SOFTWARE AND DRIVER SYNTHESIS FROM TRANSACTION LEVEL MODELS

Haobo Yu, Rainer Dömer, Daniel D. Gajski
Center of Embedded Computer Systems
University of California, Irvine

Abstract This work presents a method of automatically generating embedded software including bus driver code from a transaction level model (TLM). For the application software, a real time operating system (RTOS) adapter is introduced to model scheduling and synchronization at C level. ANSI-C code is generated targeting this RTOS adapter. Bus drivers are also automatically created for HW/SW communication. Finally, the software image file is created from the C code, bus driver code, RTOS adapter and RTOS library code.

As a result, efficient embedded software is synthesized from abstract, target CPU independent source code, eliminating the need for manual RTOS targeting, I/O driver coding and system integration.

1. Introduction

The rapid development of semiconductor process technology and the increasing use of RISC/DSP cores contribute to an increased importance of embedded software in SoC. A typical SoC design today includes one or more processors, memory, dedicated hardware, and a complex communication architecture. To drive the hardware, target specific SoC software is needed which contains real time operating systems and bus drivers, along with the specific application software. The increasing complexity of software in such SoC designs requires that a large period of the design time will actually be used for software development.

Transaction level models (TLM) are widely used in SoC modeling for early design space exploration. After the SoC architecture is fixed, separate HW/SW models are created from the TLM. Usually, the TLM is written in a system level description language (SLDL) (e.g. SystemC [9] or SpecC [6]). Today however, the TLM is mostly used only as a reference model for software engineers. Most of the embedded software is still written manually from scratch. This is a slow and error-prone

process. Moreover, to validate the manually written software, designers have to simulate the compiled binary through either a hardware prototype or slow instruction set simulators (ISS). Both approaches hinder the SoC development process significantly since the prototype is usually only available in the later stages of the design process and the ISS simulation is extremely slow.

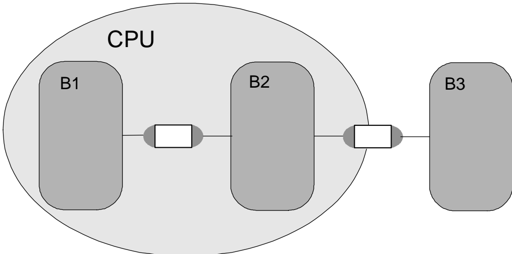
To tackle this problem, it is desirable that we can derive SoC software directly from the TLM and the generated software can be validated and tested before any platform prototype is available. This paper solves these problems by introducing software synthesis which automatically generates application as well as bus driver code from the TLM. Rather than binary code simulation, the generated software can be directly re-imported into the original TLM and simulated with the rest of the system at C level. As a result, simulation speed increases by orders of magnitude, while cycle and pin accurate I/O is still available at the bus level.

The rest of this paper is organized as follows: After a brief overview of related work, Section 2 describes the overall design flow of our software synthesis process. Section 3 through Section 6 then address RTOS targeting, application code generation, bus driver synthesis, and binary image generation in detail. Finally, experimental results are listed in Section 7, and Section 8 concludes this work.

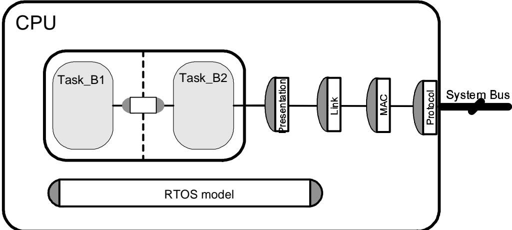
1.1 Related Work

A lot of work has been spent on software synthesis. There are approaches to code generation from UML [1], from graphical finite state machine design environments (e.g. StateCharts [10]), from DSP graphical programming environments (e.g. Ptolemy [13]), or from synchronous programming languages (e.g. Esterel [2]). In POLIS [5], a way of generating C code from co-design finite state machines is described. However, this work targets mainly reactive real-time systems and cannot be easily applied to more general applications. There are also works on software scheduling, including quasi-static scheduling in Petri-Nets [12], and a combination of static and dynamic task scheduling [3]. Operating system based software synthesis can be found in [4] and [7].

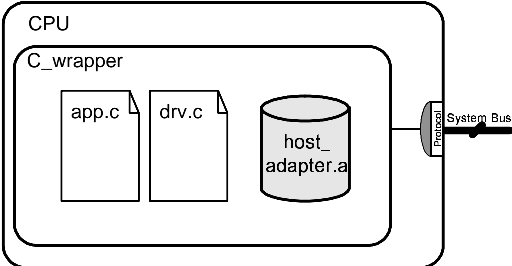
The focus in this paper is similar to the approaches [11] and [9]. [11] presents software generation from SystemC based on the redefinition and overloading of SystemC class library elements. In [9], a software-software communication synthesis approach by substituting SystemC modules with an equivalent C structure is proposed. However, the code generated by these two approaches can not be validated through insertion into



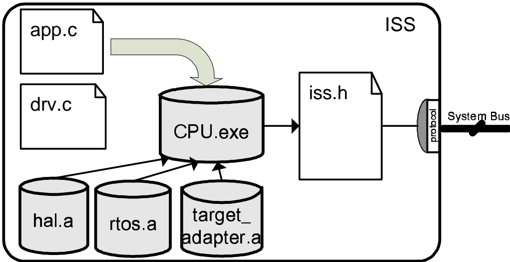
(a) Specification model



(b) TLM



(c) C model



(d) Implementation model

Figure 1. Software synthesis flow.

the original system model and the I/O drivers of the system are not addressed.

2. Design Flow

The system design process starts with a specification model written by the designer to specify the desired system functionality, as shown in Fig. 1(a). Then, a system architecture model is derived through system partitioning and system bus selection. During this process, the system functionality is partitioned onto multiple processing elements (PEs) and a communication architecture consisting of abstract communication channels is inserted to represent the refined communication between PEs [14]. The resulting model, shown in Fig. 1(b), serves as the input for the software code generation.

During system partitioning, RTOS scheduling is inserted for the processor PEs that require dynamic scheduling support [8]. Software tasks are created from behaviors mapped to the same PE and an abstract RTOS model is inserted to manage the generated software tasks so that the software can be simulated at the fast transaction level.

Next, from the TLM, software C code is generated, as shown in Fig. 1(c). Application code (*app.c*) is generated for the software tasks and bus drivers (*drv.c*) are created for the abstract communication adapter channels. For validation, the generated C code is re-imported into the system model through a wrapper. Designers can use this C model for fast co-simulation of the system, avoiding the time consuming instruction set simulator (ISS) co-simulation for many cases.

Finally the generated C code is compiled into the processors instruction set and linked against the target RTOS to produce the final binary image. For final timing analysis, the binary code can also be simulated by use of an ISS, as shown in Fig. 1(d).

3. RTOS Adapter

To support multitasking, a RTOS kernel is usually needed for the generated C code. However, there exists a large variety of RTOS providing different interfaces. One solution is to create specific software code for each target RTOS. An alternative solution is to use a general interface which abstracts away the underlining target specific RTOS implementations. In other words, a middleware layer can be used to adapt the specific RTOS to a general API. Our approach follows the second solution, using a RTOS adapter which provides a common interface to specific RTOS services. The RTOS adapter is essentially a middleware layer between the specific RTOS and the generated application software.

```
void   OSStart(void);
void   OSInit(void);
void   OSWaitfor(sim_time);
/*Task Management*/
task_t TaskCreate(task_f *f,task_a arg,int pri);
void   TaskDelete(task_t t);
void   TaskJoin(task_t t);
void   TaskRun(task_t t1, task_t t2);
void   TaskSuspend(task_t t);
void   TaskResume(task_t t);
/*Inter Task Synchronization*/
void   SemRelease(sem_t  *sem);
void   SemAquire(sem_t  *sem);
void   EventNotify(evt_t e);
void   EventWait(evt_t e);
/*Channels*/
void   MutexAquire(mtx_t m);
void   MutexRelease(mtx_t m);
void   QueueSend(const void *d, unsigned long l);
void   QueueReceive(void *d, unsigned long l);
...
```

Figure 2. RTOS adapter interface

3.1 Adapter Procedural Interface

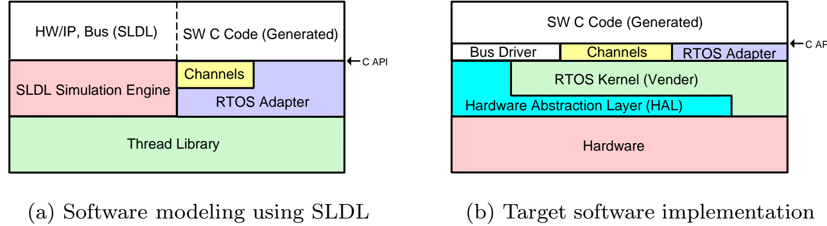
The interface of our RTOS adapter is defined in Fig. 2. *OSInit* initializes the relevant kernel data structures while *OSStart* starts the task scheduling. In addition, *OSWaitfor* is provided to enable time modeling in the simulation. That is, the software tasks can call *OSWaitfor* to advance the system simulation time. This is used only for simulation and it will be ignored later in the real code.

Task management is the most important part of the RTOS adapter. This includes standard functions for task creation (*TaskCreate*), task completion (*TaskJoin*), task termination (*TaskDelete*), and temporary task suspension (*TaskSuspend*, *TaskResume*).

The RTOS adapter also provides two kinds of task synchronization services: semaphore and event. Inter-task communication is provided by abstract channels. Together, these functions support resource sharing, connection oriented data exchange, and any combination of these services. Note that the RTOS adapter provides a similar interface as the standard SLDL channel library. Thus, during code generation, most standard SLDL synchronization methods can be directly converted to this interface.

3.2 Host Adapter Library

In our approach, two RTOS adapter libraries are created for the interface defined above, one for the host and one for the target platform. The host adapter library is linked against the SLDL simulation engine



(a) Software modeling using SLDL (b) Target software implementation

Figure 3. Software implementation layers

so that the C code can co-simulate with the rest of the system code in SLDL. As shown in Fig. 3(a), the lowest layer of the host software stack is the thread library. This can be any host library supporting thread management (e.g POSIX thread library, Win32 thread library, or QuickThreads). On top of the thread management layer, both the SLDL simulation engine and the host RTOS library are implemented. The SLDL simulation engine provides standard system level constructs (channels, behaviors, interfaces etc.) for modeling custom hardware, IP cores and system busses, while the host RTOS adapter provides the C API for the OS related functions.

For host simulation, the highest layer is a TLM for the application including the hardware, software and the communication channels. The software (*app.c*, *drv.c*) for each processor is encapsulated in a SLDL wrapper. The RTOS adapter provides the generated C code with standard OS services. Communication between software C code and the rest of the system is performed through the ports of the wrapper.

3.3 Target Adapter Library

Fig. 3(b) shows the target software implementation. This is based upon the processor's hardware including its instruction set architecture (ISA), its I/O interfaces and its interrupt handling mechanisms. The hardware abstraction layer (HAL) is then implemented in software on top of this layer. It provides an abstraction of the processor hardware and is used as an interface for higher software layers. The next layer is the RTOS kernel for the target processor, which can be a commercial or a custom RTOS. The RTOS kernel implements basic multitasking and synchronization functionality corresponding to the services provided by the RTOS adapter. On top of the RTOS kernel is the target RTOS adapter which resembles a middleware layer translating task management APIs of the RTOS adapter into the corresponding APIs of the target RTOS kernel. Furthermore, the inter-task communication functionality of the RTOS adapter is directly implemented by using inter-process communication (IPC) mechanisms that are part of the target RTOS kernel. Note

that bus drivers are implemented on top of both HAL library and RTOS kernel.

4. Application Code Generation

In the code generation process, C code is generated for all the parts of the system mapped to a processor. Specifically, SLDL constructs, including behaviors, channels and interfaces, are converted into C code. Essentially, this step synthesizes the hierarchy and port mapping elements contained in the SLDL description into ANSI C code.

For details on this code generation process, please refer to [15].

4.1 OS Targeting for Task Management

Generally, computation within the TLM is described through hierarchical and parallel composition of behaviors. This can be implemented in software using hierarchical C structures. For concurrent behaviors, however, multiple software tasks are needed to implement the specification.

This process is illustrated in Fig. 4. In the specification model (Fig. 4(a)), two behaviors $B1$ and $B2$ are running in parallel inside behavior CPU. After the RTOS scheduling step, two software tasks ($Task_B1$ and $Task_B2$) are created dynamically and scheduled by the abstract RTOS model, as shown in Fig. 4(b) [16]. Next, C code is generated for these tasks, as shown in Fig. 4(c) [15].

```
behavior B1()
{void main(void)
  ...}
behavior B2()
{void main(void)
  ...}
behavior CPU()
{B1 b1();
 B2 b2();
void main(void)
{




   par
    { b1.main();
      b2.main();
    }

}
```

(a) Spec. model

```
behavior Task_B1()
{void main(void)
  ...}
behavior Task_B2()
{void main(void)
  ...}
behavior Task_CPU(RTOS os)
{Task_B1 task_b1(os);
 Task_B2 task_b2(os);
 void main(void) {
 Task me;
 task_b1.os_task_create();
 task_b2.os_task_create();
 me = os.fork();
 par {
   b1.main();
   b2.main();
 }
 os.join(me);}
}
```

(b) Multi task TLM

```
struct Task_B1 {...};
void  Task_B1_main(struct
        Task_B1 *This)
{...}
struct Task_B2 {...};
void  Task_B2_main(struct
        Task_B2 *This)
{...}
struct Task_CPU         {
 struct Task_B1 task_b1;
 struct Task_B2 task_b2 }
void Task_CPU_main(struct
        Task_B1B2 *This)
{
TaskCreate(&Task_B1_main,
       &This->task_b1, 1);
TaskCreate(&Task_B2_main,
       &This->task_b2, 2);

TaskJoin(NULL);
}
```

(c) C code

Figure 4. C code generation for task management

4.2 OS Targeting for Task Communication

Channels in the TLM can be divided into two categories: intra- and inter-PE channels. For the software implementation, the former are converted into task communication, while the latter are implemented as bus drivers.

The intra-PE channels can be further divided into two categories, SLDL standard channels and user defined channels. During the code generation process, methods of SLDL standard channels can be directly converted into the corresponding channel APIs of the RTOS adapter. User defined channels, on the other hand, are implemented the same way as the behaviors.

5. Bus Driver Generation

In the partitioned system specification, communication between different PEs is performed through message passing channels with different semantics (blocked vs. non-blocked) and different data types. Then, the bus driver synthesis step refines the system communication architecture from an abstract message-passing down to an actual implementation over pins and wires.

Channel refinement is performed before the bus driver code can be created. This includes the definition of the overall network topology and generation of point-to-point communication links. The point-to-point links are then grouped into physical links and packet transfers for each link are implemented. Note that four layers of communication channels are inserted to drive the low layer communication media interfaces of each PE, as illustrated in Fig. 1(b) earlier.

Table 1 summarizes the communication channels, which refine the message passing channel to protocol word/frame transactions. The highest layer is the presentation channel which provides services to send and receive messages of arbitrary, abstract data type between different PEs. The next layer is the link channel which provides services to exchange data packets in the form of uninterpreted byte blocks. Typically, in a bus-based master/slave arrangement, each logical link is split into a data stream under the control of the master and a handshake (interrupt) from slave to master. So, in the implementation, the master side waits for a semaphore (which will be released by a client interrupt) before initiating a write or read transfer.

The media access (MAC) channel implements external interfaces of the HAL library for a processor PE. It is responsible for slicing blocks of bytes into unit transfers available at the bus interface. Finally, the

Name	C code

Presentation
• Typed, named messages
• Data formatting
	App_send(struct App *This, struct S *buf)
	{
	Link_send(This->link,(void *)&buf, sizeof(buf));
	}

Link
• Point-to-Point logical links
• Synchronization
• Addressing
	Link_send(struct Link *This,void *d, unsigned l)
	{
	SemAquire(This->sem);
	MAC_write(This->mac,This->addr,d,l);
	}

Media Access
• Shared medium streams
• Data slicing
	MAC_write(struct Mac *This,unsigned addr,
	void *d,unsigned l){
	for(...)
	word = ...;
	Protocol_writeWord(addr,word);
	}

Protocol
• Word/frame transmission
• Protocol timing
	Protocol_writeWord(U32 addr, WORD data)
	{
	*addr = data; /*memory mapped IO*/
	}

Table 1. C code generation for communication adapter channels.

protocol channel provides services to transfer words or frames over the physical medium.

In our implementation, the MAC and protocol channels are taken out of the processor database, while the presentation and link channels are created automatically. As we can see from Table 1, the bus driver (*drv.c*) is created by converting the four communication adapter channels into C and assembly code. This is then used by the application to drive the protocol channel. Note that the protocol channel implementation varies depending on how the processor is connected to the system bus. Usually, in a typical memory mapped I/O arrangement, the protocol layer send/receive primitives correspond to load and store instructions in the processor.

6. Target Specific Binary Creation

As the final step of software synthesis, the output C code is then compiled into the target processor's instruction set using the C compiler available for the processor. During this process, a HAL library is needed to provide target specific initialization and run time environment routines. It also provides the implementation for the MAC and protocol channels used during the bus driver synthesis.

Finally, the compiled object code is linked against the RTOS kernel, the target RTOS adapter library and the processor HAL library to generate executable code for the processor. A final simulation model can be created by replacing the component model of the processor with the ISS wrapper behavior of the target processor.

7. Experimental Results

We have implemented the proposed software synthesis tool and applied it to a set of design examples: a GSM voice codec, a JPEG encoder, a motor control system, and a MP3 decoder. For each example application, we have created a set of architectures varying in the number of hardware units that accellerate some part of the computation.

Using our software synthesis tool, we were able to generate the entire embedded software for each target architecture automatically. Moreover, code generation took less than a second in every case.

Design (loc), architecture		CPU, num. of co-proc.	Scheduling policy	Bhvrs./ Chnls.	SW Tasks	C code (loc)	Time (sec)
Vocoder 9,191	arch1	DSP56600, 1	RR	109/3	2	8,297	0.34
	arch2	DSP56600, 2	RR	104/4	2	8,098	0.35
	arch3	Coldfire, 3	priority	109/5	2	8,334	0.44
	arch4	Coldfire, 4	priority	111/6	2	8,537	0.50
JPEG 2,251	arch1	DSP56600, 1	static	26/3	1	1,119	0.11
	arch2	DSP56600, 2	static	37/4	1	1,553	0.10
	arch3	Coldfire, 3	static	39/5	1	1,636	0.09
	arch4	Coldfire, 4	static	39/6	1	1,679	0.11
Motor 2,049	arch1	TX-49, 1	RR	28/9	34	1,931	0.07
	arch2	TX-49, 2	RR	27/10	6	1,916	0.06
	arch3	TX-49, 3	priority	25/8	4	1,720	0.05
	arch4	TX-49, 4	priority	25/9	4	1,745	0.08
MP3 8,592	arch1	Coldfire, 1	RR	148/6	7	27,191	0.76
	arch2	Coldfire, 5	RR	147/7	16	25,524	0.85

Table 2. Software synthesis results.

Details of our experimental results are summarized in Table 2. We have targeted three different CPUs, the Motorola DSP 56600, the Motorola Coldfire processor, and the Toshiba TX-49 processor. Each CPU is assisted by a number of hardware accelleration units, as listed in the table. Also listed are the scheduling policy (round-robin, priority-based, or static), the number of behaviors and channels in the SLDL model, the number of parallel software tasks, the number of lines of code (loc) of generated C code, and the run-time of our software synthesis tool (on a 2.4 GHz AMD Opteron PC).

As discussed in Section 6, MAC and protocol layer channels are inserted from the HAL library during the bus driver synthesis. Based on these channels, our synthesis tool creates the bus drivers for the different target processors. Then, to create the final executable image, the C code is compiled into binary code for the target RTOS. For our experiments, we have used the μC/OS-II RTOS which requires only a few lines of interface code for each function in the adapter API.

It could be argued that there is little or no productivity gain if our software synthesis flow is applied to just a single target architecture, because the amount of work in writing the specification TLM is about the same when writing the target C code directly. However, this argument does not hold if the target architecture changes or multiple architectures are analyzed during system design exploration. Then, the specification model is written only once, but many target architecture models can be generated automatically within seconds. Thus, the productivity gain is tremendous and true design space exploration becomes possible.

8. Conclusions

In this work, we have proposed steps to synthesize embedded software code and bus drivers from a TLM. A RTOS adapter library is introduced to facilitate the OS targeting process as well as to enable the generated C code to co-simulate with the rest of the system model. C code is automatically synthesized from the SLDL description of the input TLM. Parallel behaviors are converted into concurrent software tasks. Intra-PE channels are converted into inter-process synchronization and communication primitives, whereas inter-PE channels are converted into software bus drivers.

The automation of the SoC software generation process frees the designer from the tedious and error-prone tasks of creating software manually after SW/HW partitioning. Since the final software is directly derived from the TLM, validation of the software code becomes significantly easier than for manually written code.

In summary, we have developed a software synthesis tool that supports the automatic generation of efficient embedded software from an abstract TLM. Our experiments clearly demonstrate the applicability and benefits of the software synthesis approach in a system design environment.

Currently, our synthesis tool is written for the SpecC SLDL because of its simplicity and easy availability. Future work includes the extension of this methodology for SystemC SLDL, as well as the optimization of the generated code and support for more target RTOS and processors.

References

[1] Rational. http://www.rational.com/uml/index.html.

[2] F. Boussinot and R. de Simone. The ESTEREL Language. In *Proceedings of the IEEE*, September 1991.

[3] J. Cortadella. Task Generation and Compile Time Scheduling for Mixed Data-Control Embedded Software. In *Proceedings of the Design Automation Conference*, pages 489–494, June 2000.

[4] D. Desmet, D. Verkest, and H. Man. Operating System Based Software Generation for System-on-Chip. In *Proceedings of the Design Automation Conference*, pages 396–401, June 2000.

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

[6] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, March 2000.

[7] L. Gauthier, S. Yoo, and A. Jerraya. Automatic Generation and Targeting of Application-Specific Operating Systems and Embedded Systems Software. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Nov 2001.

[8] A. Gerstlauer, H. Yu, and D. Gajski. RTOS Modeling in System Level Design. *Proceedings of Design Automation and Test in Europe (DATE)*, 2002.

[9] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[10] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trachtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, April 1990.

[11] F. Herrera, H. Posadas, P. Sanchez, and E. Villar. Systematic Embedded Software Generation from SystemC. *Proceedings of Design Automation and Test in Europe (DATE)*, 2003.

[12] B. Lin. Software Synthesis of Process-Based Concurrent Programs. In *Proceedings of the Design Automation Conference*, 1998.

[13] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software Synthesis for DSP using Ptolemy. *Journal of VLSI Signal Processing*, 1995.

[14] D. Shin, S. Abdi, and D. Gajski. Automatic Generation of Bus Functional Models from Transaction Level Models. In *Proceedings of the Asia and South Pacific Design Automation Conference*, 2004.

[15] H. Yu, R. Dömer, and D. D. Gajski. Embedded Software Generation from System Level Design Languages. In *Proceedings of the Asia and South Pacific Design Automation Conference*, Jan 2004.

[16] H. Yu, A. Gerstlauer, and D. D. Gajski. RTOS Scheduling in Transaction Level Models. In *Proceedings of the International Symposium on System Synthesis*, 2003.