RTOS Modeling in System Level Synthesis

Haobo Yu and Daniel Gajski

CECS Technical Report 02-25
Aug 14, 2002

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

{haoboy,gajski}@ics.uci.edu

RTOS Modeling in System Level Synthesis

Haobo Yu and Daniel Gajski

CECS Technical Report 02-25
Aug 14, 2002

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

{haoboy,gajski}@ics.uci.edu

Abstract

System level synthesis is widely seen as the solution for closing the productivity gap in system design. High level system models are used in system level synthesis for early design exploration. While real time operating systems (RTOS) are an increasingly important component in system design, specific RTOS implementations can not be used directly in high level models. On the other hand, existing system level design languages (SLDL) lack support for RTOS modeling. In this paper we propose a RTOS model built on top of existing SLDLs which, by providing the key features typically available in any RTOS, allows the designer to model the dynamic behavior of multi-tasking systems at higher abstraction levels to be incorporated into existing design flows. Experimental result shows that our RTOS model is easy to use and efficient while being able to provide accurate results.

Contents

List of Figures

RTOS Modeling in System Level Synthesis

Haobo Yu and Daniel Gajski
Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine

Abstract

System level synthesis is widely seen as the solution for closing the productivity gap in system design. High level system models are used in system level synthesis for early design exploration. While real time operating systems (RTOS) are an increasingly important component in system design, specific RTOS implementations can not be used directly in high level models. On the other hand, existing system level design languages (SLDL) lack support for RTOS modeling. In this paper we propose a RTOS model built on top of existing SLDLs which, by providing the key features typically available in any RTOS, allows the designer to model the dynamic behavior of multi-tasking systems at higher abstraction levels to be incorporated into existing design flows. Experimental result shows that our RTOS model is easy to use and efficient while being able to provide accurate results.

1. Introduction

In order to handle the ever increasing complexity and time-to-market pressures in the design of systems-on-chip (SOCs), raising the level of abstraction is generally seen as a solution to increase productivity. Various system level design languages (SLDL) [Spe, Sys] and methodologies have been proposed in the past to address the issues involved in system level design. However, most SLDLs offer little or no support for modeling the dynamic real-time behavior often found in embedded software. In the implementation, this behavior is typically provided by a real time operating system (RTOS) [QNX, VxW]. At an early design phase, however, using a detailed, real RTOS implementation would negate the purpose of an abstract system model. Furthermore, at higher levels, not enough information might be available to target a specific RTOS. Therefore, we need techniques to capture the abstracted RTOS behavior in system level models.

In this paper, we address this design challenge by introducing a high level RTOS model for system design. It is written on top of existing SLDLs and doesn't require any specific language extensions. It supports all the key concepts found in modern RTOS like task management, real time scheduling, preemption, task synchronization, and interrupt handling [C.B99]. On the other hand, it requires only a minimal modeling effort in terms of refinement and simulation overhead. Our model can be integrated into the existing system level synthesis flows to accurately evaluate a potential system design (e.g. in respect to timing constraints) for early and rapid design space exploration.

The rest of this paper is organized as follows: Section 2 gives an insight into the related work on software modeling and synthesis in system level design. Section 3 describes how the RTOS model is integrated with the system level design flow. Details of the RTOS model, including its interface and usage as well as the implementation are covered in Section 4, Section 6 and Section 7. Experimental results are shown in Section 8 and Section 9 concludes this paper with a brief summary and an outlook on future work.

2. Related Work

A lot of work recently has been focusing on automatic RTOS and code generation for embedded software. In [G$^+$01], a method for automatic generation of application-specific operating systems and corresponding application software for a target processor is given. In [Cor00], a way of combining static task scheduling and dynamic scheduling in software synthesis is proposed. While both approaches mainly focus on software synthesis issues, their papers do not provide any information regarding high level modeling of the operating systems integrated into the whole system.

In [T$^+$01], a technique for modeling fixed-priority preemptive multi-tasking systems based on concurrency and exception handling mechanisms provided by SpecC is shown. However, their model is limited in its support for different scheduling algorithms and inter-task communication, and its complex structure makes it very hard to use.

Our method is similar to [D$^+$00], where they present

a high-level model of OS called SoCOS. The main difference is that our RTOS model is written on top of existing SLDLs whereas SoCOS requires its own proprietary simulation engine. By taking advantage of the SLDL's existing modeling capabilities, our model is simple to implement yet powerful and flexible, and it can be directly integrated into any system model and design flow supported by the chosen SLDL. Besides, while their model focused on task concurrency issues, our model is mainly focused on simulating the real time activities(task preemption,interrupt handling,real time scheduling)since real time analysis is a critical task in embedded software design. As a result, our RTOS model has to contain much more complex and complete services related with real time issues.

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. This refinement is achieved in a stepwise manner through several level of abstraction. After each step, a refined system model allows validation of the implementation detail introduced.

Figure 1 shows a typical system level design flow. 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), some or all of the concurrent processes mapped to a PE are statically scheduled, and 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, processes mapped to the same PE need to be serialized. Depending on the nature of the PE and the data inter-dependencies, processes are scheduled statically or dynamically. In case of dynamic scheduling, in order to validate the system model at this point, a representation of the dynamic scheduling implementation, which is usually handled by a RTOS in the real system, is required. Therefore, a high level model of the underlying RTOS is needed for inclusion into the system model during system synthesis. The RTOS model provides an abstraction of the key features that define a dynamic scheduling behavior independent of any specific RTOS implementation.

The dynamic scheduling step in Figure 1 refines the unscheduled system model into the final architecture model. Based on the selected scheduling strategy, a corresponding RTOS model is imported from the library. Processes are converted into tasks with assigned priorities. Synchronization as part of communication between processes is refined into OS-based task synchronization.

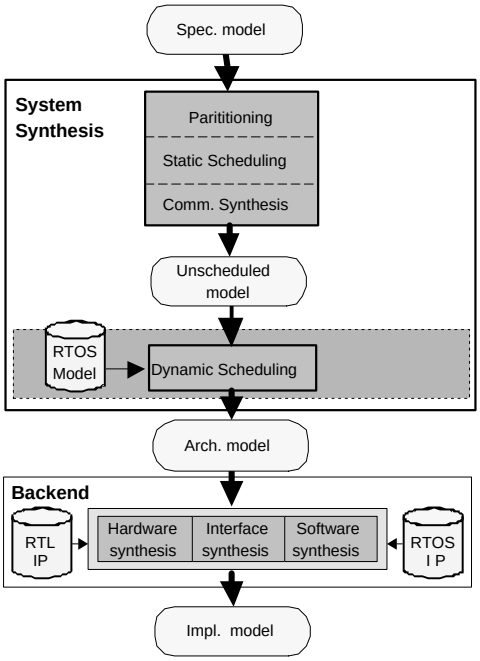
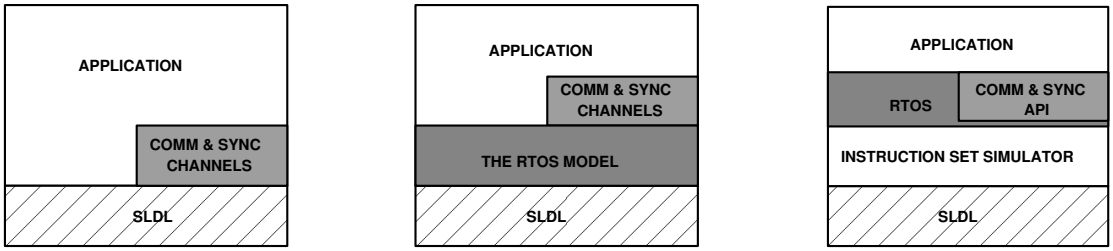


Figure 1. Design flow.

The resulting model can be validated through simulation or verification to evaluate different dynamic scheduling approaches as part of system design space exploration. From the simulation result, we can check wether the timing constraint as well as the resource constraint are met or not. If the model fails to meet the constraints, we can either change the scheduling algorithm, assign different priorities to the tasks, or change the software/hardware partitioning to get a different system model. This simulation and adjusting process continues until we find a system partitioning in which the software model can satisfy all the relevant timing and resource constraints.

In the backend, each PE in the architecture model is then implemented separately. Custom hardware PEs are synthesized into a RTL description. Bus interface implementations are synthesized in hardware and software. Finally, software synthesis generates code from the PE description of the processor in the architecture model. During this process, services of the RTOS model are mapped onto the API of a specific off-the-shelf or custom RTOS. At the same time, tasks generated in the dynamic scheduling step are converted into RTOS-dependent compilable C code for the chosen processor. Finally, the C code is then compiled into the processor's instruction set and linked against the RTOS libraries to produce the final executable.



(a) specification model (b) architecture model (c) implementation model

Figure 2. Modeling layers.

4. The RTOS Model

As mentioned previously, the RTOS model is implemented on top of an existing SLDL kernel. Figure 2 shows the modeling layers at different steps of the design flow. In the specification model (Figure 2(a)), the application is a serial-parallel composition of SLDL processes. Processes communicate and synchronize through variables and channels. Channels are implemented using primitives provided by the SLDL core and are usually part of the communication library provided with the SLDL.

In the architecture model (Figure 2(b)), the RTOS model is inserted as a layer between the application and the SLDL core. The SLDL primitives for timing and synchronization used by the application are replaced with corresponding calls to the RTOS layer. In addition, calls of RTOS task management services are inserted. The RTOS model implements the original semantics of SLDL primitives plus additional details of the RTOS behavior on top of the SLDL core. Existing SLDL channels (e.g. semaphores) from the specification are reused by refining their internal synchronization primitives to map to corresponding RTOS calls. Using existing SLDL capabilities for modeling of extended RTOS services, the RTOS library can be kept small and efficient. Later, as part of software synthesis in the backend, channels are implemented by mapping them to an equivalent service of the actual RTOS or by generating channel code on top of RTOS primitives if the service is not provided natively.

Finally, in the implementation model (Figure 2(c)), the compiled application linked against the real RTOS libraries is running in an instruction set simulator (ISS) as part of the system co-simulation in the SLDL.

We implemented the RTOS model on top of the SpecC SLDL [Spe]. In the following sections we will discuss the interface between application and the RTOS model, the refinement of specification into architecture using the RTOS interface, and the implementation of the RTOS model behavior.

```
1  interface RTOS {
2      /* OS management */
3      void init(void);
4      void start(int sched_alg);
5      void interrupt_return(void);
6      /* Task managment */
7      proc task_create(char *name, int type,
8           sim_time period, sim_time wcet);
9      void task_terminate(void);
10     void task_sleep(void);
11     void task_activate(proc tid);
12     void task_endcycle(void);
13     void task_kill(proc tid);
14     proc par_start(void);
15     void par_end(proc p);
16     /* Event handling */
17     evt  event_new(void);
18     void event_del(evt e);
19     void event_wait(evt e);
20     void event_notify(evt e);
21     /* Time modeling */
22     void time_wait(sem_time nsec);
23 };
```

Figure 3. Interface of the RTOS model.

4.1 RTOS Interface

Figure 3 shows the interface of the RTOS model. The RTOS model provides four categories of services: operating system management, task management, event handling, and time modeling.

Operating system management mainly deals with initialization of the RTOS during system start where *init* initializes the relevant kernel data structures while *start* starts the multi-task scheduling. In addition, *interrupt_return* is provided to notify the RTOS kernel at the end of an interrupt service routine.

Task management is the most important function in the RTOS model. It includes various standard routines such as task creation (*task_create*), task termination (*task_terminate*, *task_kill*), and task suspension and activation (*task_sleep*, *task_activate*). Two special routines are introduced to model dynamic task forking and joining: *par_start* suspends the calling task and waits for the child tasks to finish after which *par_end* resumes the calling task's execution. Our RTOS model supports both periodic hard real time tasks and non-periodic real time tasks. In modeling of periodic tasks, *task_endcycle* notifies the kernel that a periodic task has finished its execution in the current cycle.

Event handling re-implements the semantics of SLDL synchronization events in the RTOS model. SpecC events are replaced with RTOS events (allocated and deleted through *event_new* and *event_del*). Two system calls *event_notify* and *event_wait* are used to replace the SpecC primitives for event `notify` and event `wait`.

During simulation of high level system models, the logical time advances in discrete steps. SLDL primitives (such as `waitfor` in SpecC) are used to model delays. For the RTOS model, those delay primitives are replaced by *time_wait* calls which model task delays in the RTOS while enabling support for modeling of task preemption.

4.2 Scheduling in the abstract RTOS model

Real world embedded systems usually consist of computational activities having different characteristics. For example, tasks may be periodic,aperiodic, time-driven,and event driven and may have different levels of criticalness. However, in our abstract RTOS model, only two classes of tasks are considered:

- RT_PERIODIC : Periodic hard real time tasks,having a critical deadline,
- RT_APERIODIC : Non periodic real time tasks, having a fixed priority.

Periodic hard real time tasks are the highest priority tasks , they are scheduled using the earliest deadline first(EDF) algorithm, whereas non periodic real time tasks are executed in background based on their priority. However, other options of scheduling algorithm,such as first-come-first-serve algorithm, round robin algorithm for non periodic real time tasks can also be integrated in our abstract RTOS model. Generally, our abstract RTOS model can include any kind of scheduling mechanism, it is up to the user to select the real time scheduling algorithms for the design.

4.3 The extended RTOS service layer

As the extension to the basic RTOS service layer, we add some pre-defined channels to be used in task synchronization and communication. These are the features that most RTOS will offer, so later when we implement the software model , we can convert these channels directly to the actual RTOS system calls. These channels and their functionalities are described in Table 1.

```
1  behavior task0(RTOSLIB os) implements Init
2  {
3   proc me;
4     void init(void)
5    {
6      me=os.task_create("task0",
7                        RT_APERIODIC,3,2);
8    }
9
10    void main(void)
11   {
12     int a;
13     //task0 code start
14     os.task_activate(me);
15
16     //...code block1
17     os.time_wait(200);
18     if (a > 0){
19         //...code block2
20         os.time_wait(300);
21     } else {
22         //...code block3
23         os.time_wait(200);
24     }
25
26     //...code block4
27     os.time_wait(500);
28
29     //task0 code end
30     os.task_terminate();
31   }
32 };
```

Figure 4. Example code of a task

5. Preemptive multi-task system modeling

By using the RTOS model, multi-task systems can be specified in higher level easily. In this section, we start by showing the timing granularity in high level modeling, then we illustrate system modeling by an small example.

5.1 Timed computation and granularity

In our methodology, during the system synthesis, after behaviors have been partitioned onto processing elements

	Channels in the extended RTOS layer
c_semaphore	protected access to shared resources
c_mutex	protected access to one shared resource
c_mailbox	message passing services between tasks
c_critical_section	protected access to a critical section
c_queue	message queues

Table 1. Channels used for inter-task communication and synchronization in the extended RTOS service layer

of a system architecture, the concept of time is introduced into the model. The computation represented by the behaviors is refined to include execution times on the target components. Behavior execution delays can be based on estimated execution times derived from a model of the target component. Alternatively, execution delays can describe a timing budget allocated for different behaviors. These budgets will later serve as timing constraints for the behavior implementation on the target PEs.

Execution times can be specified on different levels of granularity, ranging from the statement level to the behavior level. Execution delays at the behavior level are used to model average or worst-case execution times of the corresponding behavior. On the other hand, execution times at the basic-block level can accurately model even data-dependent delays. In our software modeling, we choose to have the granularity of one basic-block, for it is the best trade off between the statement level granularity and the behavior level granularity.

Execution time is introduced into the system model for feedback during simulation. This is accomplished by inserting time delay statements as timing annotations to the behaviors. In addition to validation and verification, these timing annotations will also serve as constraints for the scheduling.

5.2 Task modeling

A task is a preemptable function runs from its input ports to its output ports. Usually, the task will be divided into several basic-blocks. Inside these basic blocks, codes are executed sequentially from begin to end, without any branch instructions inside the basic blocks. At the end of each basic-block, a *time_wait* statement is inserted to represent the execution delay of the codes inside the basic-block.

The reason for modeling task code in this way is that, to our simulation engine, the only way of advancing simulation time is by using *time_wait* function, all other code executions are carried out in zero time. So the *time_wait* is used to model the delay of codes in a basic-block . Also, all the tasks can only be preempted at the boundary of basic-blocks,i.e. it is inside the *time_wait* or other system call that the context switch happens.

Figure 7 is what a typical task would looks like: the task is modeled as a behavior, there's two functions inside the behavior, function *init* is used to create the task and function *main* is the main body of the task. There's four basic-blocks inside *main*, block 1 is all the code before the if statement and block 2 and block 3 are two branches of the if statement, the code after the if statement is block 4. The *time_wait* statement inside each block represents the execution time of the codes above it.

5.3 Modeling example

We demonstrate how to model the preemptive multi-task system using our abstract RTOS model by an example. Appendix A is the code modeling a preemptive system consisting of three non-periodic real time tasks(task0,task1 and task2)(line 5,28,55). The priority of task0, task1 and task2 are 3,2 and 1 (line 12,35,62) with task0 having the lowest priority and task2 having the highest priority. Task0 is executed sequentially and the code is divided into two code blocks, with two time delay statement *time_wait(20)* (line 20) and *time_wait(30)* (line 22) to model the execution time of these two blocks. Task1 is waiting on a semaphore sem1 (line 43) to to start its execution. And there's two code blocks inside task1 with the delay of 30 (line 46) and 40 nano seconds (line 49) respectively. Finally, task2 has two code blocks,the first block has delay time of 10 nano seconds (line 71) and after task2 finishes executing the first block, it waits on semaphore sem2 (line 74) before continuing to execute the second block, which has delay time of 30 nano seconds (line 77).

The three tasks are running concurrently on a processor,which is modelled by the par statement (line 96) in the TASKS behavior. Also, there are two interrupts e1 and e1,they are modelled by the *try* {...} *interrupt (e1,e2)* {... } (line 143,144) statement in behavior Processor. These two interrupts will evoke the interrupt service routine behavior ISR (line 104), which will release the semaphore sem1 (line 111) and sem2 (line 116) to make task1 and task2 continue execution.

Figure 5 is the simulation timing of the model. Initially, when the system starts, it selects the task with the highest priority,i.e task2 to execute. When task2 finishes code block1 at time 10, it waits on the semaphore sem2 and the scheduler selects the next highest priority task, i.e. task1, to execute. Task1 starts the execution and wait on semaphore sem1. Finally, the lowest priority task task0 is selected to execute. It executes the first block and finishes at time 30. Meanwhile, the first interrupt e1 occurs at time 20 and the interrupt service routine releases semaphore sem1 so that

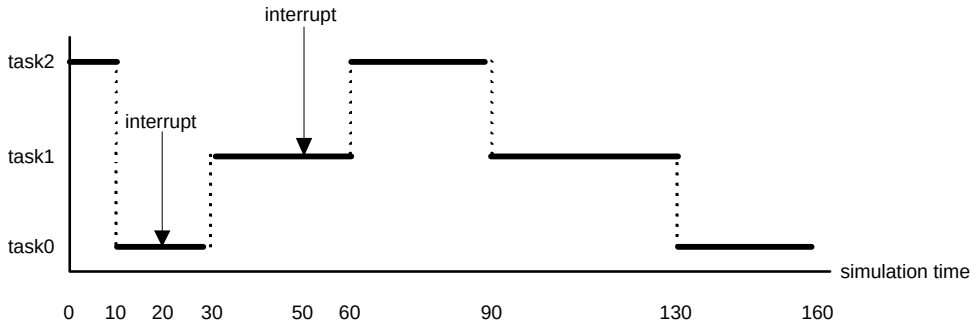


Figure 5. Gantt chart of the multi-task model simulation result

task1 can continue execution. So at time 30 the scheduler chooses task1, which has higher priority than task0, to execute. Task1 executes its first code block and stops at time 60. During this time period, the second interrupt e2 happens at time 50 and the interrupt service routine releases semaphore sem2 and puts task2 into the ready queue. At time 60, task2 is the task with the highest priority in the ready queue and is scheduled to execute. It executes the second code block and terminates at time 90. After that, task1 continues execution and terminates at time 130. Finally, task0 executes and terminates at time 160.

As we can find in the above example, behavior Tasks, ISR and Processor are three most important behaviors in the software system model. Behavior Tasks contains all the tasks in the system, behavior ISR is used to model the interrupt service routine and behavior Processor is the container behavior for the whole system. All the tasks are created(yet not activated)in behavior Tasks. And later, these tasks can be activated at proper time according to the system specification. In the above example, the three tasks are activated at the same time inside the par statement. But it's not always the case that all the tasks are activated simultaneously inside a par statement, it is up to the designer to choose when the tasks should be activated. We can, of course, activate task0 first and later activate task1 and task2 simultaneously in a par statement for the three-task system example shown above.

6. Model Refinement

In this section, we will illustrate application model refinement based on the RTOS interface presented in the previous section through a simple yet typical example (Figure 6). The unscheduled model (Figure 6(a)) 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. As part of the bus interface implementation, the interrupt handler *ISR* for external events signals the main bus driver through a semaphore channel *sem*.

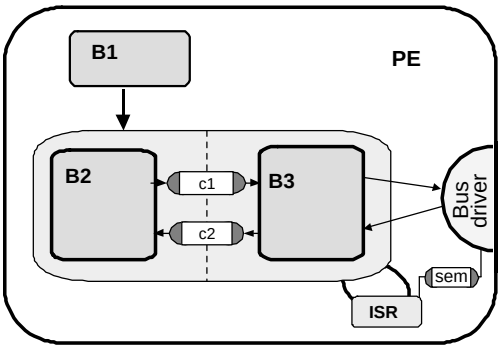
The output of the dynamic scheduling refinement process is shown in Figure 6(b). The RTOS model implementing the RTOS interface is instantiated inside the PE in the form of a SpecC channel. Behaviors, interrupt handlers and communication channels use RTOS services by calling the RTOS channel's methods. Behaviors are refined into three tasks. *Task_PE* is the main task which executes as soon as the system starts. When *Task_PE* finishes executing *B1*, it spawns two concurrent child tasks, *Task_B2* and *Task_B3*, and waits for their completion.

6.1 Task refinement

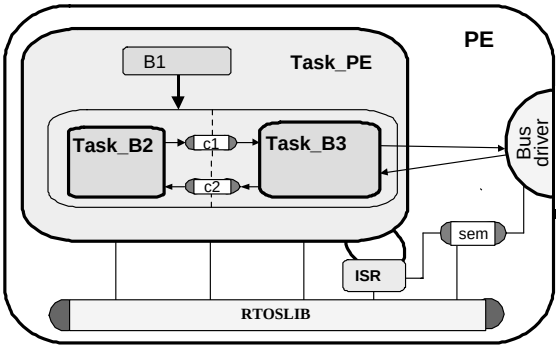
Task refinement converts parallel processes/behaviors in the specification into RTOS-based tasks in a two-step process. In the first step (Figure 7), behaviors are converted into tasks, e.g. behavior *B2* (Figure 7(a)) is converted into *Task_B2* (Figure 7(b)). A method *init* is added for construction of the task. All `waitfor` statements are replaced with RTOS *time_wait* calls to model task execution delays. Finally, the main body of the task is enclosed in a pair of *task_activate* / *task_terminate* calls so that the RTOS kernel can control the task activation and termination.

The second step (Figure 8) involves dynamic creation of child tasks in a parent task. Every `par` statement in the code (Figure 8(a)) is refined to dynamically fork and join child tasks as part of the parent's execution (Figure 8(b)). The *init* methods of the children are called to create the child tasks. Then, *par_start* suspends the calling parent task in the RTOS layer before the children are actually executed in the `par` statement. After the two child tasks finish execution and the `par` exits, *par_end* resumes the execution of the parent task in the RTOS layer.

In summary, for a model with P `par` statements, T tasks, and W `waitfor` statements in task bodies, task refinement requires modification of $W + 2T$ code lines and adding of $4P + 4T$ lines of code.



(a) unscheduled model



(b) architecture model

Figure 6. Model refinement example.

6.2 Synchronization refinement

In the specification model, all synchronization in the application or inside communication channels is implemented using SLDL events. Synchronization refinement replaces all events and event-related primitives with corresponding event handling routines of the RTOS model (Figure 9). All `event` instances are replaced with instances of RTOS events *evt* and `wait` / `notify` statements are replaced with RTOS *event_wait* / *event_notify* calls. For a model with E events and N/W `wait` / `notify` statements, synchronization refinement requires changes in $E+N+W$ lines of code.

After model refinement, both task management and synchronization are implemented using the system calls of the RTOS model. Thus, the dynamic system behavior is completely controlled by the the RTOS model layer.

7. Implementation

The RTOS model library is implemented in 2000 lines of SpecC channel code. Task management in the RTOS model is implemented in a customary manner [C.B99] where tasks transition between different states and a task queue is associated with each state. Task creation (*task_create*) allocates the RTOS task data structure and *task_activate* inserts the task into the ready queue. The *par_start* method suspends the task and calls the scheduler to dispatch another task while *par_end* resumes the calling task's execution by moving the task back into the ready queue.

Event management is implemented by associating additional queues with each event. Event creation (*event_new*) and deletion (*event_del*) allocate and deallocate the corresponding data structures in the RTOS layer. Blocking on an event (*event_wait*) suspends the task and inserts it into the event queue whereas *event_notify* moves all tasks in the event queue back into the ready queue.

In order to model the time-sharing nature of dynamic task scheduling in the RTOS, the execution of tasks needs to serialized according to the chosen scheduling algorithm. The RTOS model ensures that at any given time only one task is running on the underlying SLDL simulation kernel. This is achieved by blocking all but the current task on SLDL events. Whenever task states change inside a RTOS call, the scheduler is invoked and, based on the scheduling algorithm and task priorities, a task from the ready queue is selected and dispatched by releasing its SLDL event. Note that replacing SLDL synchronization primitives with RTOS

```
behavior B2() {
   void main(void) {
    ...
    waitfor(500);
    ...
  }
};
```

(a) specification model

```
behavior task_B2(RTOS os)
implements Init {
  proc me;
  void init(void) {
    me = os.task_create("B2",
         APERIODIC, 0, 500);
  }
  void main(void) {
    os.task_activate(me);
    ...
    os.time_wait(500);
    ...
    os.task_terminate();
  }
};
```

(b) architecture model

Figure 7. Task modeling.

```
...
/* two parallel behaviors */
par
{
   b2.main();
   b3.main();
}
...
```

(a) specification model

```
...
task_b2.init();
task_b3.init();
/* two parallel tasks */
os.par_start();
par
{
   task_b2.main();
   task_b3.main();
}
os.par_end();
...
```

(b) architecture model

Figure 8. Task creation.

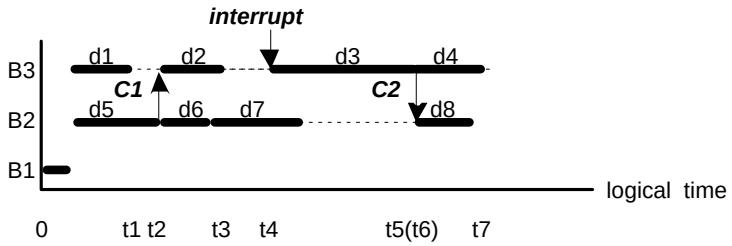
calls is necessary to keep the internal task state of the RTOS model updated.

In high level system models, simulation time advances in discrete steps based on the granularity of `waitfor` statements used to model delays (e.g. at behavior or basic block level). The time-sharing implementation in the RTOS model makes sure that delays of concurrent task are accumulative as required by any model of serialized task execution. However, additionally replacing `waitfor` statements with corresponding RTOS time modeling calls is necessary to accurately model preemption. The *time_wait* method is a wrapper around the `waitfor` statement that allows the RTOS kernel to reschedule and switch tasks whenever time increases, i.e. in between regular RTOS system calls.

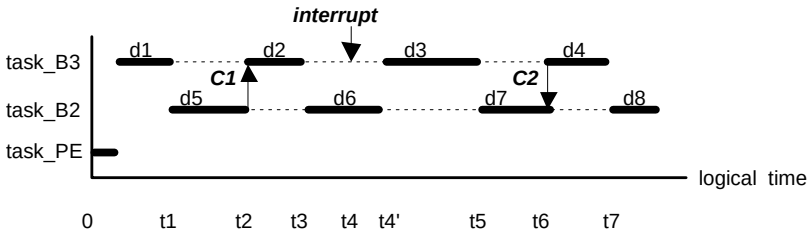
Normally, this would not be an issue since task state changes can not happen outside of RTOS system calls. However, external interrupts can asynchronously trigger task changes in between system calls of the current task in which case proper modeling of preemption is important for the accuracy of the model (e.g. response time results). For example, an interrupt handler can release a semaphore on which a high priority task for processing of the external event is blocked. Note that, given the nature of high level models, the accuracy of preemption results is limited by the granularity of task delay models.

Figure 10 illustrates the behavior of the RTOS model based on simulation results obtained for the example from Figure 6. Figure 10(a) shows the simulation trace of the unscheduled model. Behaviors *B2* and *B3* are executing truly in parallel, i.e. their simulated delays overlap. After executing for time d_1, *B3* waits until it receives a message from *B2* through the channel *c1*. Then it continues executing for time d_2 and waits for data from another PE. *B2* continues for time $(d_6 + d_7)$ and then waits for data from *B3*. At time t_4, an interrupt happens and *B3* receives its data through the bus driver. *B3* executes until it finishes. At time t_5, *B3* sends a message to *B2* through the channel *c2* which wakes up *B2* and both behaviors continue until they finish execution.

Figure 10(b) shows the simulation result of the architecture model for a priority based scheduling. It demonstrates that in the refined model *task_B2* and *task_B3* execute in an interleaved way. Since *task_B3* has the higher priority, it executes unless it is blocked on receiving or sending a message from/to *task_B2* (t_1 through t_2 and t_5 through t_6), waiting for an interrupt(t_3 through t_4), or it finishes (t_7) at which points execution switches to *task_B2*. Note that at time t_4, the interrupt wakes up *task_B3* and *task_B2* is preempted by *task_B3*. However, the actual task switch is delayed until



(a) unscheduled model



(b) architecture model

Figure 10. Simulation trace for model example.

```
channel  c_queue()
{
  event eReady, eAck;
  void send(...)
  { ...
    notify eReady;
    wait(eAck);
    ...
  }
};
```

(a) before

```
channel c_queue(RTOS os) implements Init
{
  evt eRdy, eAck;
  void send(...)
  { ...
    os.event_notify(eRdy);
    os.event_wait(eAck);
    ...
  }
};
```

(b) after

Figure 9. Synchronization refinement.

	unsched.	arch.	impl.
Lines of Code	13,475	15,552	79,096
Execution Time	24.0 s	24.4 s	5 h
Context switches	0	326	326
Transcoding delay	9.7 ms	12.5 ms	11.7 ms

Table 2. Vocoder experimental results.

the end of the discrete time step d_6 in *task_B2* based on the granularity of the task's delay model. In summary, as required by priority based dynamic scheduling, at any time only one task, the ready task with the highest priority, is executing.

8. Experimental Results

We applied the RTOS model to the design of a voice codec for mobile phone applications [G+99]. Table 2 shows the results for this vocoder consisting of two tasks for encoding and decoding running in software. For the implementation model, the model was compiled into assembly code for the Motorola DSP56600 processor and the RTOS model was replaced by a small custom RTOS kernel. The transcoding delay is the latency when running encoder and decoder in back-to-back mode and it is related to response time in switching between encoding and decoding tasks.

The results shows that refinement based on the RTOS

model requires only a minimal effort. Refinement into the architecture model took less than one hour by hand and required changing or adding 104 lines or less than 1% of code. The simulation overhead introduced by the RTOS model is negligible while providing accurate results. Compared to the huge complexity required for the implementation model, the RTOS model enables early and efficient evaluation of dynamic scheduling implementations.

9. Summary and Conclusions

In this paper, we proposed a RTOS model for system level synthesis. To our knowledge, this is the first attempt to model RTOS features at such high abstraction levels integrated into existing languages and methodologies. The model allows the designer to quickly validate the dynamic real time behavior of multi-task systems in the early stage of system design by providing accurate results with minimal overhead. It is designed in such a way that makes the refinement from the unscheduled specification model to the RTOS based architecture model as easy as possible. Using a very small number of system calls, the RTOS model is capable of providing all key features found in any standard RTOS: dynamic task creation/termination, real time scheduling, task synchronization, preemption and interrupt handling. These are features not available in current SLDLs. Currently the RTOS model is written in SpecC because of its simplicity. However, the concepts can be applied to any SLDL (SystemC, Superlog) with support for event handling and modeling of time.

Future work includes development of tools for automatic refinement of the unscheduled model into an architecture model and for software synthesis of target-specific application code linked against off-the-shelf or custom-generated RTOS libraries from the architecture model.

References

[C.B99] Giorgio C.Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1999.

[Cor00] J. Cortadella. Task generation and compile time scheduling for mixed data-control embedded software. In *DAC*, June 2000.

[D+00] Dirk Desmet et al. Operating system based software generation for system-on-chip. In *DAC*, June 2000.

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

[G+01] Lovic Gauthier et al. Automatic generation and targeting of application-specific operating systems and embedded systems software. *IEEE Trans. on CAD*, November 2001.

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

[Spe] SpecC. Available: http://www.specc.org/.

[Sys] SystemC. Available: http://www.systemc.org/.

[T+01] Hiroyuki Tomiyama et al. Modeling fixed-priority preemptive multi-task systems in SpecC. In *SASIMI*, October 2001.

[VxW] VxWorks. Available: http://www.vxworks.com/.

A. Preemptive multi-task system modeling

```
#include "global.sh"
#include <sim.sh>
import "c_semaphore";

behavior task0(RTOSLIB os) implements Init
{
 proc me;
   void init(void)
  {
    //task0 is aperiodic realtime task
    //the priority of task0 is 3
    me=os.task_create("task0",RT_APERIODIC,3,0);
  }

   void main(void)
  {
    os.task_activate(me);

    //code block 1
    os.time_wait(20);
    //code block 2
    os.time_wait(30);

    os.task_terminate();
  }
};

behavior task1(i_semaphore sem1,RTOSLIB os) implements Init
{
  int me;
   void init(void)
  {
    //task1 is aperiodic realtime task
    //the priority of task0 is 2
    me=os.task_create("task1",RT_APERIODIC,2,0);
  }

   void main(void)
  {
    os.task_activate(me);

    //wait for the semaphore
    sem1.acquire();

    //code block 1
    os.time_wait(30);

    //code block 2
    os.time_wait(40);

    os.task_terminate();
```

```
  }
};

behavior task2(i_semaphore sem2,RTOSLIB os) implements Init
{
  int me;
   void init(void)
  {
    //task2 is aperiodic realtime task
    //the priority of task0 is 1
    me=os.task_create("task2",RT_APERIODIC,1,0);
  }

  void main(void)
  {

    os.task_activate(me);

    //code block 1
    os.time_wait(10);

    //wait for the semaphore
    sem2.acquire();

    //code block 2
    os.time_wait(30);

    os.task_terminate();
  }
};

behavior Tasks (i_semaphore s1,i_semaphore s2,RTOSLIB os)
{

    task0 t0(os);
    task1 t1(s1,os);
    task2 t2(s2,os);

    void main(void)
    {
        t0.init();
        t1.init();
        t2.init();
        os.start();
        par{
            t0.main();
            t1.main();
            t2.main();
        }
    }
};

behavior ISR(i_semaphore s1,i_semaphore s2,RTOSLIB os)
```

```
{
    int i=0;
    void main(void)
    {
        if (i ==0) {
            printf("Interruputed␣at␣time␣%s␣!␣\n", time2str(now()));
            s1.release();
            i++;
        }
        else {
            printf("Interruputed␣at␣time␣%s␣!␣\n", time2str(now()));
            s2.release();
            i=0;
        }
        os.ireturn();
    }
};

behavior Processor(in event e1 ,in event e2)
{
const unsigned long N=0;

    SpecC_OS         specc_osapi;

    c_semaphore      s1( ((N)), specc_osapi);
    c_semaphore      s2( ((N)), specc_osapi);


    Tasks       body(s1,s2,specc_osapi);
    ISR         isr(s1, s2,specc_osapi);



    void main(void)
    {
        specc_osapi.init();
        s1.init();
        s2.init();
        try { body.main();}
        interrupt (e1,e2) {isr.main(); } ;

    }
};

behavior Stimulus(out event e1,out event e2)
{
    void main(void)
    {
        waitfor(20);
        notify e1;
        waitfor(30);
        notify e2;
    }
```

```
158 };
159
160 behavior Main()
161 {
162     event e1,e2;
163     Processor  dsp56000(e1,e2);
164     Stimulus   stimulus(e1,e2);
165
166     int main(void)
167     {
168         par {
169             dsp56000.main();
170             stimulus.main();
171         }
172     }
173 };
```