Systematic Embedded Software Generation from SystemC

F. Herrera, H. Posadas, P. Sánchez & E. Villar
TEISA Dept., E.T.S.I. Industriales y Telecom., University of Cantabria
Avda. Los Castros s/n, 39005 Santander, Spain
{fherrera, posadash, sanchez, villar}@teisa.unican.es

Abstract

The embedded software design cost represents an important percentage of the embedded-system development costs [1]. This paper presents a method for systematic embedded software generation that reduces the software generation cost in a platform-based HW/SW codesign methodology for embedded systems based on SystemC. The goal is that the same SystemC code allows system-level specification and verification, and, after SW/HW partition, SW/HW co-simulation and embedded software generation. The C++ code for the SW partition (processes and process communication including HW/SW interfaces) is systematically generated including the user-selected embedded OS (e.g.: the eCos open source OS).

1. Introduction[1]

The evolution of technological processes maintains its exponential growth; 810 Mtrans/chip in 2003 will become 2041 Mtrans/chip in 2007. This obliges an increase in the designer productivity, from 2.6 Mtrans/py in 2004 to 5.9 Mtrans/py in 2007, that is, a productivity increase of 236% in three years [2]. Most of these new products will be embedded System-on-Chip (SoC) [3] and include embedded software. In fact, embedded software now routinely accounts for 80% of embedded system development costs [1].

Today, most embedded systems are designed from a RT level description for the HW part and the embedded software code separately. Using a classical top-down methodology (synthesis and compilation) the implementation is obtained. The 2001 International Technology Roadmap for Semiconductors (ITRS) predicts the substitution (during the coming years) of that waterfall methodology by an integrated framework where codesign, logical, physical and analysis tools operate together. The design step being where the designer envisages the whole set of intended characteristics of the system to be implemented, system-level specification acquires a key importance in this new design process since it is taken as the starting point of all the integrated tools and procedures that lead to an optimal implementation [1][2].

The lack of a unifying system specification language has been identified as one of the main obstacles bedeviling SoC designers [4]. Among the different possibilities proposed, languages based on C/C++ are gaining a wider consensus among the designer community [5], SystemC being one of the most promising proposals. Although, the first versions of SystemC were focused on HW design, the latest versions (SystemC2.x [6]) include some system-level oriented constructs such as communication channels or process synchronization primitives that facilitate the system specification independently of the final module implementation in the HW or SW partition.

Embedded SW generation and interface synthesis are still open problems requiring further research [7]. In order to become a practical system-level specification language, efficient SW generation and interface synthesis from SystemC should be provided. Several approaches for embedded SW design have been proposed [8] [9] [10]. Some of them are application-oriented (DSP, control, systems, etc.), where others utilise input language of limited use.

SoCOS [11][12] is a C++ based system-level design environment where emphasis is placed on the inclusion of typical SW dynamic elements and concurrency. Nevertheless, SoCOS is only used for system modeling, analysis and simulation.

A different alternative is based on the synthesis of an application-specific RTOS [13][14][15] that supports the embedded software. The specificity of the generated RTOS gives efficiency [16] at the expense of a loss of

[1] This work has been partially supported by the Spanish MCYT through the TIC-2002-00660 project.

verification and debugging capability, platform portability and support for application software (non firmware). Only very recently, the first HW/SW co-design tools based on C/C++-like input language have appeared in the marketplace [17]. Nevertheless, their system level modeling capability is very limited.

In this paper, an efficient embedded software and interface generation methodology from SystemC is presented. HW generation and cosimulation are not the subject of this paper. The proposed methodology is based on the redefinition and overloading of SystemC class library elements. The original code of these elements calls the SystemC kernel functions to support process concurrency and communication. The new code (defined in an implementation library) calls the embedded RTOS functions that implement the equivalent functionality. Thus, SystemC kernel functions are replaced by typical RTOS functions in the generated software. The embedded system description is not modified during the software and interface generation process. The proposed approach is independent of the embedded RTOS. This allows the designer to select the commercial or open source OS that best matches the system requirements. In fact, the proposed methodology even supports the use of an application-specific OS.

The contents of the paper are as follows. In this section, the state of the art, motivation and objectives of the work have been presented. In section 2, the system-level specification methodology is briefly explained in order to show the input description style of the proposed method. In section 3, the embedded SW generation and communication channel implementation methodology will be described. In section 4, some experimental results will be provided. Finally, the conclusions will be drawn in section 5.

2. Specification Methodology

Our design methodology follows the ITRS predictions toward the integration of the system-level specification in the design process. SystemC has been chosen as a suitable language supporting the fundamental features required for system-level specification (concurrency, reactiveness,…).

The main elements of the proposed system specification are processes, interfaces, channels, modules and ports. The system is composed of a set of asynchronous, reactive processes that concurrently perform the system functionality. Inside the process code no event object is supported. As a consequence, the *notify* or wait primitives are not allowed except for the "timing" wait, *wait(sc_time)*. No signal can be used and processes lack a sensitivity list. Therefore, a process will only block when it reaches a "timing" wait or a *wait* on event statement inside a communication channel access. All the processes start execution with the sentence *sc_start()* in the function *sc_main()*. A process will terminate execution when it reaches its associated end of function.

The orthogonality between communication and process functionality is a key concept in order to obtain a feasible and efficient implementation. To achieve this, processes communicate among themselves by means of channels. The channels are the implementation of the behavior of communication interfaces (a set of methods that the process can access to communicate with other processes). The behavior determines the synchronization and data transfer procedures when the access method is executed. For the channel description at the specification level it is possible to use *wait* and *notify* primitives. In addition, it is necessary to provide platform implementations of each channel. The platform supplier and occasionally the specifier should provide this code. The greater the number of appropriate implementations for these communication channels on the platform, the greater the number of partition possibilities, thus improving the resulting system implementation.

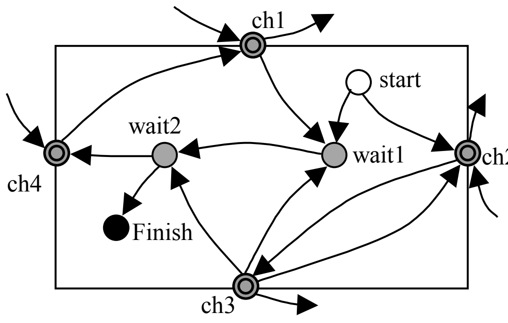


Figure 1. Process representation.

Figure 1 shows a process representation graph composed of four kinds of nodes. The process will resume in a start node and will eventually terminate in a finish node. The third kind of node is an internal node containing timing wait statements. The fourth kind of node is the channel method access node. The segments are simply those code paths where the process executes without blocking.

Hierarchy is supported since processes can be grouped within modules. Following the IP reuse-oriented design recommendations for intermodule communications, port objects are also included. Therefore, communication among processes of different module instances passes through ports. Port and module generation is static. Granularity is established at the module level and a module may contain as many processes as needed. Communication inside the inner modules must also be performed through channels.

The previously commented set of restrictions on how SystemC can be used as a system specification language do not constrain the designer in the specification of the structure and functionality of any complex system. Moreover, as the specification methodology imposes a clear separation between computation and communication, it greatly facilitates the capture of the behavior and structure of any complex system ensuring a reliable and efficient co-design flow.

3. Software Generation

Today, most industrial embedded software is manually generated from the system specification, after SW/HW partition. This software code includes several RTOS function calls in order to support process concurrency and synchronization. If this code is compared with the equivalent SystemC description of the module a very high correlation between them is observed. There is a very close relationship between the RTOS and the SystemC kernel functions that support concurrency. Concerning interface implementation, the relationship is not so direct. SystemC channels normally use *notify* and *wait* constructions to synchronize the data transfer and process execution while the RTOS normally supports several different mechanisms for these tasks (interruption, mutex, flags, …). Thus, every SystemC channel can be implemented with different RTOS functions [18].

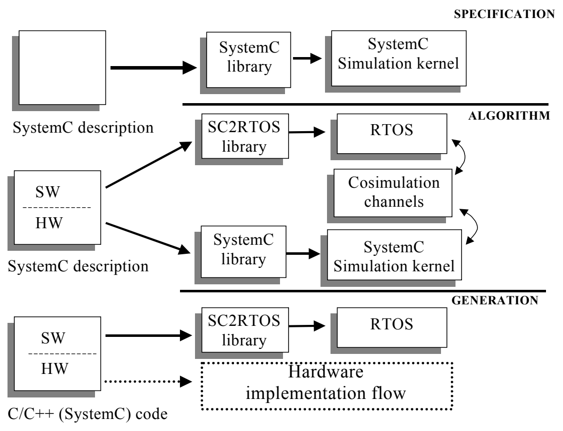


Figure 2. Proposed SW generation flow.

The proposed software generation method is based on that correlation. Thus, the main idea is that the embedded software can be systematically generated by simply replacing some SystemC library elements by behaviourally equivalent procedures based on RTOS functions. It is a responsibility of the platform designer to ensure the required equivalence between the SystemC functions and their implementation. Figure 2 shows the proposed software generation flow. The design process begins from a SystemC specification. This description verifies the specification methodology proposed in the previous section. At this level ("Specification") SW/HW partition has not been performed yet. In order to simulate the "Specification level" description, the code has to include the SystemC class library ("systemc.h" file) that describes the SystemC simulation kernel.

After partition, the system is described in terms of SW and HW algorithms ("Algorithmic level"). At this level, the code that supports some SystemC constructions in the SW part has to be modified. This replacement is performed at library level, also so it is totally hidden from the designer who sees these libraries as black boxes. Thus, the SystemC user code (now pure C/C++ code) is not modified during the software generation flow, constituting one of the most important advantages of this approach. A new library SC2RTOS (SystemC to RTOS) redefines the SystemC constructions whose definition has to be modified in the SW part. It is very important to highlight that the number of SystemC elements that have to be redefined is very small.

Table 1 shows these elements. They are classified in terms of the element functionality. The table also shows the type of RTOS function that replaces the SystemC elements. The specific function depends on the selected RTOS. The library could be made independent of the RTOS by using a generic API (e.g.: EL/IX[19]).

	Hierarchy	Concurrency	Communication	
SystemC Elements	SC_MODULE SC_CTOR sc_module sc_module_name	SC_THREAD SC_HAS_PROCESS sc_start	wait (sc_time) sc_time sc_time_unit	sc_interface sc_channel sc_port
RTOS Functions		Thread management	Synchronization management Timer management	Interruption management Memory access

Table 1. SystemC elements replaced

At algorithmic level, the system can be co-simulated. In order to do that, an RTOS-to-SystemC kernel interface is needed. This interface models the relation between the RTOS and the underlying HW platform (running both over the same host).

Another application of the partitioned description is the objective of this paper: software generation. In this case ("Generation" level in Figure 2), only the code of the SW part has to be included in the generated software. Thus, the code of the SystemC constructions of the HW part has to be redefined in such a way that the compiler easily and efficiently eliminates them.

The analysis of the proposed SW generation flow concludes that the SystemC constructions have to be replaced by different elements depending on the former levels (Specification, Algorithm or Generation) and the partition (SW or HW) in which the SystemC code is implemented. This idea is presented in Figure 3:

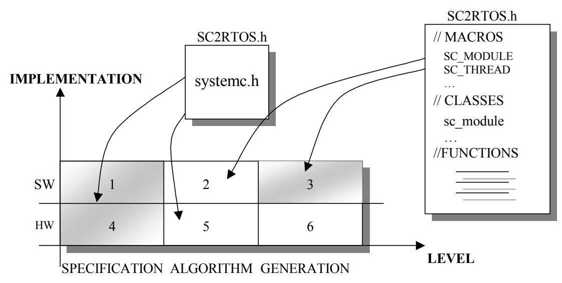
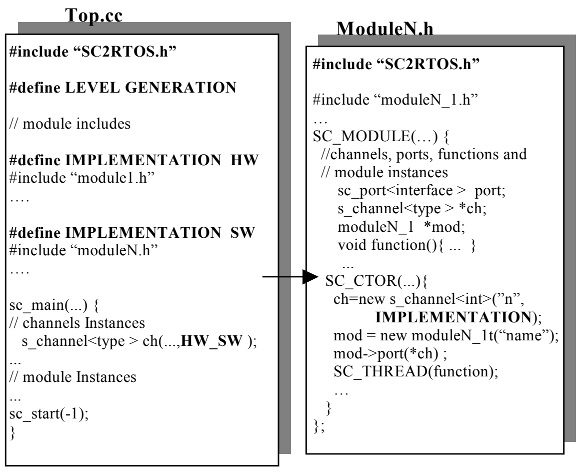


Figure 3. SC2RTOS library implementation

Thus, there are 6 possible implementations of the SystemC constructions. In the proposed approach all these implementations are included in a file (SC2RTOS.h) whose content depends on the values of the LEVEL and IMPLEMENTATION variables. For example, at SPECIFICATION level only the LEVEL variable is taken into account (options 1 and 4 are equal) and the SC2RTOS file only includes the SystemC standard include file (systemc.h). However, in option 3, for example, the SC2RTOS file redefines the SystemC constructions in order to insert the RTOS. Figure 4 presents an example (a SystemC description) that shows these concepts:



Top.cc

```
#include "SC2RTOS.h"

#define LEVEL GENERATION

// module includes

#define IMPLEMENTATION  HW
#include "module1.h"
….

#define IMPLEMENTATION  SW
#include "moduleN.h"
….

sc_main(...) {
// channels Instances
   s_channel<type > ch(...,HW_SW );
...
// module Instances
...
sc_start(-1);
}
```

ModuleN.h

```
#include "SC2RTOS.h"

#include "moduleN_1.h"
…
SC_MODULE(…) {
  //channels, ports, functions and
  // module instances
     sc_port<interface >  port;
     s_channel<type > *ch;
     moduleN_1  *mod;
     void function(){ ...  }
       ...
   SC_CTOR(...){
     ch=new s_channel<int>("n",
              IMPLEMENTATION);
     mod = new moduleN_1t("name");
     mod->port(*ch) ;
     SC_THREAD(function);
     …
   }
};
```

Figure 4. SystemC description example

The "#define" preprocessing directives will be introduced by the partitioning tool. These directives are ignored at specification level. In this paper, it is assumed that the platform has only one processor, the partition is performed at module level in this top hierarchical module and every module is assigned to only one partition. Thus, only the modules instantiated in the top module are assigned to the hardware or software partition. Hierarchical modules are assigned to the same parent-module partition. Only the channels instantiated in the top hierarchical module can communicate processes assigned to different partitions. The channels instantiated inside the hierarchy will communicate processes that are assigned to the same partition. This is a consequence of the assignment of the hierarchical modules.

In Figure 4, the top hierarchical module (described in file *top.cc*) and one of the hierarchical modules (described in file *ModuleN.h*) are presented. All the statements that introduce partition information in these descriptions have been highlighted with bold fonts. The LEVEL and IMPLEMENTATION variables are specified with pre-processor statements. With a little code modification, these variables could be defined, for example, with the compiler command line options. In this case, the partition decisions will only affect the compiler options and the SystemC user code will not be modified during the software generation process.

Before a module is declared, the library "SC2RTOS.h" has to be included. This allows the compiler to use the correct implementation of the SystemC constructions in every module declaration. Concerning the channels, an additional argument has been introduced in the channel declaration. This argument specifies the type of communication (HW_SW, SW_HW, SW or HW) of a particular instance of the channel. This parameter is used to determine the correct channel implementation. The current version of the proposed software generation method is not able to determine automatically the partitions of the processes that a particular instance of a channel communicates, thus this information must be explicitly provided.

4. Application example

In order to evaluate the proposed technique a simple design, a car Anti-lock Braking System (ABS) [17] example, is presented in this section. The system description has about 200 SystemC code lines and it includes 5 modules with 6 concurrent processes.

The system has been implemented in an ARM-based platform that includes an ARM7TDMI processor, 1Mb RAM, 1Mb Flash, two 200Kgate FPGAs, a little configuration CPLD and an AMBA bus. The open source eCos operating system has been selected as embedded RTOS. In order to generate software for this platform-OS pair, a SystemC-to-eCos Library has to be defined. This library is application independent and it will allow the generation of the embedded software for that platform with the eCos RTOS.

The SC2ECOS (SystemC-to-eCos) Library has about 800 C++ code lines and it basically includes the concurrency and communication support classes that replace the SystemC kernel. This library can be easily adapted to a different RTOS.

The main non-visible elements of the concurrency support are the *uc_thread* and *exec_context* classes. The *uc_thread* class maintains the set of elements that an eCos thread needs for its declaration and execution. The exec_context class replaces the SystemC *sc_simcontext* class during software generation. It manages the list of declared processes and its resumption. These elements call only 4 functions of eCos (see Table 2):

	Thread management	Synchronization management	Interruption management
eCos Functions	cyg_thread_create cyg_thread_resume cyg_user_start cyg_thread_delay	cyg_flag_mask_bits cyg_flag_set_bits cyg_flag_wait	cyg_interrupt_create cyg_interrupt_attach cyg_interrupt_acknowledge cyg_interrupt_unmask

Table 2. Ecos functions called by the SC2ECOS Library.

In order to allow communication among processes, several channel models have been defined in the SC2ECOS library. The ABS application example uses two of them; a blocking channel (one element sc_fifo channel) and an extended channel (that enables blocking, non-blocking and polling accesses).

The same channel type could communicate SW processes, HW processes or a HW and a SW process (or vice versa). Thus, the proposed channel models have different implementations depending on the HW/SW partition. In order to implement these channel models only 7 eCos functions have been called (center and right columns of Table 2). These eCos functions control the synchronization and data transfer between processes.

The ABS system has 5 modules: the top ("Abs_system"), 2 modules that compute the speed and the acceleration ("Compute Speed" and "Compute Acceleration"), another ("Take decision") that decides the braking, and the last one ("Multiple Reader") that provides speed data to the "Compute Acceleration" and "Take Decision" modules. Several experiments, with different partitions have been performed.

Several conclusions have been obtained from the analysis of the experimental results shown in Figure 5:

- There is a minimum memory size of 53.2Kb that can be considered constant (independent of the application). This fixed component is divided in two parts: a 31K contribution due to the default configuration of the eCos kernel, that includes the scheduler, interruptions, timer, priorities, monitor and debugging capabilities and a 22.2Kb contribution, necessary to support dynamic memory management, as the C++ library makes use of it.
- There is a variable component of the memory size that can be divided into two different contributions. One of them is due to the channel implementation and asymptotically increases to a limit. This growth is non-linear due to the compiler optimizing the overlap of necessary resources among the new channel implementations. It reaches a limit depending on the number of channel implementations given for the platform. Thus, each platform enables a different limit. The channel implementation of the ABS system requires 11.4Kb. The other component depends on the functionality and module specification code. This growth can be considered linear (at a rate of approximately 1Kb per 50-70 source code lines in the example), as can be deduced from Table 3.

The total software code generated is limited to 68.7Kb:

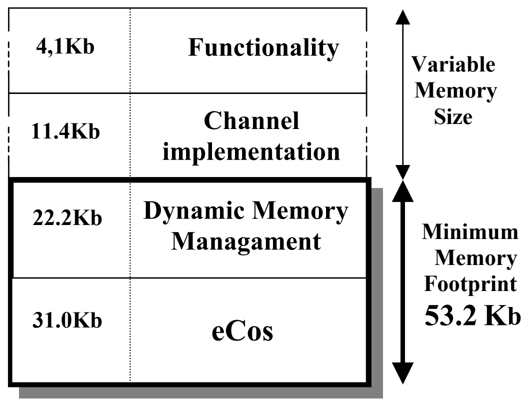


Figure 5. Memory footprint for the ABS SW implementation.

Table 3 shows in more detail how the functionality size component in the ABS system is distributed in each module:

SC_MODULE	Abs System	Compute Speed	Compute acceleration	Take Decision	Multiple Readers
C++ Code lines	30	44	46	110	12
Blocking channels	3	0	0	1	0
Extended channels	1	0	0	0	0
Generated SW	1130 bytes	700 bytes	704 bytes	3510 bytes	140 bytes

Table 3. Code size introduced by each module.

The addition of module memory sizes gives the value of 6.2Kb. This exceeds by 2.1Kb the 4.2Kb shown in

Figure 5. This is due to the shared code of instanced channels that appears in modules including structural description (Take Decision and ABS System).

The overall overhead introduced by the method respect to manual development is negligible because the SystemC code is not included but substituted by a C++ equivalent implementation that uses eCos. In terms of memory, only the 22.2Kbytes could be reduced if an alternative C code avoiding dynamic memory management is written.

5. Conclusions

This paper presents an efficient embedded software generation method based on SystemC. This technique reduces the embedded system design cost in a platform based HW/SW codesign methodology. One of its main advantages is that the same SystemC code is used for the system-level specification and, after SW/HW partition, for the embedded SW generation. The proposed methodology is based on the redefinition and overloading of SystemC class library construction elements. In the software generated, those elements are replaced by typical RTOS functions. Another advantage is that this method is independent of the selected RTOS and any of them can be supported by simply writing the corresponding library for that replacement. Experimental results demonstrate that the minimum memory footprint is 53.2 Kb when the eCos RTOS is used. This overhead is relatively low taking into account the great advantages that it offers: it enables the support of SystemC as unaltered input for the methodology processes and gives reliable support of a robust and application independent RTOS, namely eCos, that includes an extensive support for debugging, monitoring, etc… To this non-recurrent SW code, the minimum footprint has to be increased with a variable but limited component due to the channels. Beyond this, there is a linear size increment with functionality and hierarchy complexity of the specification code.

References

[1] A. Allan, D. Edenfeld, W. Joyner, A. Kahng, M. Rodgers, Y. Zorian: “2001 Technology Roadmap for Semiconductors”. IEEE Computer. January 2002.

[2] International Technology Roadmap for Semiconductors. 2001 Update. Design. Editions at http://public.itrs.net.

[3] J. Borel: "SoC design challenges: The EDA Medea Roadmap", in E. Villar (Ed.): "Design of HW/SW embedded systems". University of Cantabria. 2001.

[4] L. Geppert: "Electronic design automation", IEEE Spectrum, V.37, N.1, January 2000.

[5] G. Prophet: “System Level design Languages: to C or not to C?”. EDN Europe. October 1999. www.ednmag.com.

[6] "SystemC 2.0 Functional specification", www.systemc.org, 2001.

[7] P. Sánchez: "Embedded SW and RTOS", in E. Villar (Ed.): "Design of HW/SW embedded systems". University of Cantabria. 2001.

[8] PtolemyII. http://ptolemy.eecs.berkeley.edu/ptolemyII.

[9] F.Baladin, M. Chiodo, P.Giusto, H.Hsieh, A. Jurecska, L.Lavagno, C.Passerone, A.Sangiovanni-Vicentelli, E. Sentovich, K. Suzuki, B.Tabbara “Hardware-Software Co-design of Embedded Systems: The POLIS Approach”. Kluwer. 1997.

[10] R.K.Gupta. “Co-synthesis of Hardware and Software for Digital Embedded Systems”. Ed. Kluwer. August 1995. ISBN 0-7923-9613-8.

[11] D. Desmet, D. Verkest and H. de Man, “Operating System Based Software Generation for System-On-Chip”, Proc.Design Automation conf., June 2000.

[12] D. Verkest, J. da Silva, C. Ykman, K.C roes, M. Miranda, S. Wuytack, G. de Jong, F. Catthoor and H. de Man. “Matisse: A system-on-chip design methodology emphasizing dynamic memory management”. Journal of VLSI signal Processing, 21(3): 277-291, July 1999.

[13] M. Diaz-Nava, W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya. “Component-Based Design Approach for Multicore SoCs”, Proc. Design Automation Conf. June 2002.

[14] L. Gauthier, S. Yoo and A.A. Jerraya “Automatic Generation of Application-Specific Operating Systems and Embedded Systems Software”, Proc. of Design Automation and Test in Europe. Mar. 2001.

[15] S. Yoo; G. Nicolescu; L. Gauthier & A.A. Jerraya: "Automatic generation of fast simulation models for operating systems in SoC design", Proc. of DATE'02, IEEE Computer Society Press, 2002.

[16] K. Weiss; T. Steckstor and W. Rosenstiel. “Emulation of a Fast Reactive Embedded System using a Real Time Operating System”. Proc of DATE’99. March 1999.

[17] Coware, Inc., ”N2C”, http://www.coware.com/N2C.html.

[18] F. Herrera, P. Sánchez & E. Villar: “HW/SW Interface Implementation from SystemC for Platform Based Design”. FDL’02. Marseille. September 2002.

[19] EL/IX Base API Specification DRAFT. http://sources.redhat.com/elix/api/current/api.html.