An Ideal API for RTOS Modeling at the System Abstraction Level

Adnan Shaout, Khalid Mattar, and A. Elkateeb,
The University of Michigan-Dearborn
The Electrical and Computer Engineering Department
shaout@umich.edu

Abstract—In System Level Design, specification languages, especially System Level Design Languages (SLDL), are used to build high level models to allow fast design space exploration, and to assist designers in exploring alternatives early in the design process. Current SLDL languages lack built-in support for modeling RTOS at the System Level, and specific RTOS implementation cannot be used directly in models at the System Level. In this paper, we define and provide the primitives for an ideal API for a generic RTOS model to be used on top of existing SLDL. The model is based on the key features provided by a typical RTOS, and is generic such that it can be used with any SLDL. Using the API defined in this paper, we describe the refinement of the RTOS model so that the model can be integrated into the System Level co-design process.

I. INTRODUCTION

A. System Level Design

System Level Design is concerned with addressing the challenges encountered in designing heterogeneous embedded systems. In System Level Design, complexities are managed by starting the design process at the highest level of abstraction (System Level) and utilizing automated design methodologies to enable step-wise refinements during the design process [15]. Designing at a higher level of abstraction reduces the number of components with which the designer has to deal with, and thus increasing design productivity. System Level Design maps a high-level abstract specification model of an entire system onto target architecture [16].

Hardware/Software co-design (also referred to system synthesis) is a System Level Design top-down approach that starts with system behavior, and generates the architecture from the behavior. It is performed by gradually adding implementation details to the design. Generally, Hardware / Software co-design consists of the following activities: (1) specification and modeling, (2) design and refinement and (3) validation [17].

B. System Level Design Languages (SLDL) and RTOS modeling

Specification is the first step in the co-design process where the system behavior, at the system level, is captured. Specification languages, specifically System Level Design Languages (SLDL), can model and simulate concurrent behaviors, and have primitives to allow modeling of synchronization and time delays. In addition, SLDL have primitives to control execution of a process via events. However, most SLDL lack primitives for controlling one process from another and the ability to model the affect of task serialization and preemption on a processor when using RTOS. As shown in table 1, current specification languages lack features needed to support RTOS modeling.

C. Approaches for RTOS modeling and simulation

Le Moigne, et al. in [5] describe three approaches for supporting RTOS modeling at the system level.

1) Using a model and a simulator dedicated to a specific RTOS. This approach provides precise results since the model is designed for a specific RTOS. The drawback of this approach is that it limits design space exploration to the selected RTOS.

2) Using a generic abstract RTOS model with its own simulation engine. This approach addresses the issue of limiting the design space exploration by using a generic abstract RTOS model. However, since this approach uses its own simulation engine, it requires synchronizing RTOS simulation with hardware simulation which uses the simulation engine provided the SLDL.

3) Using a generic abstract RTOS model on top of existing SLDL. This approach is not limited to specific RTOS since it provides a generic model. In addition, it utilizes existing SLDL primitives and simulation engines. Thus, avoiding potential synchronization difficulties. Approach # 3, using a RTOS model on top of existing SLDL, is the main focus of this paper.

D. Outline of the paper

In section 2, we compare five RTOS modeling techniques that have been proposed in the literature. In section 3, we describe the requirements for designing an RTOS model on top of a SLDL. In section 4, we define an ideal API for a
generic RTOS model derived from the key features provided by a typical RTOS. Section 5 describes the RTOS model using the API we defined.

In section 6, we describe in the steps for refining the RTOS model so that it can be used in as part of the co-design process. Lastly, section 7 concludes the paper.

II. COMPARISON OF RTOS MODELING TECHNIQUES

Several RTOS modeling techniques using System Level Design Languages (SLDL) have been proposed in the literature to enable assessing the dynamic real time behavior of embedded systems at the System Level. The majority of RTOS modeling techniques in the literature are based on the SystemC SLDL. Only two techniques were found based on SpecC. Table 2 provides a comparison of five RTOS modeling techniques.

<table>
<thead>
<tr>
<th>Task</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Comparison of RTOS modeling techniques

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RIOS model on top of SLDL</td>
<td>SpecC</td>
<td>SpecC behaviors &amp; Task Control Block (TCB) to hold task characteristics</td>
<td>Supports tasks state transition, but the different states are not mentioned in the paper</td>
<td>Estimated at the block level. Delay primitives inserted at the end of each block</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>RIOS model on top of SLDL</td>
<td>SystemC</td>
<td>SystemC threads that use SystemC events</td>
<td>Ready, running &amp; waiting</td>
<td>Estimated using a delay procedure based on the work done by [14]</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>RIOS model on top of SLDL</td>
<td>SystemC</td>
<td>SystemC threads that use SystemC events</td>
<td>Idle, ready, executing &amp; preempted</td>
<td>Not specified</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Using native SpecC exception handling &amp; concurrency constructs</td>
<td>SpecC</td>
<td>SpecC behaviors: Tasks are modeled as running in parallel using the &quot;par&quot; specC construct systemC thread that runs continuously</td>
<td>No explicit states are defined</td>
<td>Not specified</td>
<td>Ready, running, preempted, postponed &amp; sleep</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>RTOS model on top of SLDL</td>
<td>SystemC</td>
<td>SystemC threads &amp; Task Control Block (TCB) to hold task characteristics</td>
<td>Uses a stochastic timing model based on the Gumbel probability density function</td>
<td>Uses the appropriate function based on the scheduling policy selected</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

*As ac_mExports in SystemC 2.1*
III. DESIGN CRITERIA FOR RTOS MODELING AT SYSTEM LEVEL

As mentioned earlier, the objective of modeling RTOS at the System Level is to accurately model the real time and dynamic behavior of a system, and to identify any timing constraints early in the design process. In this section we describe the major criteria / requirements for designing an abstract RTOS model on top of existing SLDL [5, 6].

1) Support all key features of a typical RTOS. A RTOS model must provide features such as:
   a) OS management services. Supports initialization of the RTOS and its data structures.
   b) Scheduling service
      i) Support several scheduling policies
   c) Task management services. Supports both periodic and non-periodic tasks
   d) Tasks synchronization services
   e) Support modeling of interrupts
   f) Support modeling of preemption
   g) Support modeling of time

2) Requires minimal modeling efforts in terms of modeling refinement and simulation overhead.

3) The RTOS model should be generic, and not limited to a specific SLDL so that it can be used on top of different System Level Design Languages (SLDL).

4) The RTOS model should be designed such that it can easily integrate into existing System Level co-design methodologies.

5) The RTOS model should be independent of any specific RTOS implementation.

IV. AN IDEAL API FOR A GENERIC RTOS MODEL

In this section we define and describe an API of a generic RTOS model. The API defines key services found in typical RTOS [6-8]. The API for the RTOS model defines the interface between the system application and the RTOS model.

A. OS management services

The main functionality of the OS management services is to initialize the RTOS data structures and to start the scheduler. Table 3 provides the API for the OS management services.

B. Task management services

The purpose of the task management services is to provide a mechanism so that the system application can be specified as a set of tasks. In addition, the API provides services to allow preemption. Tables 4 and 5 provide details about the API for the task management services.
C. Task synchronization services

The task synchronization services provide means to synchronize concurrent task, and to enable inter-task communication. Table 6 shows the API for the task synchronization primitives

<table>
<thead>
<tr>
<th>Description</th>
<th>Parameter</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>void end_periodic_task(task_id)</td>
<td>task_id</td>
<td>None</td>
</tr>
<tr>
<td>void reset_periodic_task(task_id)</td>
<td>task_id</td>
<td>None</td>
</tr>
<tr>
<td>void end_task(task_id)</td>
<td>task_id</td>
<td>None</td>
</tr>
<tr>
<td>void kill_task(task_id)</td>
<td>task_id</td>
<td>None</td>
</tr>
</tbody>
</table>

D. Time services

<table>
<thead>
<tr>
<th>Description</th>
<th>Parameter</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>void rtos_delay(time)</td>
<td>Time</td>
<td>None</td>
</tr>
</tbody>
</table>

V. THE RTOS MODEL

A. Task model

A Task must be modeled such that it holds all information needed for task execution, and it allows preemption and resume by the scheduler. Tasks are modeled using the Task Control Block (TCB) model [9]. The TCB model uses a circular linked list for the system tasks. The TCB structure for each task holds information such as task_id, state, priority, period, Worst Case Execution Time (wcet) and Best Case Execution Time (bcet). The create_task() primitive is used to create a task and populate its TCB. To support preemption, the model provides the suspend_task(), and resume_task() primitives. The model provides two primitives to support periodic tasks. When a periodic task completes its execution, it invokes the end_periodic_task() primitive to notify the scheduler that it has finished execution. For non-periodic tasks the end_task() primitive is used to notify the scheduler when a task completes its execution.

B. Scheduling model

The scheduler is the heart of the RTOS. Its main purpose is to select the next task to run from the list of ready tasks according to a scheduling policy.

To accurately assess the influence of scheduling on the system's real time performance at the System Level, both the scheduler behavior and its timing properties [5] need to be modeled. The scheduler behavior is characterized by its scheduling policy. The scheduler timing properties are characterized by the scheduler latency (how long it takes to select a ready task to execute), and by the context switching duration (for both context-save and context-load operations). The RTOS model described in this paper supports modeling of both properties. The scheduling policy is selected by passing a parameter to the start_scheduler() primitive. The timing properties of the scheduler are defined in the RTOS configuration file.

In the RTOS model, tasks can be in one of four states: dormant, ready, running or suspended. When a task is created using the create_task() primitive, its default state is the dormant state. A task in the dormant state goes into the ready state when (1) a periodic task enters a new execution time or (2) a non periodic task has all required data to execute. A ready task goes to the running state once selected by the scheduler based on the scheduling policy used in the RTOS model. When in the running state, a task goes to the ready state when (1) preempted by a higher priority task or (2) the time slice allocated to the task expires. When a task in the running state requires data or a resource that is not available, it goes to the suspended state. When the resource becomes available, the suspended task goes to the ready state so that it can be scheduled. When a periodic task finishes its execution, it goes to the dormant state. A periodic task in the dormant state goes to the ready state when its release time is up.

C. Timing and Preemption model

In System Level models, the execution time of a task is more important than the exact functionality of the task. Execution times can be estimated at the statement level or the behavior level [7]. At the behavior level execution times can be estimated based on the task's average and/or worst case execution timings. Most SLDL provide primitives to model timings and delay. The RTOS model described in this paper provides the rtos_delay() primitive for time modeling. To ensure synchronization between the RTOS model and SLDL simulation engine, the RTOS rtos_delay() primitive is implemented as a wrapper around the SLDL delay primitive [7]. The execution time of a task is modeled by passing the estimated execution time as a parameter to the RTOS rtos_delay() primitive. Typically, the code of a task is divided into atomic blocks.
(i.e. blocks that execute until completion), and the RTOS delay primitive is inserted at the end of an atomic block to denote the execution time of an atomic block [6]. Preemption is based on the timing model build using the RTOS rtos_delay() primitive such that preemption occurs during the time delay. The suspend() and resume() RTOS primitives are used to model Interrupt preemption.

D. Synchronization model

Synchronization of tasks in a multitasking system is critical to allow data and resource sharing, and ensure data integrity. The purpose of the RTOS synchronization model is to provide services to synchronize concurrent tasks. The RTOS model uses two primitives: rtos_wait() and rtos_notify() for synchronization.

The rtos_wait() primitive blocks a task until another task invoke the rtos_notify() primitive. When a task is blocked waiting for data it goes to the suspended state. The rtos_notify() primitive signals a suspended task for the availability of data. At such point, the task goes from the suspended state to the ready state.

VI. RTOS MODELING AND CO-DESIGN FLOW

During the design & refinement stage of the co-design process, the specification model is partitioned into hardware and software components. Yu et al. [10] refers to the result of the hardware/Software partition step as the unscheduled Transaction Level Model (TLM). The unscheduled TLM model is represented using primitives provided by the SLDL supported in the co-design system. Tasks (processes) mapped to the same processor need to be scheduled either statically or dynamically. In order to simulate the behavior of dynamically scheduled tasks and assess the dynamic real time behavior of the system, a high level RTOS model need to be incorporated into the unscheduled TLM model. To do so, the unscheduled TLM model must be refined such that it uses the RTOS-based primitives instead of / along with the SLDL primitives [10]. This is referred to as model refinement. Figure 1 shows model refinement incorporated into the co-design flow.

A. Model Refinement

The purpose of the model refinement is to transform the unscheduled TLM model into a scheduled RTOS-based model. The output of model refinement is a scheduled model where each processor runs multiple tasks on top an RTOS model. Based on the work done in [10] and [7], the four main steps for model refinement are described below. These steps will be described below using the RTOS API defined in this paper.

4) RTOS model instantiation includes the following activities

a) An RTOS model is selected from the RTOS library based on the scheduling algorithm to be simulated

b) For each processor in the unscheduled TLM model, create an RTOS run time environment, which initializes the internal data structures of the RTOS model

5) Task refinement transforms all behavior tasks in the TLM model to RTOS based tasks. Task refinement consists of the following steps:

a) Step 1: Flatten all hierarchal tasks and parallel processes mapped to each processor by creating new tasks and processes.

Step 2: For all tasks in the TLM model, insert the RTGS primitive for task creation along with the parameters that characterize the task (i.e. priority, period, ...). Using the API defined in this paper, this step inserts the task_create() primitive.

c) Step 3: For all tasks in the TLM model, insert the RTOS primitive responsible for informing the scheduler that the task has completed its execution. Using our API, this step inserts the end_periodic_task() primitive for periodic tasks, and the end_task() primitive for non-periodic tasks.

6) Synchronization refinement wraps all SLDL synchronization primitives in the TLM model with RTOS synchronization primitives. Using the API defined in this paper, synchronization refinement wraps SLDL synchronization primitives with
rtos_wait() and rtos_notify() primitives. That way the RTOS can intercept any SLDL synchronization primitives, and switch tasks as appropriate.

7) **Preemption Refinement** replaces all SLDL time delay primitives with the corresponding RTOS primitives (i.e. rtos_delay() primitive in the API defined in this paper). As stated earlier, the task code is divided into atomic blocks, and execution time for each atomic block is estimated by inserting SLDL time delay primitives at the end of each block. Based on this scheme, tasks can only be preempted inside the SLDL time delay primitive. Therefore, preemption refinement is necessary to accurately model preemption.

VII. CONCLUSION

Designing at the System Level abstraction layer reduces the complexities encountered in designing heterogeneous embedded systems. Real Time Operating System modeling at the System Level abstraction layer is critical since it allows the designer to assess the real time characteristics and dynamic behavior of the system early in the design process before committing to a specific RTOS. System Design Languages (SLDL) are gaining wide usage for specifying and modeling heterogeneous embedded systems at the System Level. However, current SLDL such as SystemC and SpecC lack built-in support for RTOS modeling at the System Level. Several RTOS modeling techniques on top SLDL have been proposed in the research literature. Only two techniques were found that are based on SpecC, the rest are based on SystemC. In this paper, we compared five RTOS modeling techniques: two based on SpecC and three based on SystemC. All of the SystemC based techniques use the approach of defining RTOS interface on top SystemC. One of SpecC techniques uses the native SpecC constructs to provide RTOS modeling capabilities, while the other uses RTOS interface on top of SystemC. A comparison of RTOS modeling techniques is presented in table 2. We described in this paper the major requirements for supporting RTOS modeling at the System Level, and defined an ideal API to support RTOS modeling on top of SLDL. The proposed API is based on services provided by a typical RTOS, and is generic such that it can be used with any SLDL.

REFERENCES


