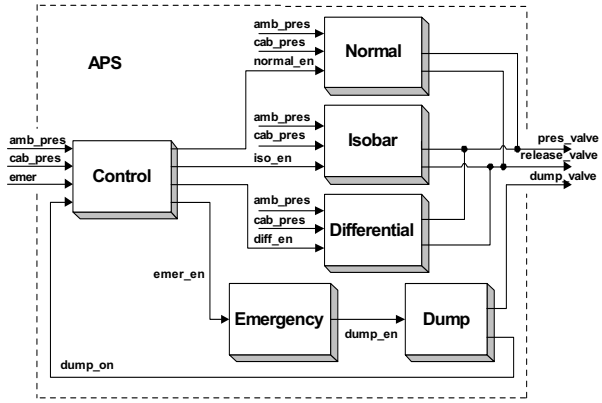

Figure 4. Detailed block diagram of APS

As the aircraft flies to altitudes higher than 5000 ft, there is a fall in ambient pressure which makes it necessary to pressurize the cabin. The APS moves itself into *isobar* state to maintain a constant cabin pressure equivalent to the ambient pressure at 5000 ft, even when the real altitude is greater.

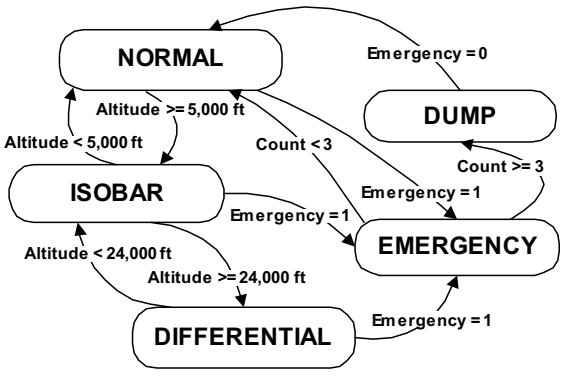

Figure 5. Finite state machine digram of APS

In *isobar* state, if necessary, the pressure valve is opened and the release valve remains closed. When the altitude of the aircraft reaches 24000 ft, the APS changes its state of operation from *isobar* to *differential*. In *differential* state the APS maintains a differential of 6.5 psi between cabin pressure and ambient pressure. This is accomplished either opening or closing the pressure or release valves.

The APS should be able to handle an emergency situation of excessive cabin pressure during any state of operation. The APS goes into *emergency* mode if the emergency signal is high. The emergency block checks if the emergency was real or just a glitch. If the emergency signal remains high continuously for 3 clock cycles the APS goes to *dump* state, else APS returns to *normal* state to resume operation. In the *dump* state the dump valve is opened to cause a quick decay of the cabin pressure. Once the cabin pressure becomes less than or equal to ambient pressure and the emergency signal goes low, the APS goes back to *normal* state of operation.

4. C/C++ Based Design Methodologies

SystemC is a modeling platform consisting of C++ class libraries and a simulation kernel for designing at the system and register transfer level. Besides, providing a common high-level language, for modeling, analyzing and simulating an embedded system, it can be also linked to commercial tools such as Synopsys design compiler [7], for hardware synthesis.

SpecC is a super-set of the C language. It is a complete language, not just a library, and it was specifically conceived for the specification and design of digital embedded systems. Detailed information on the syntax and semantics of SpecC and SystemC is available in ref. [6] and [8].

In this section, we briefly compare C++ based methodology (SystemC) and C based methodology (SpecC) with respect to three aspects: 1) capability of modeling functionality, 2) capability of modeling the transfer of information between functional blocks, and 3) capability of modeling the execution sequence among functional blocks.

Both SpecC and SystemC support hierarchical modeling of system behavior. In SpecC, the term behavior indicates a consolidate representation of both functionality and structure. Behavior and structure of the system are represented by a hierarchy of behaviors. A leaf behavior may contain hierarchical calls to functions but it does not contain any further

sub-instance. SystemC isolates functionality and structure into processes and modules respectively

SpecC models data transfer among behaviors through the use of *variables* or *channels*. SystemC supports data transfer by connecting module ports through either *signals* or *channels*. A channel is a class that encapsulates communication. In SpecC a channel consists of a set of variables and functions (also called methods), which operate on the variables and define the communication protocol. Similarly, in SystemC, a signal is a type of a channel. SystemC signal (sc_signal) is a primitive channel with no user defined communication behavior included. Communication is done in similar way, using channels, in both SystemC and SpecC. The difference in using variables and signals is that, changes on variables are scheduled immediately, while changes on signals are queued and scheduled at the occurrence of the next event (i.e., the value of the signal is updated only after a delta delay). SystemC does not allow binding of variables to ports of modules, thus the use of variables for data transfer between processes in different modules is not permitted.

In SpecC the order of execution is by default sequential, however two mechanisms are provided to alter the execution sequence: 1) static scheduling and 2) dynamic scheduling. In static scheduling the sequence of execution is explicitly specified using dedicated constructs *par* (for parallel execution), *pipe* (for pipelined execution), *fsm* (for Finite State Machine execution). For dynamic scheduling, both SpecC and SystemC rely on the data type *event* and the *wait* and *notify* statements for synchronization between behaviors. A process can wait and notify events. When a pending event is notified the process starts/resumes execution. However, in case of SystemC, ports of modules cannot be connected through an event (*sc event*), therefore the *event-wait-notify* cannot be used for synchronization between processes in different modules. Designers have to encapsulate events into a channel in order to achieve synchronization between such processes.

In SpecC, static scheduling permits to precisely determine the execution sequence and as a consequence make architecture exploration much easier than with SystemC. In addition, since SystemC is a C++ library extension, the computation needs of the system under design are tightly coupled with the computational needs of the SystemC kernel, so the profiling of the model becomes prohibitive.

At the hardware level, in order to specify cycle-accurate finite state machines SpecC provides the construct *fsm,* while SystemC provides two mechanisms: 1) implicit modeling using the class SC_THREAD and *wait* statement, and 2) explicit modeling using the class SC_METHOD and *switch* statement.

At the software level, in order to obtain C code compilable and executable on the target microprocessor from SpecC code, users need to remove all SpecC specific constructs (par, pipe, fsm, wait, etc.). The equivalent task in SystemC needs two steps, converting SystemC code to C++ code by removing SystemC specific constructs (module, port, channel), and converting C++ code to C code.

5. Results

In order to validate the approach proposed we applied it to the design of an APS. The APS is mapped on a heterogeneous hardware/software platform consisting of two ASICs and a microcontroller, thus the 6 blocks forming the system can be implemented in 3^6=729 different ways.

The design of the APS has been completed both in SystemC and in SpecC. The effectiveness of the two methodologies has been evaluated with respect to many facets.

SpecC and SystemC are comparable in terms of time required to complete the design, and time taken to execute the system model. SystemC provides better support for the register transfer level (RTL) model of hardware design. However, the program length (number of lines of code) of the SystemC model is about 10% longer than the SpecC model. SystemC, being a C++ library, is hard to profile accurately, hence is not very suited for architecture exploration. SpecC, being a super-set of C language, is superior for architecture exploration both in terms of profiling and determination of execution sequence. As a result when many different design choices are possible the impact of SpecC in helping to optimize design time and design quality is considerably higher than SystemC. Fig. 6 through Fig.8 illustrate some of the capabilities provided by SpecC: Fig.6 shows one of the many possible mappings of the APS into the two ASICs and the microcontroller available for implementation. The small FSM symbol before behavior *MyFSM*, is automatically generated by SpecC and indicates that

the behavior is a FSM and is a clean behavior. Behaviors are defined hierarchically; each behavior can also contain a number of behavior instantiations of other behaviors. In clean leaf behavior, there is only a sequence of statements, without any behavior instances. Fig 7 summarizes the profiling of the system model. It shows all the variables used by our system, number of code expressions, computation operations etc. Fig. 8 indicates the computations performed by each component of the system.

6. Conclusion

Although both C and C++ design methodologies have many similarities, we observed that C based methodology is easier to use, and is better suited for architecture exploration. This results in smaller design time and better design quality, especially when there are many design alternatives to consider. On the other hand, currently, C++ methodology is better linked with commercial hardware synthesis tools.

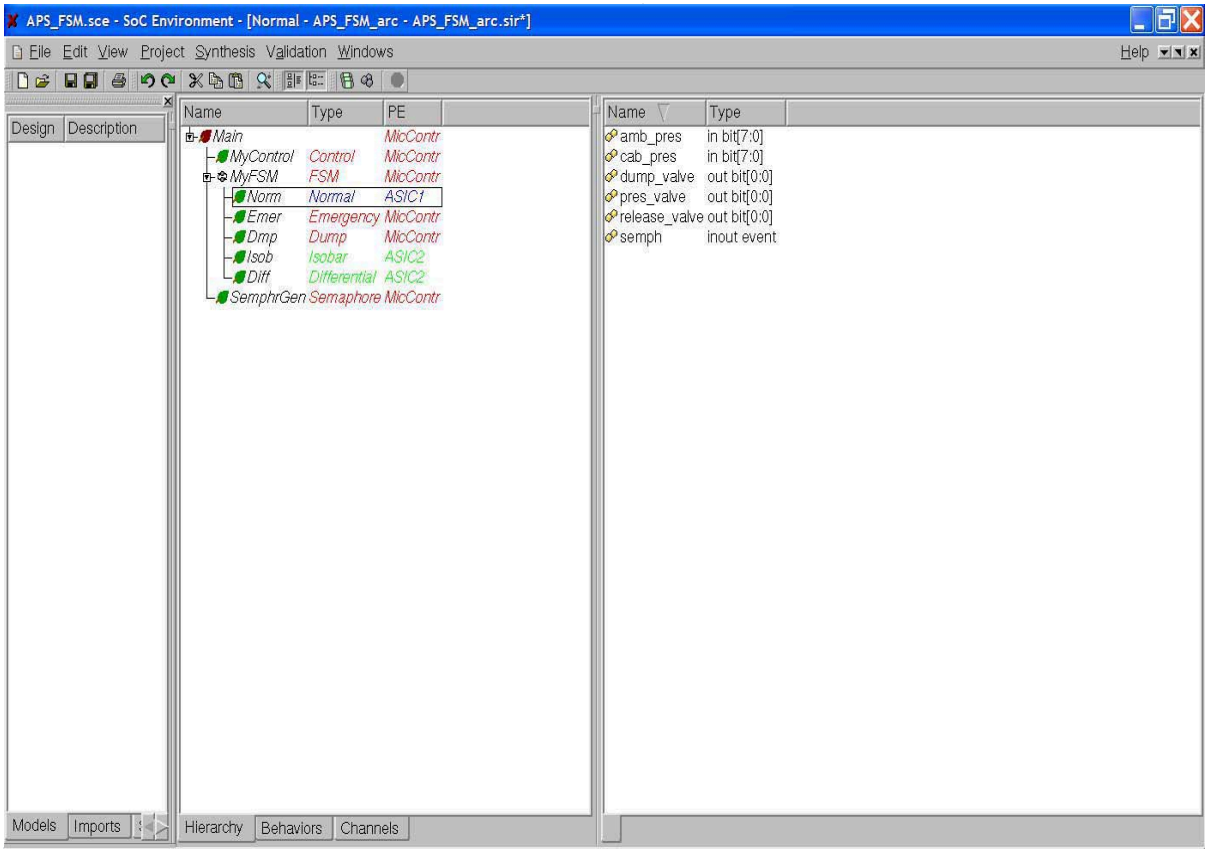

Figure 6. SpecC: mapping of the system into two ASICs and one Microcontroller.

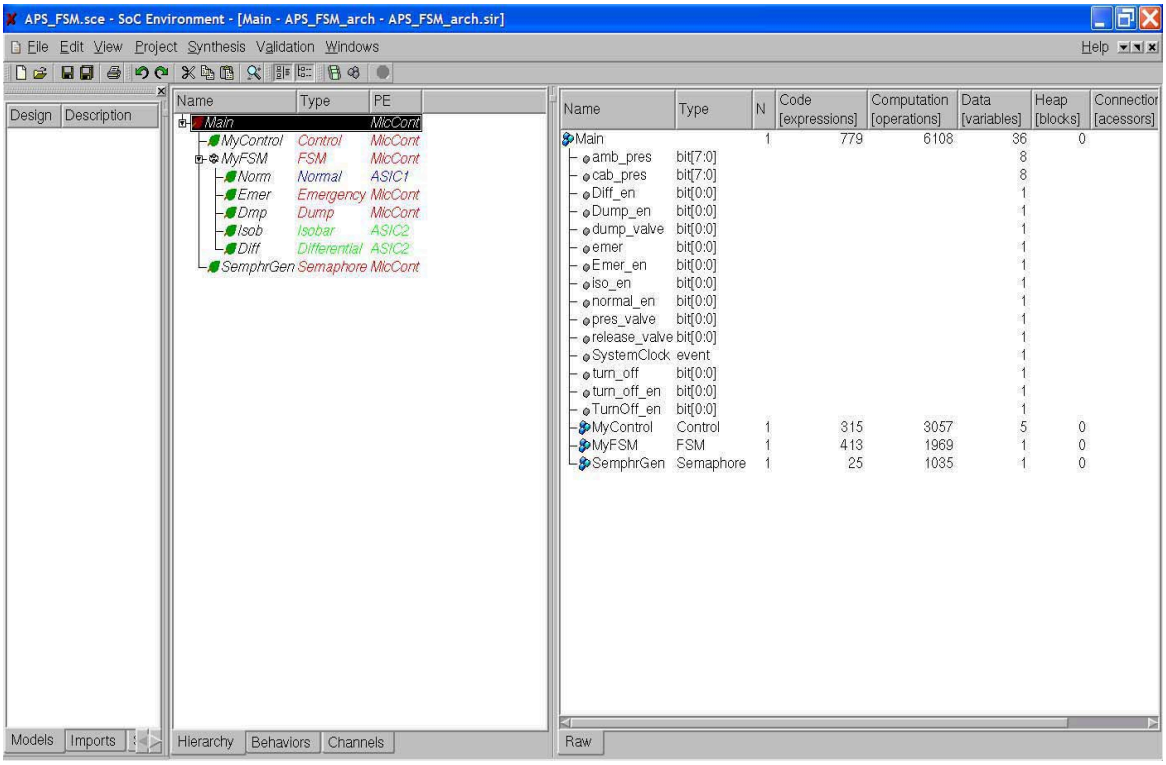

Figure 7. SpecC: profiling of the system model.

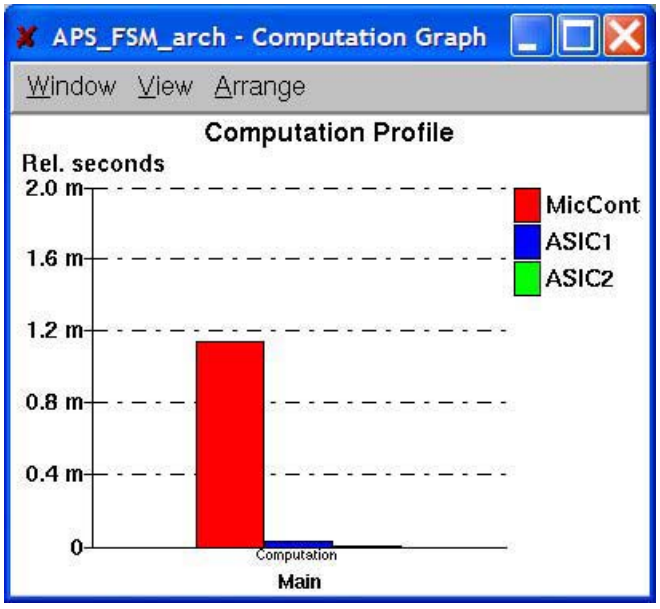

Figure 8. SpecC: computation profile for each component of the system.

7. References

[1] 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 Trans. CAD Integrated Circuits and Systems*, vol. 19, no. 12, 2000, pp. 1523-1543.

[2] C. Talarico, J.W. Rozenblit, A. Gupta, and E. Peter, "Performance Analysis of Embedded Systems with SystemC," *Proc. Int'l Conf. Computing, Communications and Control Technologies* (CCCT 2004), IIIS Press, 2004, pp. 46-51.

[3] T. Givargis, F. Vahid, and J. Henkel, "System-Level Exploration for Pareto-Optimal Configurations in parameterized System-on a-Chip," *IEEE Trans. VLSI Systems*, vol. 10, no. 4, 2002, pp. 416-422.

[4] D.D. Gajski, J. Zhu, R. Doemer, A. Gerstauler, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic, 2000.

[5] Open SystemC Initiative, *Functional Specification for SystemC 2.0*, 2002, http://www.systemc.org

[6] SpecC Open Technology Consortium, *SpecC Language Reference Manual, version 2.0*, 2004, http://www.specc.org

[7] Synopsys, Design Compiler Reference Manual, June 2003

[8] Open SystemC Initiative, SystemC 2.0 Version 2.0 User's Guide, 2002, www.systemc.org