# RTOS Modeling for System Level Design

Andreas Gerstlauer Haobo Yu Daniel D. Gajski

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697, USA
E-mail: {gerstl,haoboy,gajski}@cecs.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 design 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) [1, 2] 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) [3, 4]. 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 [5]. On the other hand, it requires only a minimal modeling effort in terms of refinement and simulation overhead. Our model can be integrated into existing system level design 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. Experimental results are shown in Section 5 and Section 6 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 [8], a method for automatic generation of application-specific operating systems and corresponding application software for a target processor is given. In [6], 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 [10], 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 [7] where they present a high-level model of a RTOS called SoCOS. The main difference is that our RTOS model is written on top of existing SLDLs

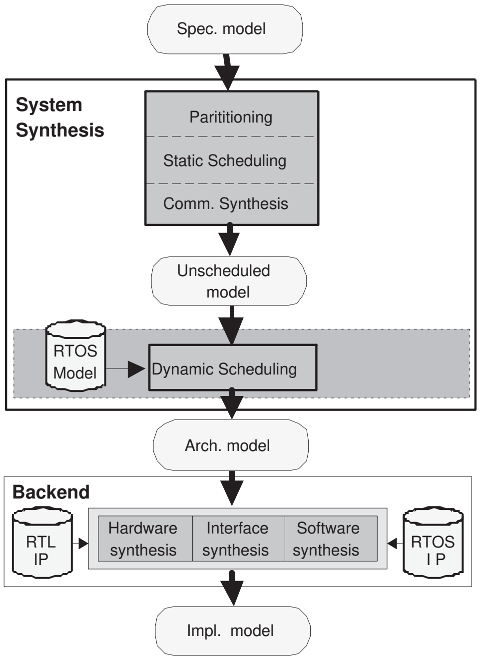



**Figure 1. Design flow.**

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.

## 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 heterogeneous multiprocessor implementation. This refinement is achieved in a stepwise manner through several levels of abstraction. With each step, more implementation detail is introduced through a refined system model. The purpose of high-level, abstract models is the early validation of system properties before their detailed implementation, enabling rapid exploration.

Figure 1 shows a typical system level design flow. The system design process starts with the specification model. It is written by the designer to specifiy and validate the desired system behavior in a purely functional, abstract manner, i.e. free of any unnecessary implementation details. During system design, 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.

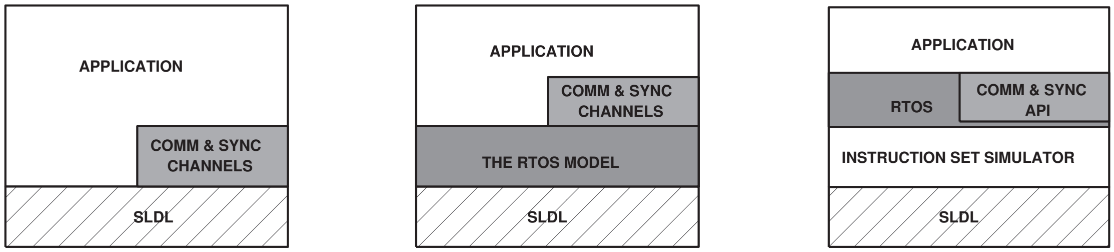
The dynamic scheduling step in Figure 1 refines the unscheduled system model into the final architecture model. In general, for each PE in the system a RTOS model corresponding to the selected scheduling strategy is imported from the library and instantiated in the PE. Processes inside the PEs are converted into tasks with assigned priorities. Synchronization as part of communication between processes is refined into OS-based task synchronization. The resulting architecture model 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 architecture model can be validated through simulation or verification to evaluate different dynamic scheduling approaches (e.g. in terms of timing) as part of system design space exploration.

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. In the process, services of the RTOS model are mapped onto the API of a specific standard or custom RTOS. The code is then compiled into the processor's instruction set and linked against the RTOS libraries to produce the final executable.

## 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




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

**Figure 2. Modeling layers.**

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 [1]. 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. Due to space restrictions, implementation details are limited. For more information, please refer to [11].

### 4.1. RTOS Interface

Figure 4 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

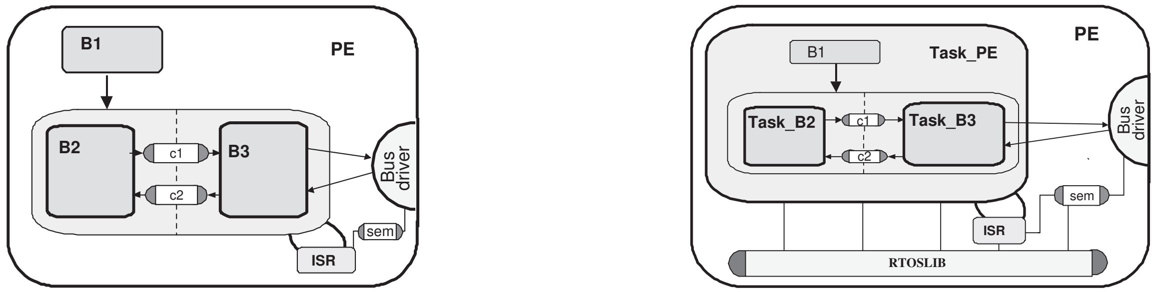
```
interface RTOS {
    /* OS management */
    void init(void);
    void start(int sched_alg);
    void interrupt_return(void);
    /* Task managment */
    proc task_create(char *name, int type,
         sim_time period, sim_time wcet);
    void task_terminate(void);
    void task_sleep(void);
    void task_activate(proc tid);
    void task_endcycle(void);
    void task_kill(proc tid);
    proc par_start(void);
    void par_end(proc p);
    /* Event handling */
    evt  event_new(void);
    void event_del(evt e);
    void event_wait(evt e);
    void event_notify(evt e);
    /* Time modeling */
    void time_wait(sem_time nsec);
};
```

**Figure 4. Interface of the RTOS model.**

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 with a critical deadline and non-periodic real time tasks with a fixed priority. 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




(a) unscheduled model    (b) architecture model

**Figure 3. Model refinement example.**

*time_wait* calls which model task delays in the RTOS while enabling support for modeling of task preemption.

### 4.2. 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 of a single *PE* (Figure 3). In general, the same refinement steps are applied to all the PEs in a multi-processor system. The unscheduled model (Figure 3(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 3(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.

#### 4.2.1 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 5), behaviors are converted into tasks, e.g. behavior *B2* (Figure 5(a)) is converted into *Task_B2* (Figure 5(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 6) involves dynamic creation of child tasks in a parent task. Every `par` statement in the code (Figure 6(a)) is refined to dynamically fork and join child tasks as part of the parent's execution (Figure 6(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.

```
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 5. Task modeling.**

#### 4.2.2 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 7). All `event` instances are replaced with instances of RTOS events *evt* and `wait` / `notify` statements are replaced with RTOS *event_wait* / *event_notify* calls.

```

...

par
{
   b2.main();
   b3.main();
}

...
```

(a) before

```
...
task_b2.init();
task_b3.init();
os.par_start();
par {
   task_b2.main();
   task_b3.main();
}
os.par_end();
...
```

(b) after

**Figure 6. Task creation.**

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

(a) before

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

(b) after

**Figure 7. Synchronization refinement.**

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.

### 4.3. Implementation

The RTOS model library is implemented in 2000 lines of SpecC channel code [11]. Task management in the RTOS model is implemented in a customary manner [5] 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.

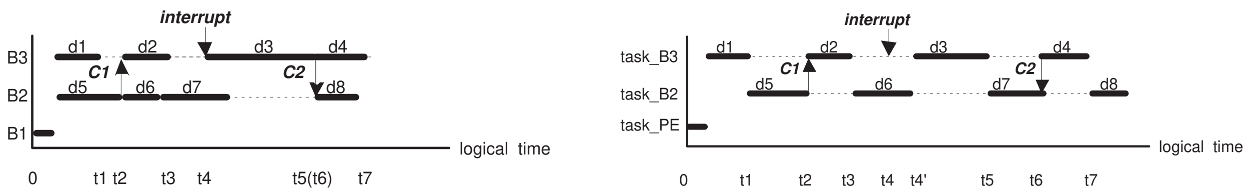
In order to model the time-sharing nature of dynamic task scheduling in the RTOS, the execution of tasks needs to serialized according to the chosen scheduling algorithm. The RTOS model ensures that at any given time only one task is running on the underlying SLDL simulation kernel. This is achieved 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 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 8 illustrates the behavior of the RTOS model based on simulation results obtained for the example from Figure 3. Figure 8(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 8(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 8. Simulation trace for model example.**

| | 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 1. 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.

## 5. Experimental Results

We applied the RTOS model to the design of a voice codec for mobile phone applications [9]. Table 1 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, described in more detail in [9]. 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 show that refinement based on the RTOS model requires only a minimal effort. Refinement into the architecture model was done by converting relevant SpecC statements into RTOS interface calls following the steps described in Section 4.2. For this example, manual refinement took less than one hour and required changing or adding 104 lines or less than 1% of code. Moreover, we have developed a tool that performs the refinement of unscheduled specification models into RTOS-based architecture models automatically.

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.

## 6. Summary and Conclusions

In this paper, we proposed a RTOS model for system level design. 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. Using a minimal number of system calls, the model provides all key features found in any standard RTOS but not available in current SLDLs. Based on this RTOS model, refinement of system models to introduce dynamic scheduling is easy and can be done automatically. 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 implementing the RTOS interface for a range of custom and commercial RTOS targets, including the development of tools for software synthesis from the architecture model down to target-specific application code linked against the target RTOS libraries.

## References


[1] SpecC[online]. Available: http://www.specc.org/.

[2] SystemC[online]. Available: http://www.systemc.org/.

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

[4] VxWorks[online]. Available: http://www.vxworks.com/.

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

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

[7] D. Desmet et al. Operating system based software generation for system-on-chip. In *DAC*, Jun 2000.

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

[9] A. Gerstlauer et al. Design of a GSM Vocoder using SpecC Methodology. Technical Report ICS-TR-99-11, UCI, Feb 1999.

[10] H. Tomiyama et al. Modeling fixed-priority preemptive multi-task systems in SpecC. In *SASIMI*, Oct 2001.

[11] H. Yu et al. RTOS Modeling in System Level Synthesis. Technical Report CECS-TR-02-25, UCI, Aug 2002.