# Automatic Software Generation for System Level Design

Haobo Yu, Rainer Doemer, Daniel Gajski

CECS Technical Report 03-18
May 14, 2003

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{haoboy,doemer,gajski}@ics.uci.edu

# Automatic Software Generation for System Level Design

Haobo Yu, Rainer Doemer, Daniel Gajski

CECS Technical Report 03-18
May 14, 2003

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425,USA
(949) 824-8059

{haoboy,doemer,gajski}@ics.uci.edu

## Abstract

*Raising the level of abstraction to system level promises to enable faster exploration of the design space at early stages. While software is playing an increasingly important part in embedded systems, today it is still a common practice to implement embedded software manually after HW/SW partitioning of the system. It is much desired that the designer can get the embedded software implementation directly from the system specification used in high level exploration. This paper presents a method of automatically creating embedded software from system level specification. We demonstrate the effectiveness of the proposed method by a tool which can generate efficient C code from system models written in system level design language(SLDL). The code generated is compliant to the ANSI C standard thus can be accepted by most compilers.*

## Contents

**1 Introduction** **1**

**2 Related Work** **2**

**3 Design Flow** **2**

**4 Task Creation** **3**
4.1 Concurrency . . . . . . . 3
4.2 Synchronization . . . . . . . 4

**5 Task Code Generation** **4**
5.1 Code Generation Rules . . . . . . . 4
5.2 An Illustrative Example . . . . . . . 5
5.3 The Algorithm . . . . . . . 6
5.4 CoSimulation with System Model . . . . . . . 7

**6 Operating System Targeting** **7**
6.1 Task Management . . . . . . . 7
6.2 Task Synchronization . . . . . . . 7
6.3 Binary Code Creation . . . . . . . 8

**7 Experimental Results** **8**

**8 Summary and Conclusions** **8**

## List of Figures


1 System design flow . . . 2
2 Software generation flow . . . 3
3 Refinement example . . . 4
4 Task creation . . . 4
5 Synchronization refinement . . . 4
6 An illustrative example . . . 5
7 Task management implementation . . . 7
8 Channel implementation . . . 8

# Automatic Software Generation for System Level Design

Haobo Yu, Rainer Doemer, Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697, USA
{haoboy,doemer,gajski}@cecs.uci.edu

## Abstract

*Raising the level of abstraction to system level promises to enable faster exploration of the design space at early stages. While software is playing an increasingly important part in embedded systems, today it is still a common practice to implement embedded software manually after HW/SW partitioning of the system. It is much desired that the designer can get the embedded software implementation directly from the system specification used in high level exploration. This paper presents a method of automatically creating embedded software from system level specification. We demonstrate the effectiveness of the proposed method by a tool which can generate efficient C code from system models written in system level design language(SLDL). The code generated is compliant to the ANSI C standard thus can be accepted by most compilers.*

## 1 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)[3, 4] and methodologies 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 [14]. 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 [18]. 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.

Mapping functionality described in SLDL to a software implementation usually means getting rid of all SLDL language elements (e.g. module declarations, process declarations, channels or complex data-types). Since the predominant SLDLs are C/C++ extensions [3, 4], 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, most target compilers for embedded processors may not even support C++ at all but just C [16]. 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*, *synchronization*). However, these features are not necessarily needed for the target SW code. Considering the limited memory space and execution power of embedded processors, we need to generate compact and efficient software code for implementation.

Various design methodologies exits for designing embedded software. Many of these starts from an abstract model of (UML[2], EFSM[19]). However,little work has been done on software code generation from system level design language (SLDL) [3, 4]. In this paper, we address this problem by describing a tool which can automatically generate efficient C code from the system specification written in SLDL.

The rest of this paper is organized as follows: Section 2 gives an insight into the related work on software code generation in system design. Section 3 describes how the automatic software generation is integrated with the system level design flow. Details of the code generation process are

covered in Section 4, Section 5 and Section 6. Experimental results are shown in Section 7 and Section 8 concludes this paper with a brief summary and an outlook on future work.

## 2 Related Work

Some related work can be found on code generation for embedded software. There are approaches to automatic code generation from DSP graphical programming environments, such as Ptolomy [21], from graphical finite state machine design environments (e.g StateCharts [17]), or from synchronous programming languages (e.g Esterel) [7].

In [20], a software synthesis approach from a concurrent process specification through intermediate Petri-Net is given. The proposed method applies quasi-static scheduling to a set of Petri-Nets to produce a set of corresponding state machines, which are then mapped syntactically to the final software code. In [8], a way of combining static task scheduling and dynamic scheduling in software synthesis is proposed. In [12], a method for automatic generation of application-specific operating systems and corresponding application software for a target processor is given. While these approaches mainly focus on software scheduling issues, no efficient code generation method from system specification is described.

In POLIS [6], software synthesis from *Co-design Finite State Machine* (CFSM) is presented. Code generation is performed in two steps: (1) transformation of the CFSM specification into a *s-Graph* and (2) translation of *s-Graph* into portable C code. A small customized operating system consisting of a scheduler and drivers for the I/O channels is used to correctly implement the run-time behavior of the input CFSMs. This work, however, is mainly for reactive real time systems and can't be easily extended to other more general frameworks.

In [9], software code generation from a high-level model of operating system called SoCOS is presented. However, SoCOS requires its own proprietary simulation engine and it requires manual refinement to get the software code. In [18], software generation from SystemC SLDL based on the redefinition and overloading of SystemC class library elements is presented. Their approach use the same SystemC code both for the system-level specification and for the target binary code generation. However, they have very strict requirements with regards to the input SystemC model (e.g no event wait/notify are allowed inside process and the processes lack a sensitivity list). Besides, there is no information regarding how the processes in the input SystemC model get scheduled in the final implementation, which is very important considering the real time requirements of the embedded software.

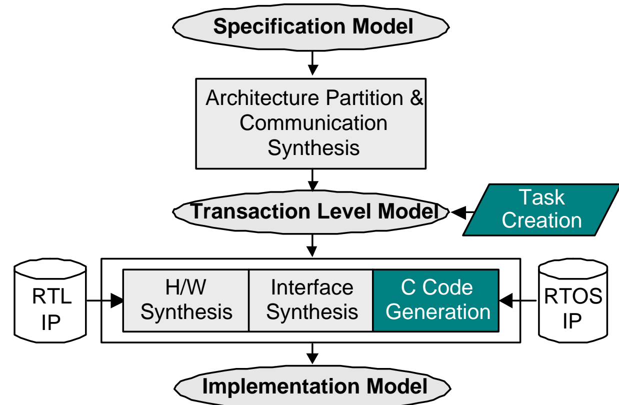



Figure 1. System design flow

## 3 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 [11]. The system design process starts with the specification model written by the designer to specify the desired system functionality. During system synthesis, the specification functionality is then partitioned onto multiple processing elements (PEs) and the communication synthesis generates the bus functional model in which a communication architecture consisting of busses and bus interfaces is synthesized to implement communication between PEs. Note that during communication synthesis, interrupt handlers will be generated inside the PEs as part of the bus drivers.

Due to the inherently sequential nature of PEs, behaviors mapped to the same PE need to be serialized. Depending on the nature of the PE and the data inter-dependencies, behaviors are scheduled statically or dynamically (In case of dynamic scheduling, a scheduling kernel is required for the final implementation). For each PE, the design specification is written in SLDL as a hierarchical network of behaviors/processes interconnected by channels. While in the implementation, the SW portion of the design is described as a set of tasks (processes) scheduled by a real time kernel (usually a RTOS). Thus creating multiple tasks from system specification and generating ready to compile C code for each task are the two major tasks of software implementation in system level design.

In our software generation flow shown in Figure 2, these two tasks are carried in two separate steps. Task creation is the first step, where the behaviors are converted into 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 [15]. 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 model 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 architecture model is then implemented separately. During this process, tasks generated in the previous step are converted into compilable C code for the chosen processor by our automatic software generation tool. By replacing the SLDL description of the multi-task system with the C code created in this step, the designer can co-simulate and validate the output C code using the system 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, the validated C code is compiled into the processor's instruction set and services of the RTOS model are mapped onto the API of a specific commercial or custom RTOS. The final binary executable for the chosen processor is generated by linking the compiled code against the RTOS libraries.

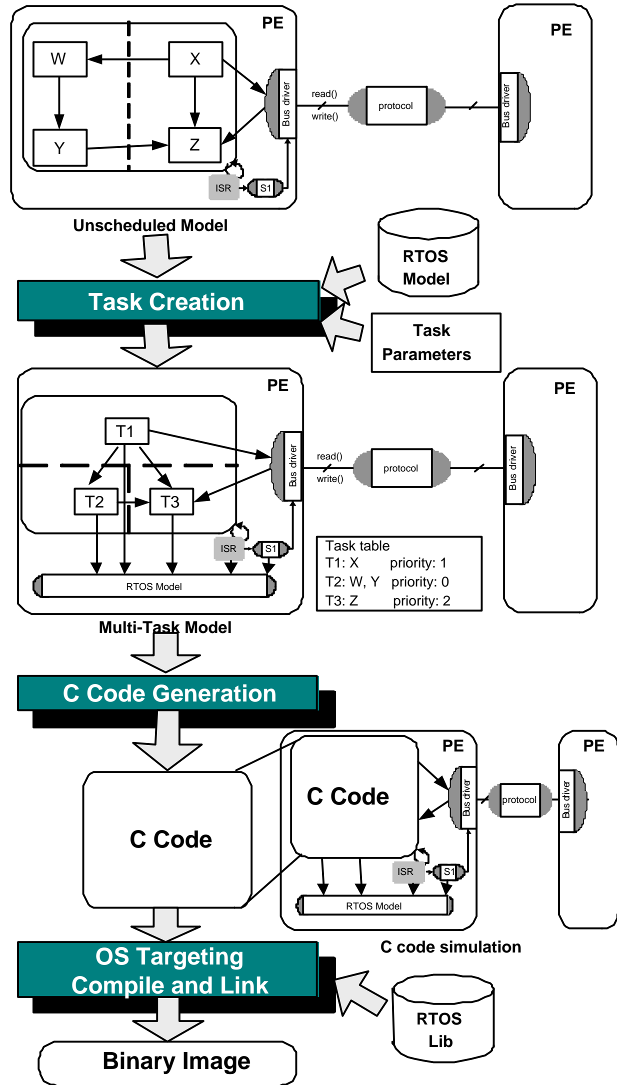



Figure 2. Software generation flow

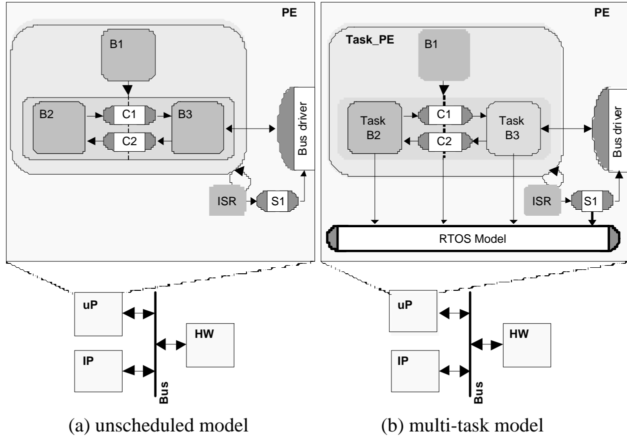
## 4 Task Creation

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 *synchronization* elements contained in the SLDL description.

In this section, we illustrate the task generation process through a simple yet typical example (Figure 3). The unscheduled model (Figure 3a) executes behavior *B1* followed by the parallel composition of behaviors *B2* and *B3*. Behaviors *B2* and *B3* communicate via two channels *C1* and *C2* while *B3* communicates with other PEs through a bus driver. Note that as part of the bus interface implementation, the interrupt handler *ISR* for external events signals the main bus driver through a semaphore channel *S1*.

### 4.1 Concurrency

Concurrency is supported by SLDL to express the parallel executing behaviors in system specification (e.g. `par` statement). However, due to the sequential execution nature of the processors, the concurrent processes in software PE need to be scheduled for the final implementation. Depending on the data inter-dependencies between these processes, they are scheduled statically and dynamically. In case of dynamic scheduling, we need to convert these 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 exe-




(a) unscheduled model (b) multi-task model

Figure 3. Refinement example

(a) before

```
1  behavior B2B3()
2  {B2 b2();
3   B3 b3();
4  void main(void)
5  {
6
7
8
9    par
10     { b2.main();
11       b3.main();
12     }
13
14 }
```

(b) after

```
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  {
10    task_b2.main();
11    task_b3.main();
12  }
13  os.join(t);
14 }
```

Figure 4. Task creation

cution. 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 [15].

This step is illustrated by our example in Figure 4. The `par` statement in the input model (line 9-12 in Figure 4a) is converted to dynamically fork and join child tasks as part of the parent's execution (line 6-13 in Figure 4(b)). During this refinement process, the *os_task_create* methods of the children are called to create the child tasks (line 6,7 in Figure 4(b)). 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.

(a) before

```
1  channel C1()
2  {event eRdy;
3   event eAck;
4   void send(...)
5   { ...
6     notify eRdy;
7     ...
8     wait(eAck);
9     ...
10  }
11 };
```

(b) after

```
1  channel C1(RTOS os)
2  {os_event eRdy;
3   os_event eAck;
4   void send(...)
5   { ...
6     os.evet_notify(eRdy);
7     ...
8     os.event_wait(eAck);
9     ...
10   }
11 };
```

Figure 5. Synchronization refinement

## 4.2 Synchronization

Synchronization in the system specification is implemented using channels or SLDL events. During the task creation process, the RTOS model provide routines to replace the SLDL synchronization primitives. All `event` instances are replaced with instances of RTOS model events *os_event* and `wait` / `notify` statements are replaced with RTOS model *event_wait* / *event_notify* calls [15]. Later all these RTOS model routines will be implemented by real RTOS library calls.

Figure 5 shows the synchronization refinement for our example: the `notify` / `wait` statement inside `channel` *C*1 in the input model (line 6,8 in Figure 5a) is refined into two event handling interface routines of the RTOS model in the output model (line 6,8 in Figure 5(b)).

# 5 Task Code Generation

After the task creation, those behaviors and channels mapped to a processor are refined into multiple software tasks scheduled by a RTOS model. Tasks communicate by using the synchronization primitives of the RTOS model. Each software task is written in SLDL as hierarchical behaviors (computation part of the task) and channels (communication part of the task). The task code generation step creates the C code for each task from the SLDL task descriptions.

## 5.1 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, i.e. a hierarchy of structs. Rules for C code generation are as follows:

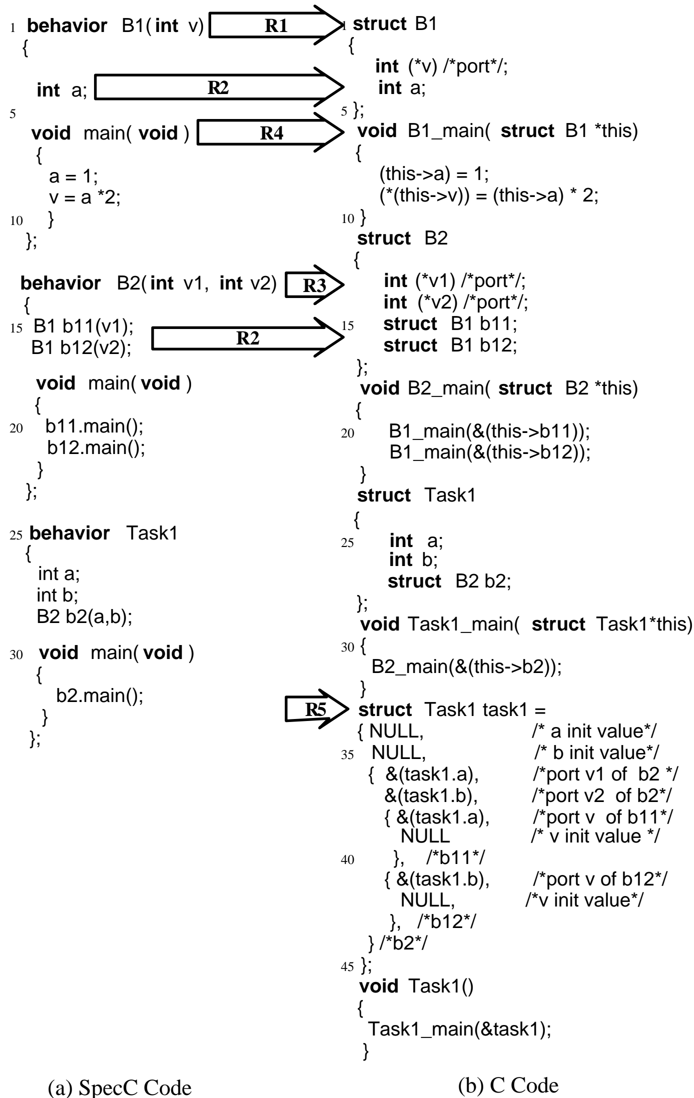



Figure 6. An illustrative example

- 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` member inside the parent `struct`. Variables defined inside a behavior or channel are converted into data members of the corresponding C `struct`;
- R3: Ports of behavior or channel are converted into data members of the corresponding C `struct`;
- R4: Functions inside a behavior or channel are converted into global functions with an additional parameter added representing the behavior to which the function belongs;
- R5: A static `struct` instantiation for each PE is added in the end of the output C code to allocate the data used by the task. Port mappings for the behaviors and channels inside the task are reflected in the C `struct` initialization;

## 5.2 An Illustrative Example

Figure 6 is a simple example illustrating the code generation process. Figure 6a shows a software task *Task1* in SpecC language. It consists of one instance of behavior *B2*. Behavior *B2* is hierarchical and it consists of two instances of behavior *B1* executing sequentially. Figure 6(b) is the automatically generated C code from our tool. In figure 6, we can find the examples of the six rules for code generation process:

- R1: behavior *B1* (line 1 in Figure 6a) is converted into `struct` *B1*(line 1 in Figure 6b);
- R2: `int` *a* defined in behavior *B1*(line 4 in Figure 6a) is converted to `int` *a* inside `struct` *B1* (line 4 of Figure 6b). In the input code, behavior *B2* contains two instances of behavior *B1*(line 15,16 in Figure 6a), in the output C code, `struct` *B1* contains two instance of `struct` *B2* (line 15,16 in Figure 6b).
- R3: function *main* inside behavior *B1* (line 13 in Figure 6a) is converted to a global function *B1_main* in the C output code (line 6 in Figure 6b). One additional parameter, `struct` *B1 *this* is added reflecting that this function belongs to behavior *B1* in the specification. 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 6a) is replaced by *this*$\rightarrow$*a* in the output code (line 8 in Figure 6b);
- R4: port `int` $v1$ and port `int` $v2$ of behavior *B2* (line 13 in Figure 6a) are represented by `int` $*v1$, `int` $*v2$ inside `struct` *B2* (line 13,14 in Figure 6b). Note that all the ports are represented by pointer type. Again, optimization maybe be applied to get rid of these pointers later.
- R5: the data used by task *Task1* is statically allocated through the instantiation of `struct` *Task1* (line 33 to line 45 in Figure 6b). Note that the initial values for data members in side `struct` *Task1* are all set at

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

```
1:  for all Behavior B ∈ SIR_Design do
2:      if IsChildBehavior(B,B_Task) then
3:          GenerateC4Behavior(B);
4:      end if
5:  end for
6:  for all Channel C ∈ SIR_Design do
7:      if IsChildChannel(C,B_Task) then
8:          GenerateC4Channel(C);
9:      end if
10: end for
11: for all Function F ∈ SIR_Design do
12:     if IsCalledInBehavior(F,B_Task) then
13:         if IsMemberFunc(F,B_Task) then
14:             B = GetParentBehavior(F);
15:             GenerateC4MemberFunction(F,B);
16:         else
17:             GenerateC4GlobalFunction(F);
18:         end if
19:     end if
20: end for
21: TopInst = FindInstance(B_Task)
22: GenerateStructInstance(TopInst);
23: GenerateTaskCall(B_Task);
```

this time. This includes the port mapping information for behavior instances *b11,b12,b2*. For example, the port mapping of behavior instance *b2* (line 29 in Figure 6a) is reflected in `struct` *B2* instantiation(line 36,37 of Figure 6b). 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 run-time is reduced to a minimum.

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

### 5.3 The Algorithm

We have implemented a code generation tool that can convert the software part of an embedded system described in SpecC into efficient, read-to-compile ANSI C code. The two main algorithms for this sofware, *Algorithm*1 and *Algorithm*2 reflect the five rules for software code generation in the previous section.

**Algorithm 2** GenerateStructInstance(*Inst*)

```
1:  for all Port P ∈ InstS do
2:      PrintPortMapping(P,Inst);
3:  end for
4:  for all Memember Variable V ∈ InstS do
5:      PrintInitValue(V,Inst);
6:  end for
7:  for all Behavior Instance bhvrinst ∈ Inst do
8:      GenerateStructInstance(bhvrinst);
9:  end for
10: for all Channel Instance chnlinst ∈ Inst do
11:     GenerateStructInstance(chnlinst);
12: end for
```

*Algorithm*1 generate C code for a task described in SpecC. The input to *Algorithm*1 is the SpecC Internal Representation[10] for the whole design $SIR_{Design}$ and the top level behavior for a software task $B_{Task}$.

In *Algorithm*1, there are several functions used :

- `IsChildBehavior(`$B$`,`$B_{Task}$`)` returns true if $B$ is a child behavior inside behavior $B_{Task}$;
- `IsChildChannel(`$C$`,`$B_{Task}$`)` returns true if $C$ is a child channel inside behavior $B_{Task}$;
- `IsCalledInBehavior(`$F$`,`$B_{Task}$`)` returns true if function $F$ is used inside behavior $B_{Task}$ or any child behavior of behavior $B_{Task}$;
- `IsMemberFunc(`$F$`,`$B_{Task}$`)` returns true if function $F$ is a member function of behavior $B_{Task}$ or is a member function of $B_{Task}$'s child behavior;
- `GenerateC4Behavior(`$B$`)` generates the C `struct` definition for behavior $B$ and `GenerateC4Channel(`$C$`)` generates the C `struct` definition for channel $C$;
- `GenerateC4MemberFunction(`$F$`,`$B$`)` generates the C code for member function $F$ of behavior $B$ and `GenerateC4GlobalFunction(`$F$`)` generates the C code for global function $F$;
- `GenerateTaskCall(`$B_{Task}$`)` generates the main function code for task $B_{Task}$;
- `GenerateStructInstance(`*TopInst*`)` generates the C `struct` instantiation for *TopInst*. It is described in *Algorithm*2;

*Algorithm* 2 implements *Rule* 5 in Section 5.1. `GenerateStructInstance(`*Inst*`)` is a recursive function. It prints port mapping of *Inst* (line 1-3 in *Algorithm*2) and set the initial value for its data members (line 4-6 in *Algorithm*2). Then it instantiates the child behaviors and child

| RTOS model API | POSIX pthread API |
|---|---|
| *task_create* | *pthread_create* |
| *task_terminate* | *pthread_exit* |
| *task_kill* | *pthread_kill* |
| *fork* | |
| *join* | *pthread_join* |
| *task_sleep* | |
| *task_resume* | |

Table 1. Task management routines

channels inside *Inst* by calling itself recursively(line 7-12 in *Algorithm* 2).

### 5.4 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 wrapper using the simulator's plug'n'play capabilities. Task scheduling and intertask communication are supported by the RTOS model. From the simulation result, the designer can get feedback as regards to the timing properties of the system implementation. Some import parameters (i.e. scheduling algorithm and task priority) are determined and checked in this step.

## 6 Operating System Targeting

After the generated task code(in ANSI C) is validated through the co-simulation. The operating system targeting step generates the final read-to-compile C code. In this process, all the RTOS model interfaces routines will be mapped to real RTOS system calls.

### 6.1 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]. As shown in Table 1, some of the task management routines in the RTOS model can be mapped to a corresponding pthread interface routine. For those routines which can't be mapped to pthread interfaces (*task_sleep* and *task_resume*), we implemented them using target specific instructions.

Figure 7 shows the generated C code for the example in Figure 4. In the output C code, two behaviors *Task_B*2 and *Task_B*3(which represents two tasks in RTOS model) are turned into two POSIX threads (line 5 and line 10). They

```
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 {
28         task_b2.main();
29         task_b3.main();
30      }*/
31    /*os.join(t)*/
32    pthread_join(*task_b2, (void **)&status);
33    pthread_join(*task_b3, (void **)&status);
34    pthread_exit(NULL);
35 }
```

Figure 7. Task management implementation

are created dynamically inside the thread *B*2*B*3_*main* (line 21 and line 24). The RTOS model routine calls (code commented out) are replaced by their corresponding phtread interface counterparts.

### 6.2 Task Synchronization

In our RTOS model, tasks are synchronized by event wait/notify or through SLDL channels. In the final target software implementation, IPC (inter process communication) mechanisms (mutex, semaphore, mailbox, FIFO etc) are normally provided by RTOS to support task synchronization. Implementing synchronization means replacing all events and event-related primitives in the specification with corresponding event handling routines of the RTOS library. Also, if the semantics of channels are known (e.g. fifo channel, semaphore channel), they can be replaced directly by an equivalent service of the actual RTOS library routines.

During the operating system targeting process, task synchronization routines in the RTOS model will be re-

```
struct C1
{ event_handle eRdy;
  event_handle eAck;
};

void C1_send(struct C1 *this...)
{...
 os_notify(this->eRdy);
 ...
 os_wait(eAck->eAck);
 ...
}
```

Figure 8. Channel implementation

placed by RTOS IPC routines: the RTOS model event handling routines `event_wait` and `event_notify` are replaced by two C functions: $os_wait(event_handle)$ and $os_notify(event_handle)$. Depending on the target RTOS, these two functions can be implemented in different ways (for example, these two functions can be implemented by using *WAIT/POST* operation on a zero-valued semaphore). At the same time, all the standard library channels (*c_os_semaphore*, *c_os_mutex*, etc) can be directly mapped to the corresponding RTOS IPC routines. The remaining user defined channels can be implemented by using combinations of $os_wait(event_handle)$ and $os_notify(event_handle)$.

Figure 8 shows the generated C code for the example in Figure 5. In the output C code, a C struct $C1$ (line 1) contains the the data members inside channel $C1$. Note that the SLDL event is replaced by C data structure *event_handle*. Function *C1_send* (line 6) implement the *send* method of channel $C1$. Note that the first parameter of function *C1_send* is a pointer to the C struct $C1$.

### 6.3 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 compiled and linked against the RTOS libraries to create the final binary executable file.

## 7 Experimental Results

We applied the software generation tool to the design of a voice codec for mobile phone applications [13]. The original specification contains 11,570 lines of SpecC code. After SW/HW partitioning and scheduling, the system model contains 26,476 lines of code. We used the code profiling tool to the partitioned system model, about 13,288 code operations are inside the S/W part. Finally, we applied our software generation tool to the design and it generated 7,882 lines of C code from the system specification.

To create the final executable, the model was compiled into binary code for the ARM processor and the RTOS model was replaced by the $\mu$C/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).

## 8 Summary and 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 ( software specification and software implementation ).

We developed a tool of creating ANSI C code from the 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 works includes automatically generating co-simulation model from the generated software binary image and instruction set simulator to test the performance as well as validating the generated code with the hardware part.

## 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. Task generation and compile time scheduling for mixed data-control embedded software. In *Proceedings of the Design Automation Conference*, June 2000.

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

[10] R. Domer. The SpecC internal representation. Technical report, University of California, Irvine, January 1999.

[11] D. Gajski et al. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

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

[13] A. Gerstlauer et al. Design of a GSM Vocoder using SpecC Methodology. Technical Report ICS-TR-99-11, University of California, Irvine, Feburary 1999.

[14] A. Gerstlauer and D. Gajski. System-level abstraction semantics. In *Proceedings of International Symposium on System Synthesis*, October 2002.

[15] A. Gertlauer, H. Yu, and D. Gajski. RTOS modeling in system level design. *Proceedings of Design Automation and Test in Europe(DATE)*, 2002.

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

[17] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: a working environment for the development of complex reactive systems. *IEEE transactions on software engineering*, April 1990.

[18] F. Herrera, H. Posadas, P. Snchez, and E. Villar. Systematic embedded software generation from systemc. *Proceedings of Design Automation and Test in Europe(DATE)*, 2003.

[19] Y. Jiang and R. K. Brayton. Software synthesis from synchronous specifications using logic simulation techniques. *Proceedings of Design Automation Conference (DAC)*, June 2002.

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

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