Real-Time Operating System Services for Realistic SystemC Simulation Models of Embedded Systems

Abstract

The design process of embedded systems moves currently towards higher levels of abstraction. As a consequence, a need arises for an early and realistic assessment of system level design decisions. In order to provide the early access, an automatic generation of its simulation models is advocated. While Verilog and VHDL are being extended in order to improve their system-level capabilities, SystemC community is intensively extending SystemC to software modeling features. In this sense, SystemC support for a realistic assessment of embedded system in the process of develop-validate-and-test its software is advocated. This step is done before any decisions with respect to processor and operating system has been finalized, as well as before an executable platform model and prototype board becomes available. Moreover, while execution properties of embedded software processes, which more and more dominate the functionality of embedded systems, can considerably vary both due to processor pipeline and cache structures, the chosen scheduling policy influences distinctly the execution properties. Unfortunately, the current version of SystemC is still lack of that software modeling support. Therefore, the modeling capability of SystemC is being extended in this paper by generic real-time operating system services providing more realistic software modeling features. The features cover also modelling dynamic process creation, process control, pre-emption, process prioritization, inter-process communication, and synchronization, as well as static and dynamic scheduling. Design alternatives can thus be easily validated and different scheduling policies can be explored. The figures of merit of the proposed approach are demonstrated by means of the embedded information processing of a mobile robot.

1. Introduction

The application of embedded systems in human life are rapidly spreading in many fields – from small and simple devices used in kitchens up to complex systems, which are part of highly dependable and safety-critical systems such as nuclear reactors and aircraft flight control systems. Due to the growing complexity of such systems, the design process considering software and hardware in concert (co-design) is constantly moved towards higher levels of abstraction. The proposed co-design flow starts from an abstract specification and results a final implementation as illustrated in Figure 1. The system-level synthesis step constitutes first design decisions for the fundamental synthesis problems. Implementation alternatives should be validated as early as possible in the design flow. Therefore, a realistic system-level simulation support is mandatory for a successful design methodology. The tight ‘time-to-market’ window imposes an automatic generation of this simulation models based on the specification and first design decisions.

Especially safe-critical systems, but many other embedded systems too, belong to the class of real-time (RT) systems. As defined in literature, e.g., [1][2][3], an RT system is characterized by the fact that its overall correctness not only depends on the correctness of its functionality, but also on the timing of the response of the corresponding functionality. The RT behavior of software parts is managed by a real-time operating system (RTOS). The RTOS provides particular services required for composing the RT system that need to be considered during software design and analysis. The services provided by the operating system are mainly intended to support the implementation of the required multi-tasking or job concurrency by applying the chosen scheduling policy. Different policies can be explored for RT execution of embedded software including static event-driven, priority based, and time-triggered scheduling. These features are of
utmost interest, because the execution times of software tasks can vary considerably due to modern processor architectures, which heavily exploit pipelines and caches. Thus, it is obvious that some constructs or services are necessary, which are to be used for process creation or destruction and for communication and synchronization purposes. Our model of RTOS simulation presented in this paper provides such generic services and allows to easily explore and validate both embedded software design alternatives and different tasks scheduling policies. SystemC 2.0[4][5], a system level description language based on C++, was selected as the underlying modeling language and as implementation means for the executable computational models. The fundamental C++ class library of SystemC provides a cycle-based simulation kernel as well as all the necessary constructs required to create a cycle-accurate system model. Just one SystemC model is necessary to specify both the hardware and software parts of an embedded system. When gradual refinement of the specification is performed, then an executable model will be available at the any point in time. However, SystemC must also able to provide realistic assessment for develop-validate-and-test embedded software before any decisions with respect to processor and operating system has been finalized, as well as before an executable platform model and prototype board becomes available. The SystemC community still intends to extend SystemC to software modeling features. Unfortunately, current version of SystemC is still lack support of those facilities. Therefore, the modeling capabilities of SystemC 2.0, is being extended in this paper by the generic services of RTOS providing constructs to model software decomposition, dynamic process creation and deletion, process control, pre-emption, static/dynamic process prioritization, static/dynamic scheduling and inter-process/tasks communication and synchronization where all these thing still do not completely work well there. Embedded software (architecture) design alternatives can thus be easily explored and validated earlier.

So far, the proposed approach considers both a stochastic timing model and the effects of different scheduling policies, which can easily be observed and validated from the execution of the associated SystemC models. The figures of merit of the proposed approach are demonstrated by means of the embedded information processing of a mobile robot.

The remainder of this paper is organized as follows. The next section discusses some other work related to the methods presented in this paper. Section 3 outlines the proposed design flow and system modeling methods, where the interaction of the RTOS model along with its services required for inter tasks communication refinement in an embedded system software modeling is detailed. In Section 4 the extensions of SystemC with a generic RT operating system built on top of it is discussed. Section 5 is dedicated to the proposed system level simulation techniques. Scheduling policies, which can be used for schedulability analysis, is detailed next. Then, Section 7 summarizes the results produced by means of the application example presented in Section 6. Finally, a conclusion is given and some open questions and future work is discussed.

2. Related Work

The basis of an appropriate design process is the specification model. In contrast to control dominated specification models, such as Statecharts or SDL [6][7], task graphs are a general and powerful specification concept for data-flow dominated embedded systems at system-level. This holds especially for concurrent software functions. However, most embedded systems unveil as least a few control-flow depending behaviors. Therefore, extensions to control-flow modeling of task graphs are introduced in [8][9][10][11]. Different approaches to system-level synthesis based on task graphs are presented in [12][13][14], which address scheduling, allocation and binding solutions based on different optimization heuristics. However, no validation or simulation concepts are introduced at all, while SystemC[4] provides this by the concept of executable specifications. A first approach for automatic generation of SystemC simulation models based on an abstract specification is presented in [10].

Modeling of embedded systems at transaction level and a definition of this abstraction level can be found in [15][16]. Other concepts for modeling RTOS especially in SpecC can be found in [17] [18] where [19] propose a targeting application-specific operating systems. The main difference to the presented work is that we are working on system-level and on SystemC. Moreover, current version of SystemC is still lack of support such a generic RTOS model and other facilities required to model software especially for real-time embedded system. A similar idea was presented in [20]. Furthermore, we are more focus on providing an environment for a highly flexible schedulability analysis, by providing not only basic RTOS services but also implement the model of scheduler for static and dynamic scheduling as well as processes control,
preemption and dynamic priority assignment. Another important aspect for realistic simulations is the problem of run-time estimation of functional blocks. These difficulties arise from modern processors including pipelining and cache hierarchies. Behavioral intervals for the non-functional parameters are discussed in [21]. In addition to pure intervals, [22] considers a stochastic distribution of the execution times for schedulability analysis. In [23] the Gumbel distribution is introduced as a realistic metric to assess software execution times. In addition, different scheduling policies are explored for RT execution of the embedded software. Rate monotonic scheduling (RMS) [24][25] algorithm, dynamic scheduling such as earlier deadline first scheduling (EDF) [24], and static approaches are being covered by the proposed RTOS services. In case of rate monotonic the priority of each task will be re-assigned according its period before the simulation is being executed. Rate monotonic scheduling uses task priority values to enhance the right of execution to those tasks, which feature a higher invocation rate.

3. Design Flow and System Modeling

![Figure 1 : Proposed codesign flow for embedded systems](image)

The general design process starts from an abstract system specification. For this purposes a task graph based specification model is advocated. Task graphs capture the intended behavior by means of functional blocks and data-dependencies between them. At system-level a task consists of complex functions, e.g., described by algorithms. These tasks represent the computational units that must be executed either sequentially or concurrently thus defining a number of computational jobs to be performed within an embedded system. This basic approach of using task graphs can be easily adopted to more complex specification models such as etG [10] or hCDM [26], which capture to some extent control-flow information, too. After specifying the system with such an abstract model the initial assignment and optimization steps concerning the fundamental synthesis problems – allocation, scheduling, and binding – are to be performed next. Based on the results of these initial design decisions an executable model is highly desirable, because it allows an early assessment of the timed system behavior by simulation before implementing the core functionality.

The system level synthesis process results in a partitioning of the system. Certain tasks will be implemented as hardware modules, where others are best candidate for software implementations. Some already available hardware IP and third-party software code for re-use may inspirit this decision. The next step in the design flow is aimed to a translation of the result from the system level synthesis processes into a corresponding SystemC model. In this model each task is assigned either to hardware or software. Each processing element running software code is implemented as a SystemC module. Such models can be automatically generate and afterwards refined by support of proposed RTOS services into fully compileable and synthesisable code. Design alternatives of both general architecture of system under design and its embedded software (architecture) can thus be early explored and validated.

The software tasks, which are allocated into some processing elements of the embedded system (ES) architecture, are modeled according to the functionality that is provided by the single SystemC module. The module (Main PE module in Figure 2) is an abstraction of single processor, so all software tasks inside this module will be executed in a pseudo-parallel manner. As illustrated in Figure 2(a), due to system timing requirements and to specified quality of services, in general the tasks must be ordered in time domain
according to a certain scheduling algorithm. This means that an RTOS is required to provide such services for the software execution. Figure 2(b) illustrates, where the model of a generic RTOS is located within the SystemC model of a processing element. The example visualized in Figure 2 contains three software tasks \( P_3, P_4 \) and \( P_5 \) surrounded by four hardware tasks. Figure 2(b) also illustrates that the RT operating system model along with its services, which support the inter-tasks communication refinement. The RT operating system is responsible to provide a viable means such that tasks can be synchronized in order to be able to communicate with each other. The synchronization between software tasks needs some specific constructs as detailed in Section 4. In case of communication requirements between a hardware task and a software task, the RTOS provides the driver library of the hardware accessed by software tasks. The operating system has in addition to offer some kind of bus driver services such that software tasks can communicate with other tasks that are running on another processing element. This service is mandatory for the support of distributed embedded systems, which involve several processing elements.

\[
g(t) = \exp \left( -\left( e^{-\frac{t-\mu}{\sigma}} - \frac{t-\mu}{\sigma} \right) \right) \frac{\mu}{\sigma}
\]

(a) The trend of experimental task execution times (b) Task execution times model by Gumbel density function

**Figure 3 : Basic model of task execution times distribution**

Based on this model of task execution timing the effects of different scheduling policies may be assessed by means of the introduced RTOS model. The implementation of scheduling policies in our generic RTOS model covers both static and dynamic scheduling policies:

- Static event-driven scheduling
- Static time-triggered scheduling
- Static priority based scheduling
- Preemptive scheduling.
4. Capability Extension to SystemC

While Verilog and VHDL are being extended in order to improve their system-level capability, the community of SystemC intends to expand the usage of SystemC into software modeling. The usage models of SystemC in this sense is outlined in [27]:

(a) Design/analyze system architecture
i. HW/SW interface (partitioning, memory, bus access, etc.)
ii. SW architecture (scheduling, preemption, priorities, communication, synchronization, etc.)

(b) Develop and validate/test SW before
i. Decision w.r.t. processor and RTOS have been finalized
ii. Executable platform models (ISS, peripherals) are available
iii. Prototype boards are available

However, [27] also addresses a number of SystemC software modeling capabilities that remains not work in the current available version [4] of SystemC includes:

- Dynamic process creation
- Process control (suspend, resume, kill, etc.)
- Scheduler modeling
- Preemption

The generic RTOS services presented in the paper provide those capabilities to overcome the weakness of SystemC for software modeling in the system design and analysis. Furthermore, the RTOS model also provides the capabilities for scheduling assessment both static (event-driven, time-triggered, priority based ordering) and dynamic scheduling (RMS, EDF). In order to support software modeling in the case software coding toward a specific RTOS API, the generic RTOS model also provides a POSIX-like interface along with process ID (PID) model.

```
class sc_rtos_basic_if : virtual public sc_interface {
    /* OS & scheduling services*/
    virtual void ini() = 0;
    virtual void start(rt_scheduling sch) = 0;
    /* Task creation & termination services */
    virtual rt_task* create_task (sc_module_name name,
        void (__closure *mainfunc)(),
        rt_priority priority,
        rt_time first_ready_time,
        rt_periodicity periodicity,
        rt_time period,
        sc_time_unit time_unit ) = 0;
    virtual void kill(rt_task *t) = 0;
    virtual void abort(rt_task *t) = 0;
    /* Task syncronization services*/
    virtual void resume(rt_task *t) = 0;
    virtual void postpone(rt_task *t) = 0;
    virtual void sleep(double v, sc_time_unit tu) = 0;
    virtual void wakeup(rt_task *t) = 0;
    /* Atomic function execution time modeling*/
    virtual void await(rt_time v, sc_time_unit tu) = 0;
};
```

(a) Enhancement of SystemC by generic RTOS
(b) Basic interface of the RTOS module

Figure 4 : Generic RTOS model and provided basic services

4.1. Architecture of Generic Services

Many embedded applications for safety critical systems involve the use of RT operating systems as an inherent part of their software architecture. For a realistic system-level simulation it is thus mandatory to capture the fundamental features of an operating system. Therefore, SystemC is extended by the new library detailed in the sequel that provides basic scheduling and inter-process communication services. The generic RT operating system is placed on top of SystemC 2.0 as illustrated in Figure 4(a). It provides most of the services commonly available in usual RT operating systems, especially those services, which are required for task creation, for task management and for inter-task synchronizaion. By means of these extensions to
SystemC, the overall modeling capabilities become more suited to design space exploration as well as for early performance analysis of design decisions. The model of the RTOS is implemented in the core SystemC as a module that encapsulates all threads instances, the scheduler, as well as the simulation time and the scheduling policy. The operating system is thus responsible to provide associated services such as dynamic task creation, destruction, and management as depicted in Figure 2(b). Because the software tasks have to be scheduled according to the selected schedule (time table, tasks order) or certain scheduling algorithm, additional services for this purpose are required. Model of task priority, preemption, and inter-task synchronization in the RTOS support implementation the task scheduling as shown by the code fragments of Figure 4(b).

The scheduling service provides a way to validate task-scheduling decisions taken for created tasks. The algorithms of scheduling both static and dynamic implemented in this RTOS scheduler are based on a priority approach as detailed in Section 4.4. The task priority is determined by the scheduler at the beginning of the simulation run in case of static scheduling is selected. For example, if RMS is chosen, then priority of each task will be re-assigned according its rate (consequently, according to its period) before the simulation is executed. In case of dynamic scheduling such as EDF is selected, then the priority of each task will be re-assigned dynamically during simulation time based on both task ready time and its period. If the task is not periodic, then the deadline has to be specified when it is created.

![Model of tasks state transitions diagram managed by the RTOS model](image)

The notion of a task or process (used interchangeable in this paper) is the underlying concept to model concurrency in the abstract RTOS. It is implemented by means of the sc_thread macro in SystemC and encapsulates the thread as a task object. Thus, the RTOS model keeps the C++ object-oriented concept that is inherent in the language where SystemC is built. Task identification is represented by the task name and is mapped to a module name (sc_module) as an implication of SystemC constructs, where modules implementing the containers are the building blocks of the SystemC model architecture. The task name as well as the main code (mainfunc, see Figure 4b) that embodies the functionality and the entry point of the task has to be passed to the operating system method when a task is being created. At the same time, some parameters such as task priority, ready time, arrival periodicity status, arrival period, and simulation time unit have to be passed as well to characterize the instance of task.

In order to support process control during the simulation run, some additional services are required in the RTOS model. These services embody the postpone/resume and the sleep/wake-up mechanisms. The allowed actions regarding synchronization are shown in table of Figure 5(b) with respect to the state of suspended tasks when the active task tries to perform a synchronization action by calling the RTOS services. The feasible task state transition within the RT operating system model follows the state transitions diagram (a) Model of tasks state transitions diagram managed by the RTOS model (b) Synchronization actions possible between active tasks by means of the RTOS services

**Figure 5 : State transition due to task synchronization action**

### 4.2. Modeling Concurrency and Process Control

The notion of a task or process (used interchangeable in this paper) is the underlying concept to model concurrency in the abstract RTOS. It is implemented by means of the sc_thread macro in SystemC and encapsulates the thread as a task object. Thus, the RTOS model keeps the C++ object-oriented concept that is inherent in the language where SystemC is built. Task identification is represented by the task name and is mapped to a module name (sc_module) as an implication of SystemC constructs, where modules implementing the containers are the building blocks of the SystemC model architecture. The task name as well as the main code (mainfunc, see Figure 4b) that embodies the functionality and the entry point of the task has to be passed to the operating system method when a task is being created. At the same time, some parameters such as task priority, ready time, arrival periodicity status, arrival period, and simulation time unit have to be passed as well to characterize the instance of task.

In order to support process control during the simulation run, some additional services are required in the RTOS model. These services embody the postpone/resume and the sleep/wake-up mechanisms. The allowed actions regarding synchronization are shown in table of Figure 5(b) with respect to the state of suspended tasks when the active task tries to perform a synchronization action by calling the RTOS services. The feasible task state transition within the RT operating system model follows the state transitions diagram (a) Model of tasks state transitions diagram managed by the RTOS model (b) Synchronization actions possible between active tasks by means of the RTOS services

**Figure 5 : State transition due to task synchronization action**

### 4.2. Modeling Concurrency and Process Control

The notion of a task or process (used interchangeable in this paper) is the underlying concept to model concurrency in the abstract RTOS. It is implemented by means of the sc_thread macro in SystemC and encapsulates the thread as a task object. Thus, the RTOS model keeps the C++ object-oriented concept that is inherent in the language where SystemC is built. Task identification is represented by the task name and is mapped to a module name (sc_module) as an implication of SystemC constructs, where modules implementing the containers are the building blocks of the SystemC model architecture. The task name as well as the main code (mainfunc, see Figure 4b) that embodies the functionality and the entry point of the task has to be passed to the operating system method when a task is being created. At the same time, some parameters such as task priority, ready time, arrival periodicity status, arrival period, and simulation time unit have to be passed as well to characterize the instance of task.

In order to support process control during the simulation run, some additional services are required in the RTOS model. These services embody the postpone/resume and the sleep/wake-up mechanisms. The allowed actions regarding synchronization are shown in table of Figure 5(b) with respect to the state of suspended tasks when the active task tries to perform a synchronization action by calling the RTOS services. The feasible task state transition within the RT operating system model follows the state transitions diagram (a) Model of tasks state transitions diagram managed by the RTOS model (b) Synchronization actions possible between active tasks by means of the RTOS services

**Figure 5 : State transition due to task synchronization action**

### 4.2. Modeling Concurrency and Process Control

The notion of a task or process (used interchangeable in this paper) is the underlying concept to model concurrency in the abstract RTOS. It is implemented by means of the sc_thread macro in SystemC and encapsulates the thread as a task object. Thus, the RTOS model keeps the C++ object-oriented concept that is inherent in the language where SystemC is built. Task identification is represented by the task name and is mapped to a module name (sc_module) as an implication of SystemC constructs, where modules implementing the containers are the building blocks of the SystemC model architecture. The task name as well as the main code (mainfunc, see Figure 4b) that embodies the functionality and the entry point of the task has to be passed to the operating system method when a task is being created. At the same time, some parameters such as task priority, ready time, arrival periodicity status, arrival period, and simulation time unit have to be passed as well to characterize the instance of task.

In order to support process control during the simulation run, some additional services are required in the RTOS model. These services embody the postpone/resume and the sleep/wake-up mechanisms. The allowed actions regarding synchronization are shown in table of Figure 5(b) with respect to the state of suspended tasks when the active task tries to perform a synchronization action by calling the RTOS services. The feasible task state transition within the RT operating system model follows the state transitions diagram (a) Model of tasks state transitions diagram managed by the RTOS model (b) Synchronization actions possible between active tasks by means of the RTOS services
as shown in Figure 5(a). This diagram models all possible state transitions of the tasks, which exist in the runtime environment of operating system. Sleep-wakeup synchronization mechanisms, as shown in table of Figure 5(b) only a currently running task is allowed to perform a sleep action. This action is invoked by calling the sleep() method of the RTOS object. This action will then shift the task to sleep state either until a specified time interval passed as a method parameter is elapsed or until the task is awakened by the currently running task. If the current task has a lower priority, then the task will be preempted by the awaked task unless there are some other tasks with higher priority values get ready to be scheduled.

ReAssPriRM

for each active tasks $\tau_i$ do
  assign null priority into task $\tau_i$
  // indicating that priority has not been reassigned yet
end for

$T \leftarrow 0$ // task period with a corresponding time unit
$P \leftarrow$ smallest priority // selected priority as lowest

Repeat
  for each active tasks $\tau_i$ do
  if task $\tau_i$ has null priority value do
    if $T$ less than period of $\tau_i$ do
      $T \leftarrow$ period of $\tau_i$
    end if
  end if
end for

for each active tasks $\tau_i$ do
  if period of task $\tau_i$ equal to $T$ do
    assigned priority $P$ into tasks $\tau_i$
  end if
end for

crement $P$ // higher value for higher priority

until no tasks has null priority value

ReAssDynPriEDF

for each active tasks $\tau_i$ do
  assign null priority into task $\tau_i$
  // indicating that priority has not been reassigned yet
end for

$S \leftarrow 0$ // remaining time to the current deadline
$P \leftarrow$ smallest priority // selected priority as lowest

for each active tasks $\tau_i$ do
  $d_i \leftarrow D_i + (CPN_i - 1) \times$ period of task $\tau_i$
end for

repeat
  for each active tasks $\tau_i$ do
  if task $\tau_i$ has null priority value do
    if $S$ less than $(d_i - t)$ do
      $S \leftarrow (d_i - t)$
    end if
  end if
end for

for each active tasks $\tau_i$ do
  if $(d_i - t)$ equal to $S$ do
    assigned priority $P$ into tasks $\tau_i$
  end if
end for

crement $P$ // higher value for higher priority

until no tasks has null priority value

(a) Fixed priority re-assignment based on RM scheduling
(b) Dynamic priority re-assignment based on EDF scheduling

Figure 6: Algorithm used in the RTOS model to assign task priority

Other synchronization actions such as postpone() and kill() can be called by the currently running task to postpone or to kill, respectively, either some other tasks or the currently active task itself. The resume() and wakeup() methods can be invoked by currently active tasks to resume and to wake-up postponed or sleeping tasks, respectively.

4.3. Model of Preemption and Granularity

The preemption model is constructed in conjunction with the timing model, i.e., by exploiting mainly the await() service provided by the RTOS model. We assume that it is possible to model atomic actions such that any functionality of a task can be built from these actions only. In other words, any function representing the functionality of a task is decomposable into a number of actions whereas their executions are atomic in the sense of ‘run-to-completion’. Each atomic action code is assumed to have a fixed current execution time, which in turn is chosen from a given timing interval for this action. The timing behavior is implemented by passing the execution time value as a parameter when calling the await() method of the RTOS module. As a consequence of this conceptional model, preemption can only happen at the end of the await() statement, which is denoted as the preemption point that determines the granularity of preemption. The execution of each atomic action code provides the ‘run-to-completion’ property and the associated piece of code is denoted as the atomic block.
4.4. Scheduler and Task Timing Model

The proposed RTOS model implements the scheduler such that the scheduling algorithm can be simulated according to selected scheduling policy. The scheduler is implemented as internal thread that executes scheduler functionality embodied in its main function as outlined in Figure 7. The priority based scheduling both static and dynamic are implemented by algorithms as shown in Figure 6 (a) and (b) respectively. The timing model of task scheduling is illustrated in Figure 6(b), where deadline of task \( t_i \), \( d_i \), is expressed in absolute value (relative to the beginning of simulation, \( t_o \)). \( D \) denotes the absolute deadline at first period (first instance) at which task \( i \) is invoked in the first time. In this time the current period number (CPN) is equal to 1. The simulation time (current time stamp) is denoted by \( t \).

```
Scheduler_main()
Begin
    Repeat
        Call wait(0,SC_NS) // until simulation is started
        if RMS is selected do
            call ReAssPriRM
        end if
        evaluate tasks that become ready at current time stamp
        evaluate tasks woke up at current time stamp
        if EDF is selected do
            call ReAssDynPriEDF
        end if
        schedule (run) active tasks according their priority
        evaluate tasks killed at current time stamp
        evaluate tasks that turns to sleep at current time stamp
        evaluate tasks postponed at current time stamp
        call wait(sc_get_time_resolution()) //granularity
        until simulation time is elapsed
end scheduler_main
```

Figure 7 : Algorithm embodies the RTOS scheduler

```
class sc_rtos_posix_if: virtual public sc_interface
{
    /* POSIX-like services*/
    virtual rt_pid get_pid(sc_module_name name)=0;
    virtual rt_pid get_pid(rt_task *)=0;
    virtual rt_task * get_task(rt_pid)=0;
    virtual sc_module_name get_name(rt_pid)=0;
    /* Task creation & termination services */
    virtual rt_pid create_task(sc_module_name name, void (__closure *mainfunc)(),
    rt_priority priority, rt_time first_ready_time,
    rt_periodicity periodicity, rt_time period,
    sc_time_unit time_unit)=0;
    virtual void kill(rt_pid)=0;
    virtual void abort(rt_pid)=0;
    /* Task synchronization services*/
    virtual void resume(rt_pid)=0;
    virtual void postpone(rt_pid)=0;
    virtual void wakeup(rt_pid)=0;
};
```

Figure 8 : POSIX-like services of the generic RTOS model

4.5. Additional Services

Supporting software architecture modeling using SystemC toward a specific RTOS API, the proposed RTOS model also provides a POSIX like interface too such as handling a task pointer by means of the task identity as typically referred by process ID or PID in POSIX [28]. The associated services have been implemented in the advocated OS model, they are highlighted Figure 8.

For this purpose the RTOS module contains some methods to access the pointers to the task objects being managed by the operating system model. The method \( get\_pid() \) requires either a task pointer or a task name, respectively, to be passed as input parameters whereas the task PID will then be returned. The method \( get\_task() \) acts as an inverse operation, i.e., it requires the task PID to be passed as input parameter and a pointer to the corresponding task will then be returned. By means this mechanism, an easy handling of all tasks based on their PIDs as commonly used in POSIX becomes possible. Thus, the design flow of embedded systems is fully supported even for such applications, where the exploitation of a POSIX-compatible RTOS is a mandatory part of the systems specification.

5. System-Level Simulation

The proposed concept of a generic RTOS model built on top of SystemC is mainly aimed to an assessment of the overall effects of various scheduling policies by means of executing the associated models of the embedded system. According to our experience, the results of this model execution, i.e., simulation of the generated SystemC models, differ extremely, especially when comparing system-level synthesis results
to simulation results exploiting an RTOS, because real task execution times vary considerably due to specific properties of the architecture of the chosen processor elements, such as cache structures and data path pipelining.

Figure 9: Assessment of different scheduling policies

The assessment of the overall system behavior resulting from different scheduling policies and from stochastic task execution timing is demonstrated for the task graph specification and HW/SW task allocation of Figure 2(a) by means of Gantt charts. The variety of RT responses is illustrated in Figure 9(a to d), where different scheduling policies are applied. An average case task execution time (ACET) is derived from high-level architectural decisions. Then, it is assumed that the completion of any previous task generated an event, which, in turn, triggers the invocation of successive tasks. The static event-driven scheduling takes these events to compose the static schedule. In other words, there are no idle times between any two tasks in sequence. This means that a shorter total execution time may be yielded in the overall system behavior. However, as can be seen from Figure 9(b), the event-driven scheduling policy does not guarantee reliability, determinism, or a faster total execution time, respectively. The systems reliability is not guaranteed, because of the possibility of failures, i.e., in case that a task fails to complete on time, then this will skip the invocation of the successive task. Furthermore, the predictability of total execution time is questionable due to the unpredictability (statistical distribution) of the actual execution time of each task. Thus, the total execution time is not guaranteed to be shorter compared to other scheduling policies.

Time triggered scheduling as illustrated in Figure 9(c) shows a better performance in terms of predictability, i.e., guaranteed total system execution time, but the assumed WCET causes rather long idle times of the processor. This means that this policy is in general less efficient in terms of resource utilization. The potential disadvantages imposed by these static-scheduling policies can be overcome by dynamic scheduling. Priority-based scheduling as illustrated in Figure 9(d) for the specification given in Fig. 8 is a typical policy in order to react dynamically to changed execution conditions, which may be caused by a slower or a faster execution of tasks as can be found in EDF scheduling. Another scheduling policy, i.e., scheduling with preemption as illustrated in Figure 9(e), can considerably reduce idle processor time and speed up the overall execution because the idle time intervals can be utilized to partially execute tasks being in the ready state without destroying the optimal schedule order on the processing elements. The main advantage of scheduling with preemption is thus the possibility to apply scheduling algorithms as found in many dynamic scheduling techniques. RM scheduling, e.g., exploits both priority-based and scheduling with preemption policies. It is obvious even from the simple example given in Figure 2(a) that the results gained from high-level system synthesis given in Figure 9(a) differ considerably from more detailed simulation models featuring various scheduling policies, because these models take task execution timing intervals into
account. This is the reason why simulation and a thorough validation of its results are mandatory at each refinement stage of the design process. Especially an introduction of preemption allows a realistic assessment of the overall system behavior and provides superior implementations in terms of both reliability and performance.

6. Application Example

Next, a considerably more complex application example – an autonomous robot equipped with ultrasound distance sensors, a camera, and with a wireless communication subsystem – is introduced for the purpose of a demonstration of the feasibility of the proposed methods to embedded systems design. The entire specification as depicted in Figure 10 is composed from 25 tasks. The envisaged generic hardware architecture for the information processing of this robot – the embedded system to be developed – consist of a micro controller located in the robot and the controlling host PC, which may be equipped by a co-processor on top of a PCI FPGA board. The communication between the mobile and the fixed parts of the embedded system is established by a wireless RS232 interface between micro controller and PC. Thus, the behavioural specification denoted as an attributed task graph in the upper part of Figure 10 has to be mapped accordingly to these three hardware resources as indicated in the lower part of the figure. The execution times for the different resources are assigned to each communication and computation task. Based on this specification the system-level synthesis determines implementation alternatives. These alternatives can be validated by the automatic generated SystemC models of the distributed information processing of the robot. The models were generated according to the proposed design approach. This forms the basis for the evaluation of system-level design decisions and to explore the effects of different scheduling policies.

6.1 Results

The evaluation results regarding task execution time for different scheduling policies in contrast to the system-level synthesis result are presented in Figure 11. Based on the previous determined system level specification and design decisions the simulation models are automatically generated and then refined, exploring different scheduling policies. Due to the stochastic execution times thousand different runs are executed and evaluated. The best, average, and worst execution times are determined. Some of resulted execution time distribution is presented in Figure 11 showing that the total execution time distribution is more focus on certain value. In other word that the deviation is less than deviation selected for individual tasks. In our case, we get the deviation is about 3% of system ACET on simulation with 10% task ACET deviation selected for each individual task used to generate task execution time that statistically has Gumbel distribution.

The time-triggered execution policy leads to deterministic execution times without variation, but on the other hand to the worst overall system performance as visible in table of Figure 11. When using the 90% timing limit (See Figure 3) instead of WCET, a speed-up of 10% can be reached, but due to the time-triggered scheduling still 1% system failures occur. Event-driven static scheduling is faster in terms of overall execution speed compared to the time-triggered method, but the system behavior is less reliable and, in addition, the probability for a task to misses its deadline can logically be higher. However, in average, the system performance is still better. So far, it can be seen that the implementation results differ from the synthesis result and between the different scheduling policies.

Priority based scheduling, as discussed before and illustrated in Figure 9(d), can provide a way for system execution time optimization. A higher priority can be given into tasks on which the total execution time is sensitive. Unfortunately, in our case, selecting priority based scheduling for the mobile robot system will not give better performance and longer execution time may yield due to overhead of scheduling algorithm. In our simulation, we assume the overhead are 1 ms and 2 ms for PC and microcontroller

---

1 In the assessment, the stochastic execution time is only applied for task running on the PC and microcontroller, whereas tasks running in FPGA and communication tasks by means RS232 are assumed to be constant.
2 For event driven scheduling implementation, we assume that a completion of previous task is an event for immediate following task to trigger its execution. This assumption is applied when following task has a timing or data dependence with previous task.
3 For priority based scheduling, we order the tasks by assigning higher priority into the tasks that has to be proceed before other tasks because the following task has a timing or data dependence with previous task.
4 As illustrated in Figure 12, the tasks of mobile robot system running on the PC determine directly the total execution time. In this sense, we call that the total execution time is sensitive to the tasks running on the PC.
respectively, resulting sensitivity movement from PC into microcontroller with 3 ms overhead on total execution time. But, as can be seen in Figure 12, there is still 5 ms idle time available in the microcontroller that can be used by the tasks behind the idle time resulting less total execution time in microcontroller. Thus by the schedule adjustment the sensitivity remains on the PC.

The preemptive scheduling method can yield better timing performance, because idle times of some of the tasks can be used in a highly flexible manner. But for this example the dynamic scheduling do not lead to better results. So, since the robot is not a highly safe-critical application, thus in this example, event driven scheduling is considered as the most feasible schedule strategy and will be the best solution for its implementation. In the mobile robot example, the figure of merit given by preemptive scheduling can not be exploited due to inter-task dependence affecting the feasibility of task scheduling as illustrated in Figure 12. In this case, no tasks can be given higher right to run by preempting current running task to proceed because they have lower precedence according to the dependence implied by task graph as system requirement.
7. Conclusions and Future Work

A realistic simulation of the system-level design decisions is mandatory for a successful design process, since the assumptions for the system-level synthesis execution semantics are very general and do not consider any scheduling policies. Therefore the proposed approach extends SystemC to the capability of general RTOS modeling and allows an early and fast validation of different implementation alternatives due to an automatic generation of system-level simulation models considering the RTOS. The use of the stochastic timing model leads to realistic system simulations of the execution properties. Thereby different scheduling policies can be tested and validated based on the general SystemC RTOS library. A subject of future work will be the granularity of the basic blocks. Fine-grained blocks will lead to a higher accuracy concerning the preemption point, but fine-grained blocks will also increase simulation time. Application of the RTOS model for modeling distributed system is another subject for future work to more extend the SystemC RTOS service in providing support for assessment of inter-nodes task synchronization in a network and global scheduling analysis.

References

[26] .