Embedded Software Generation from System Level Design Languages

Haobo Yu, Rainer Dömer, Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine, USA
{haoboy,doemer,gajski}@cecs.uci.edu

Abstract— To meet the challenge of increasing design complexity, designers are turning to system level design languages (SLDLs) to model systems at a higher level of abstraction. This paper presents a method of automatically generating embedded software from system specification written in SLDL. Several refinement steps and intermediate models are introduced in our software generation flow. We demonstrate the effectiveness of the proposed method by a tool which can generate efficient ANSI C code from system models written in SLDL.

I. Introduction

In order to handle the ever increasing complexity and time-to-market pressures in the design of embedded systems, raising the level of abstraction to the system level is generally seen as one solution to increase productivity. Many system level design languages (SLDLs) and methodologies [3, 4] have been proposed in the past to address the issues involved in system level design. The typical system level design process usually starts from an abstract system specification model, partitions the specification to HW/SW components and ends with the detailed implementation model [10]. Much work has been done in synthesizing the HW part of the system. However, most industrial embedded software is still created manually from the system specification [16]. It is desired that the embedded software can be generated from the system specification automatically.

In the system specification, the functionality of an embedded system is described as a hierarchical network of modules (or processes) interconnected by hierarchical channels. Syntactically, these can be described in SLDLs as a set of behavior, channel and interface declarations. During system synthesis, the specification functionality is partitioned onto multiple processing elements (PEs), such as DSP, custom hardware. Those behaviors mapped onto general or application specific microprocessors will later be implemented as embedded software.

Deriving embedded software from system specification described in SLDL means implementing all SLDL language elements (e.g. modules, processes, channels and port mappings). Since the predominant SLDLs are C/C++ extensions [10, 14], directly compiling SLDL to produce the binary code for the target microprocessors is possible but highly inefficient. The main reason is that the large simulation kernel for the SLDL is included in the compiled code. Besides, some cross compilers for embedded processors may only support C. While SLDL is used mainly for modeling and simulation of designs at system level, much overhead is introduced to support the system level features (e.g *hierarchy*, *concurrency*, *communication*). However, these features are not necessarily needed for the target software code. Considering the limited memory space and execution power of embedded processors, we need to generate compact and efficient software code for implementation. In this paper, we address this problem by proposing a method consisting of several software refinement steps and intermediate models to generate efficient ANSI C code from the system specification written in SLDL.

The rest of this paper is organized as follows: Section II gives an insight into the related work. Section III describes the flow for software generation. Details of the software generation process are covered in Section IV, Section V and Section VI. Experimental results are shown in Section VII and Section VIII concludes this paper with a brief summary .

II. Related Work

Various design methodologies exit for designing embedded software. There are approaches to code generation from an abstract model (UML[2]), from graphical finite state machine design environments (e.g StateCharts [15], from DSP graphical programming environments(e.g. Ptolemy [18]), or from synchronous programming languages (e.g Esterel [7]).

In [17], a software synthesis approach using quasi-static scheduling in Petri-Nets to produce a set of corresponding state machines is presented. In [8], a way of combining static task scheduling and dynamic scheduling in software synthesis is proposed. In [11], a method for automatic generation of application-specific operating systems and corresponding application software is given. While these approaches mainly focus on software synthesis issues, no efficient code generation method is described.

In POLIS [6] system, code generation is performed by first transforming the *Co-design Finite State Machine* (CFSM) specification into a *s-Graph* and then translating the *s-Graph* into portable C code. However, POLIS is mainly for reactive real time systems and can't be easily extended to other more general frameworks.

The work in [9, 14, 16] are close to the topic of this paper. In [9], software generation from a high-level model of operating system called SoCOS is proposed. Compared with [9], our approach uses SLDL based RTOS model and enables automatic refinement while [9] requires its own proprietary simulation engine and needs manual refinement to get the soft-

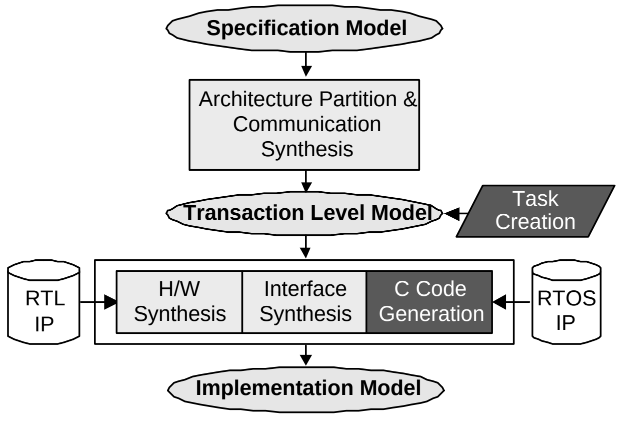


Fig. 1. System design flow

ware code. In [16], software generation from SystemC SLDL based on the redefinition and overloading of SystemC class library elements is presented. However, this approach requires the use of C++ cross-compilers and introduces SLDL language elements overhead. In [14], a software-software communication synthesis approach by substituting each SystemC module with an equivalent C struct is proposed. Our method differs from [14] in that [14] requires special SystemC modeling styles (i.e. using macro definitions and preprocessing switches in addition to the original specification code) while ours don't have any restrictions on how the system model is described.

III. Design Flow

System level design is a process with multiple stages where the system specification is gradually refined from an abstract idea down to an actual implementation. Figure 1 shows a typical system level design flow [10]. The system design process starts with the specification model. During system synthesis, the specification functionality is then partitioned onto multiple processing elements (PEs) and the communication synthesis generates the transaction level model in which a communication architecture consisting of busses and bus interfaces is synthesized to implement communication between PEs.

Our proposed software generation flow is shown in Figure 2, where we generate a set of software tasks scheduled by a preemptive, priority-driven real time kernel (usually a RTOS) from the partitioned design specification written in SLDL. It is carried out in three separate steps. Task creation is the first step, where the modules/processes are converted into software tasks with assigned priorities. Synchronization as part of communication between processes is refined into OS-based task synchronization. In order to evaluate the output multi-task system model (e.g. in terms of the scheduling algorithm) at this moment (i.e. before the actual binary implementation), we use a high level model of the underlying RTOS [12]. The RTOS model provides an abstraction of the key features that define a dynamic scheduling behavior independent of any specific RTOS implementation. The output model generated from the task creation step consists of multiple PEs communicating via a set of busses. Each PE runs multiple tasks on top of its local RTOS model instance. Therefore, the output multi-task model

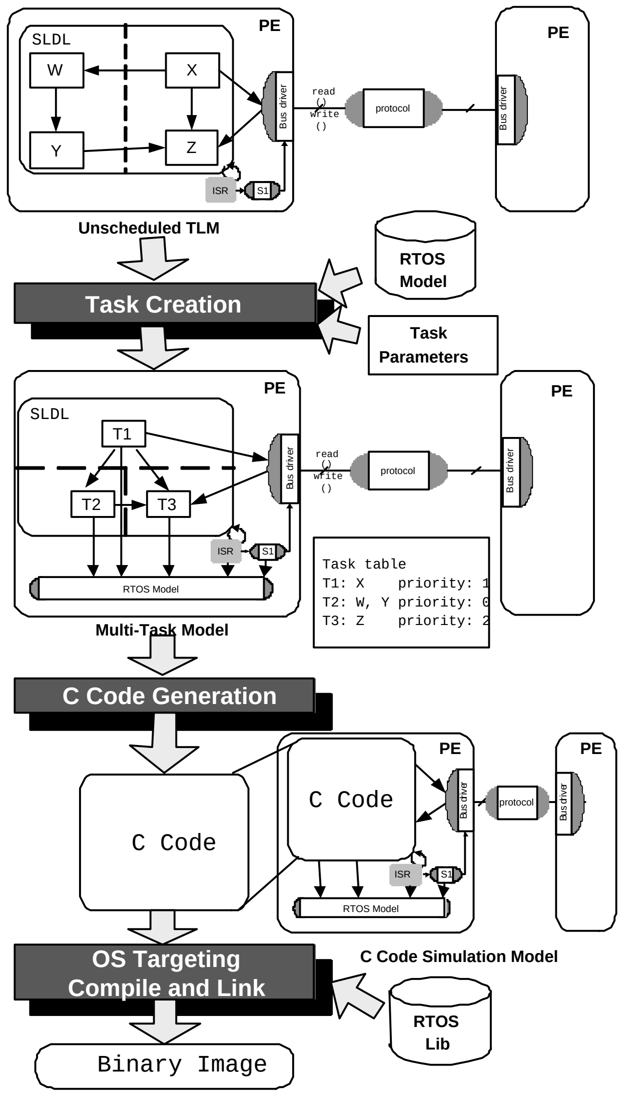


Fig. 2. Software generation flow

can be validated through simulation or verification to evaluate different dynamic scheduling approaches (i.e. round-robin, priority-based) as part of system design space exploration.

In the next system design step, each PE in the transaction level model is then implemented separately. ANSI C code is generated for tasks created in the previous step from their SLDL specification. By importing the C code into the system model and replacing the SLDL task specification, we get the C code simulation model. The designer can simulate and validate the generated C code using the C code simulation model. Note that the RTOS model is still used as an environment to provide the task-scheduling and inter-task communication support for the generated C code.

As the last step, services of the RTOS model are mapped onto the APIs of a commercial or custom RTOS and the C code is compiled into the processor's instruction set. The final binary executable for the chosen processor is generated by linking the compiled code against the RTOS libraries.

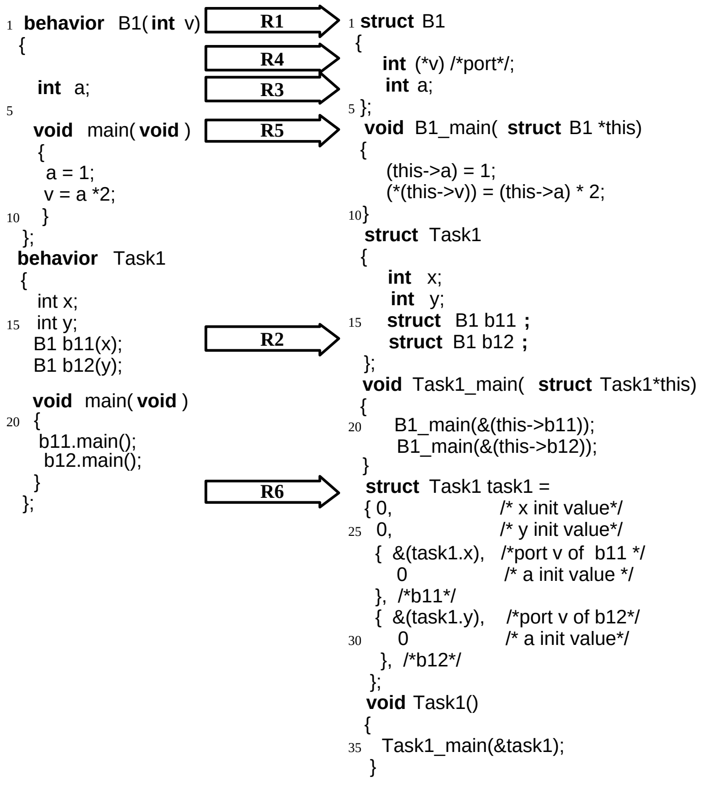
```
1  behavior B2B3()
2  {B2 b2();
3   B3 b3();
4  void main(void)
5  {
6
7
8
9    par{ b2.main();
10        b3.main();}
11
12 }
```

(a) before

```
1  behavior B2B3(RTOS os)
2  {Task_B2 task_b2(os);
3   Task_B3 task_b3(os);
4   void main(void)
5  {Task t;
6   task_b2.os_task_create();
7   task_b3.os_task_create();
8   t = os.fork();
9   par{task_b2.main();
10      task_b3.main();}
11  os.join(t);
12 }
```

(b) after

Fig. 3. Task creation



```
behavior  B1(int  v)
 {

   int  a;

   void  main( void )
   {
    a = 1;
    v = a *2;
   }
 };
behavior  Task1
 {
   int x;
   int y;
  B1 b11(x);
  B1 b12(y);

  void  main( void )
  {
   b11.main();
    b12.main();
  }
 };
```

(a) SpecC Code

```
struct B1
 {
    int (*v) /*port*/;
    int a;
};
 void  B1_main(  struct B1 *this)
 {
    (this->a) = 1;
    (*(this->v)) = (this->a) * 2;
}
 struct  Task1
 {
    int  x;
     int  y;
    struct  B1 b11 ;
     struct  B1 b12 ;
 };
 void  Task1_main(  struct Task1*this)
 {
      B1_main(&(this->b11));
       B1_main(&(this->b12));
  }
  struct  Task1 task1 =
 { 0,                  /* x init value*/
   0,                  /* y init value*/
    {  &(task1.x),   /*port v of  b11 */
         0               /* a init value */
    },  /*b11*/
    {  &(task1.y),    /*port v of b12*/
         0               /* a init value*/
     },  /*b12*/
   };
  void Task1()
  {
    Task1_main(&task1);
  }
```

(b) C Code

Fig. 4. An illustrative example

IV. Task Creation

In system design, the specification is written in SLDL as a network of hierarchical behaviors. However, in the implementation, many designers use a task-based approach, where a set of tasks are scheduled by a preemptive, priority-driven real time kernel. The software task generation process converts the design specification into a RTOS based multi-task model. Essentially, the task creation step synthesizes the *concurrency* and *communication* elements contained in the SLDL description.

A. Concurrency

Concurrency is supported by SLDL to express the parallel executing behaviors in system specification (e.g. `par` statement). The task creation step converts concurrent processes in the specification into RTOS-based tasks. Generally speaking, it involves dynamic creation of child tasks in a parent task. In this process, each SLDL concurrency statement (`par`) in the specification description is refined to dynamically fork child tasks as part of the parent's execution. After the child tasks finish execution and the `par` exits, the system joins with the children and resumes the execution of the parent task by the underlining RTOS [12].

This step is illustrated in Figure 3. The `par` statement in the input model (line 9-10 in Figure 3a) is converted to dynamically fork and join child tasks as part of the parent's execution (line 6-11 in Figure 3b). During this refinement process, the *os_task_create* methods of the children are called to create the child tasks (line 6,7 in Figure 3b). Then, *fork* is inserted before the *par* statement to suspend the calling parent task by the RTOS model before the children are actually executed in the `par` statement. After the two child tasks finish execution and the `par` exits, *join* is inserted to resume the execution of the parent task by the RTOS model.

B. Communication

In system specification, communications among processes are done through channels and control mechanisms for communication are described explicitly in the description of channels. We developed a standard channel library to be used in system specification, it includes common communication primitives, such as `c_semaphore`, `c_mutex`, `c_queue` and `c_double_handshake`. For our software generation process, only those channels inside the standard channel library are allowed in system specification. During the task creation process, these SLDL channels are replaced by the channels from the RTOS model library [12]. And the task synchronization as well as inter-task communication are handled by the RTOS model.

Now that the software part of the system model is refined to a multi-task system scheduled by the RTOS model. The designer can simulate the model and get feedback as regards to the timing properties of the system implementation. Some import parameters (e.g scheduling algorithm and task priority) are determined and checked in this step.

V. Task Code Generation

After the task creation step, those behaviors and channels mapped to the same processor are refined into multiple software tasks scheduled by a RTOS model. Each task is still written in SLDL as hierarchical behaviors (computation part of the task) and channels (communication part of the task). The task code generation step creates C code for each task from its SLDL task description. Essentially, this step synthesizes the *hierarchy* and *port mapping* elements contained in the SLDL description.

A. Code Generation Rules

The code generation process converts the SLDL description of tasks into ANSI C code. The main idea is that we convert the behaviors and channels into C `struct` and convert the behavioral hierarchy into the C language `struct` hierarchy. Rules for C code generation are as follows:

- R1: Behaviors and channels are converted into C `struct` and their structural hierarchy is represented by the C `struct` hierarchy,
- R2: Child behaviors and channels are instantiated `struct` members inside the parent `struct`,
- R3: Variables defined inside a behavior or channel are converted into data members of the corresponding C `struct`,
- R4: Ports of behavior or channel are converted into data members of the corresponding C `struct`,
- R5: Functions inside a behavior or channel are converted into global functions with an additional parameter added representing the behavior to which the function belongs,
- R6: A static `struct` instantiation for each PE is added at the end of the output C code to allocate the data used by software. Port mappings for the behaviors and channels inside the task are reflected in this `struct` initialization.

B. An Illustrative Example

Figure 4 is a simple example illustrating the code generation process. Figure 4a shows a software task *Task1* in SpecC SLDL. It consists of two instances of behavior *B1* executing sequentially. Figure 4(b) is the automatically generated C code from our tool. In figure 4, we can find the examples of the six rules for code generation process:

- R1: behavior *B1* is converted into `struct` *B1*,
- R2: In the input code, behavior *Task1* contains two instances of behavior *B1*(line 16,17 in Figure 4a). in the output C code, `struct` *Task1* contains two instances of `struct` *B1* (line 15,16 in Figure 4b),
- R3: `int` *a* defined in behavior *B1* is converted to `int` *a* inside `struct` *B1*,
- R4: port `int` v of behavior *B1* (line 1 in Figure 4a) are represented by `int` *v inside `struct` *B1* (line 3 in Figure 4b). Note that all the ports are represented by pointer type,
- R5: function *main* inside behavior *B1* is converted to a global function *B1_main* in the output C code. One additional parameter, `struct` *B1 *this* is added reflecting that this function belongs to behavior *B1* in the specification. We need this parameter because there might be multiple instances of *B1* and each has its own data members. In case only one instances exists. This parameter can be optimized away. Note that, inside function *B1_main*, references to data members of behavior *B1* are replaced by references to the data members of `struct` *B1*. For example, inside function *B1_main*, the variable *a* in the input code (line 8 in Figure 4a) is replaced by *this→a* in the output code (line 8 in Figure 4b),
- R6: the data used by task *Task1* is statically allocated through the instantiation of `struct` *Task1* (line 23 to line 36 in Figure 4b). Note that the initial values for data members in side `struct` *Task1* are all set at this time. This includes the port mapping information for behavior instances *b11,b12*. For example, the port mapping of behavior instance *b11* (line 16 in Figure 4a) is reflected in `struct` *B1* instantiation (line 26 of Figure 4b). The advantage of this approach is that the C compiler does the port mapping at compile time rather than the program calculates at run time. The goal here is to optimize the code at compile time as much as possible, such that the runtime is reduced to a minimum.

Algorithm 1 GenerateCCode($Design$, B_{Task})

```
for all Behavior $B \in Design$ do
    if IsChildBehavior($B$,$B_{Task}$) then
        GenerateC4Behavior($B$);
    end if
end for
for all Channel $C \in Design$ do
    if IsChildChannel($C$,$B_{Task}$) then
        GenerateC4Channel($C$);
    end if
end for
for all Function $F \in Design$ do
    if IsCalledInBehavior($F$,$B_{Task}$) then
        if IsMemberFunc($F$,$B_{Task}$) then
            $B$ = GetParentBehavior($F$);
            GenerateC4MemberFunction($F$,$B$);
        else
            GenerateC4GlobalFunction($F$);
        end if
    end if
end for
$TopInst$ = FindInstance($B_{Task}$)
GenerateStructInstance($TopInst$);
GenerateTaskCall($B_{Task}$);
```

Note that the generated C code has a clear structure. There are three separate code parts, namely, the `struct` definition part, the function definition part and the `struct` instantiation part. After the system compilation, the function definition part becomes the code segment while the `struct` instantiation part becomes the data segment for the final object file.

C. The Algorithm

We have implemented a code generation tool that can convert the software part of an embedded system described in SpecC into efficient ANSI C code. *Algorithm*1 reflects the six rules for software code generation in the previous section and it is used to generate C code for a task described in SpecC. The detailed explanation for *Algorithm*1 can be found in [19].

D. CoSimulation with System Model

As shown in Figure 2, to validate the software, the C code is co-simulated with the other part of the system using the SLDL simulator as a simulation backplane. In this process, the C code is imported to the design and wrapped by SLDL modules. The software part of the system specification code is then replaced by the C code with SLDL wrappers using SLDL's plug'n'play capabilities while task scheduling and inter-task communication are still provided by the RTOS model routines.

VI. Operating System Targeting

After the C code is validated through the co-simulation. The operating system targeting step generates the final read-to-compile C code. Generally, this means that each routine of the RTOS model interface will be mapped to 1 or N target RTOS APIs [12].

A. Task Management

In the target processor, task management is handled by the real RTOS calls. Without loss of generality, we use the POSIX pthread interface, which is supported by many RTOS[1, 5]. Some of the task management routines (e.g. $task\_create$) of the RTOS model can be directly mapped to the corresponding pthread interface routines. For those routines which can't be mapped to pthread interfaces, we implemented them using combinations of target RTOS APIs.

Figure 5 shows the generated C code for the example in Figure 3. In the output C code, two behaviors $Task\_B2$ and $Task\_B3$ (which represents two tasks in RTOS model) are turned into two POSIX threads (line 5 and line 10). They are created dynamically inside the thread $B2B3\_main$ (line 21 and line 24). The RTOS model routines (code commented out) are replaced by the corresponding phtread interface routines.

B. Task Communication

In our RTOS model, tasks communicate using channels from RTOS model library. In the final target software implementation, IPC (inter process communication) mechanisms (mutex, semaphore, mailbox, FIFO etc) are normally provided by RTOS to support task communication. Implementing communication means replacing functions of RTOS model channels with equivalent services of the target RTOS library routines. Currently, our software generation tool can only support channels from the RTOS model standard channel library.

C. Binary Code Creation

Depending on the number of tasks and the selected RTOS, a makefile is created for the chosen target platform. The generated C code can be cross compiled and linked against the target RTOS libraries to create the final binary executable file.

VII. Experimental Results

We developed a tool of creating ANSI C code from SpecC SLDL, it has been integrated into our system level design tool:

```
1  struct B2B3
2  { struct Task_B2 task_b2;
3    struct Task_B3 task_b3;
4  };
5  void *B2_main(void *arg)
6  { struct Task_B2 *this=(struct Task_B2*)arg;
7    ...
8    pthread_exit(NULL);
9  }
10 void *B3_main(void *arg)
11 { struct Task_B3 *this=(struct Task_B3*)arg;
12   ...
13   pthread_exit(NULL);
14 }
15 void *B2B3_main(void *arg)
16 {  struct B2B3 *this= (struct B2B3*)arg;
17    int status;
18    pthread_t *task_b2;
19    pthread_t *task_b3;
20    /*task_b2.os_task_create()*/
21    pthread_create(task_b2, NULL,
22                   B2_main, &this->task_b2);
23    /*task_b3.os_task_create()*/
24    pthread_create(task_b3, NULL,
25                   B3_main, &this->task_b3);
26    /*t = os.fork()
27      par { task_b2.main();
28            task_b3.main(); }*/
29    /*os.join(t)*/
30    pthread_join(*task_b2, (void **)&status);
31    pthread_join(*task_b3, (void **)&status);
32    pthread_exit(NULL);
33 }
```

Fig. 5. Task management implementation

	SPEC	TLM	SW(TLM)	C
Behaviors	102	127	96	0
Operations	16,614	19,526	14,573	23,868
Lines.	11,557	12,606	10,920	7,882

TABLE I
Vocoder experimental results.

the SoC Environment. We applied the software generation tool to the design of a voice codec for mobile phone applications [13]. The result is shown in Table I. The original specification contains 102 behaviors (16614 C level operations (plus,minus,multiply...), 11557 lines of SpecC code). After SW/HW partitioning and scheduling, we added 2 component behaviors representing two PEs and 23 behaviors for component communication and synchronization. The output transaction level model contains 127 behaviors (19526 operations, 12606 lines of SpecC code) in which the software part has 96 behaviors (14573 operations, 10920 lines of SpecC code). We applied the automatic software generation tool to the transaction level model and it generated 7,882 lines of C code (23868 operations) for the software part of Vocoder. The reason for more operations in the generated C code than in SLDL is that many pointer operations are introduced inside C functions to access C `struct` members.

To create the final executable, the model was compiled into binary code for the ARM processor and the RTOS model was replaced by the μC/OS-II real time operating system. The final executable image size is 75KB (47KB code/28KB data) in which the vocoder software is 52KB (33KB code/19KB data).

VIII. Conclusions

In this paper, we presented the steps for generating embedded software from system specification written in SLDL. The automation of embedded software creation process frees the designer from the tedious and error-prone work of creating software manually after SW/HW partition. Since the final software is derived from the specification, validation of the software code become easier than the manually written code. Also, the designer doesn't need to maintain two different versions of system code (for software specification and software implementation).

We developed a tool of creating ANSI C code from SLDL. Experiments are performed to show the usefulness of the tool in system design. Currently the tool is written for SpecC SLDL because of its simplicity. However, the concepts in this paper can also be applied to SystemC.

Future work includes automatically generating co-simulation model from the generated binary code to test the performance as well as validating the generated software with other parts of the system.

References

[1] QNX. Available: http://www.qnx.com/.

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

[3] SpecC. http://www.specc.org/.

[4] SystemC. http://www.systemc.org/.

[5] VxWorks.http://www.vxworks.com/.

[6] F. Balarin et al. *Hardware-Software Co-design of Embedded Systems – The POLIS approach*. Kluwer Academic Publishers, January 1997.

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

[8] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. L. Sangiovanni-Vincentelli. Task generation and compile time scheduling for mixed data-control embedded software. In *Proceedings of the Design Automation Conference*, June 2000.

[9] D. Desmet, D. Verkest, and H. D. Man. Operating system based software generation for system-on-chip. In *Proceedings of the Design Automation Conference*, June 2000.

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

[11] L. Gauthier et al. Automatic generation and targeting of application-specific operating systems and embedded systems software. *IEEE Trans. on CAD*, November 2001.

[12] A. Gerstlauer, H. Yu, and D. Gajski. RTOS modeling in system level design. In *Proceedings of Design, Automation and Test in Europe*, Mar. 2003.

[13] A. Gerstlauer, S. Zhao, D. Gajski, and A. M. Horak. Design of a GSM Vocoder using SpecC Methodology. Technical Report ICS-TR-99-11, UCI, Feburary 1999.

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

[15] D. Harel et al. Statemate: a working environment for the development of complex reactive systems. *IEEE Trans. on Software Engineering*, April 1990.

[16] F. Herrera, H. Posadas, P. Sanchez, and E. Villar. Systematic embedded software generation from systemc. In *Proceedings of Design, Automation and Test in Europe*, March 2003.

[17] B. Lin. Software synthesis of process-based concurrent programs. In *Proceedings of the Design Automation Conference*, 1998.

[18] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software synthesis for dsp using ptolemy. *Journal of VLSI Signal Processing*, 1995.

[19] H. Yu, R. Dömer, and D. Gajski. Automatic Software Generation for System Level Design. Technical Report CECS-03-18, UCI, May 2003.