TIMES: a Tool for Schedulability Analysis and Code Generation of Real-Time Systems

Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi*

Department of Information Technology,
Uppsala University, P.O. Box 337, SE-751 05 Uppsala, Sweden
Email: {tobiasa,elenaf,leom,paupet,yi}@it.uu.se

Abstract TIMES is a tool suite designed mainly for symbolic schedulability analysis and synthesis of executable code with predictable behaviours for real-time systems. Given a system design model consisting of (1) a set of application tasks whose executions may be required to meet mixed timing, precedence, and resource constraints, (2) a network of timed automata describing the task arrival patterns and (3) a preemptive or non-preemptive scheduling policy, TIMES will generate a scheduler, and calculate the worst case response times for the tasks. The design model may be further validated using a model checker e.g. UPPAAL and then compiled to executable C-code using the TIMES compiler. In this paper, we present the design and main features of TIMES including a summary of theoretical results behind the tool. TIMES can be downloaded at www.timestool.com.

1 Introduction

In classic scheduling theory, real time tasks (processes) are usually assumed to be periodic, i.e. tasks arrive (and will be computed) with fixed rates periodically. Analysis based on such a model of computation often yields pessimistic results. To relax the stringent constraints on task arrival times, we have proposed to use automata with timing constraints to model task arrival patterns [1]. This yields a generic task model for real time systems. The model is expressive enough to describe concurrency and synchronization, and real time tasks which may be periodic, sporadic, preemptive or non-preemptive, as well as precedence and resource constraints. We believe that the model may serve as a bridge between scheduling theory and automata-theoretic approaches to system modeling and analysis. The standard notion of schedulability is naturally generalized to automata. An automaton is schedulable if there exists a scheduling strategy such that all possible sequences of events accepted by the automaton are schedulable in the sense that all associated tasks can be computed within their deadlines. It has been shown that the schedulability checking problem for such models is decidable [1]. A recent work [6] shows that for fixed priority scheduling strategy,

* Corresponding author.

the problem can be efficiently solved by reachability analysis on timed automata using only 2 extra clock variables. The analysis can be done in a similar manner to response time analysis in classic Rate-Monotonic Scheduling.

The first main function of TIMES is developed based on these recent results on schedulability analysis. Its second main function is code generation. Code generation is to transform a validated design model to executable code whose execution preserves the behaviour of the model. Given a system design model in TIMES including a set of application tasks, task constraints, tasks arrival patterns and a scheduling policy adopted on the target platform, TIMES will generate a scheduler and calculate the worst-case response times for all tasks. The model may be further validated by a model-checker e.g. UPPAAL [9], and then compiled to executable C-code. We assume that the generated code will be executed on a platform on which every annotated task in the design model will not take more than the given computing time. Further assume that the platform guarantees the synchronous hypothesis in the sense that the times for handling system functions e.g. collecting external events can be ignored compared with the computing times and deadlines for the annotated tasks. Under these assumptions on the platform, code generation is essentially to resolve non-determinism in the design model. In TIMES, time non-determinism is resolved by the maximal progress assumption, that is, whenever a transition is enabled, it should be taken. External non-determinism in accepting events is resolved using priority order.

The rest of the paper is organized as follows: the next section describes the core of the input TIMES language and its informal semantics. Section 3 summarizes briefly the main theoretical work on schedulability analysis and code synthesis. Section 4 describes the main features of TIMES, the tool architecture and the main components in the implementation. Section 5 concludes the paper with a summary of ongoing work and future development.

2 Task Models in TIMES

The two central concepts in TIMES are *task* and *task model*. A task (or task type) is an executable program (e.g. in C) with task parameters: worst case execution time and deadline. A task may have different *task instances* that are copies of the same program with different inputs. A task model is a task arrival pattern such as periodic and sporadic tasks. In TIMES, timed automata are used to describe task arrival patterns.

2.1 Tasks Parameters and Constraints

Following the literature [4], we consider three types of task constraints.

Timing Constraints A typical timing constraint on a task is deadline, i.e. the time point before which the task should complete its execution. We assume that the *worst case execution times* (WCET) of tasks are known (or pre-specified). We characterize a task as a pair of natural numbers denoted (C, D) with $C \leq D$, where C is the WCET of P, D is the relative deadline for P. In general, the execution time of a task can be an interval $[C_B, C_W]$ where C_B and C_W are the best and worst case execution times. The deadline D is a relative deadline meaning that when task P is released, it should finish within D time units.

Precedence Constraints The execution of a task set may have to respect some precedence relations. These relations are usually described through a precedence graph in which nodes represent tasks and edges represent precedence relation. In TIMES, we use cyclic AND/OR-precedence graphs in which we distinguish *ordinary* and *inter-iterative* edges (denoted $\dashrightarrow$) [3] such that inter-iterative precedence constraints apply to all task instances except for the first one. An example of such graph is shown in Figure 1.

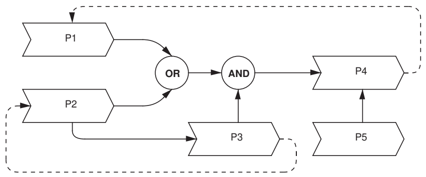


Figure1. Example of cyclic AND/OR precedence graph.

According to the graph, P_4 can start its execution only if it is preceded by P_3 and either P_1 or P_2. The first instance of task P_1 can start its execution at any time while any further instance of P_1 must be preceded by task P_4.

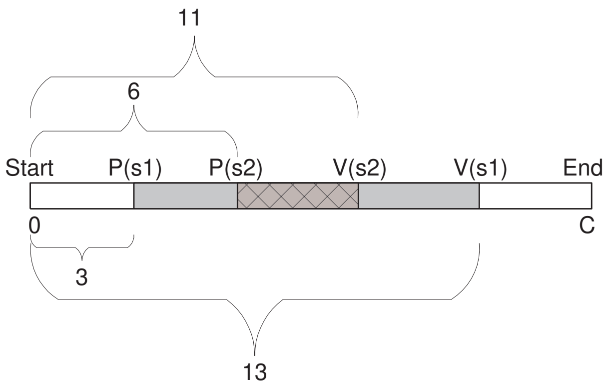


Figure2. An example semaphore access pattern.

Resource Constraints Tasks may share resources or data variables protected by semaphores. A task must follow its given semaphore access pattern to lock and unlock semaphores, which is the resource constraint on the task. The access to semaphores will be scheduled using priority ceiling protocols e.g. the highest locker protocol [10]. A semaphore access pattern for a task is a list of timed semaphore-operations in the form: $\{S_i(P_i, V_i)\}$ where S_i is the semaphore name, P_i is the accumulated execution time needed for the task to reach the lock-operation on S_i and V_i is the accumulated execution time needed for the task to reach the unlock-operation on S_i. The blocking time for S_i is $V_i - P_i$. An example semaphore access pattern $\{S_1(3, 13)\}; S_2(6, 11)\}$ of a task is illustrated in Figure 2. The task will try to lock S_1 when it has been executed for 3 time units and it will lock it for 10 time units.

2.2 Timed Automata as Task Arrival Patterns

The core of the TIMES input language is timed automata extended with data variables [9] and tasks [5] and [7]. As in the UPPAAL model, each edge of such an extended automaton is labeled with three labels:

1. a *guard* containing a clock constraint and/or a predicate on data variables.
2. an *action* which can be an input or output action in the form of $a!$ and $a?$.
3. a *sequence* of assignments in the form: $x := 0$ when x is a clock or $v := E$ when v is a data variable, where E is a mathematical expression over data variables and constants.

A location of an extended automaton may be annotated with a task or a set of tasks that will be triggered when the transition leading to the location is taken. The triggered tasks will be put in a task queue (i.e. ready queue in operating system) and scheduled to run according to a given scheduling policy. The scheduler should make sure that all the task constraints are satisfied in scheduling the tasks in the task queue. To model concurrency and synchronisation between automata, networks of automata are constructed in the standard way as in e.g. UPPAAL with the annotated sets of tasks on locations unioned.

2.3 Shared Data Variables

Four types of shared data variables can be used for communication and resource sharing:

1. Tasks may have shared variables with each others, protected by semaphores.
2. Tasks may read and update variables owned by the automata.
3. Automata can read (but not update) variables owned by the tasks.
4. Automata may have shared variables with each other.

3 Analysis and Synthesis

In TIMES, a timed automaton annotated with tasks (or network of such automata) is considered as a design model. The tool offers two main functions: schedulability analysis of design models and generation of executable code from the models.

3.1 Schedulability Analysis

In [7], an operational semantics for timed automata extended with tasks is developed. A semantic state of such an automaton is a triple (l, u, q) where l is the current control location, u denotes the current values of clocks and data variables, and q is the current task queue keeping all the released tasks to be executed. The semantics of an automaton is defined by a transition system in which the transition rules are parameterized by a scheduling policy to schedule the task queue when new tasks are released.

Given an extended automaton and a scheduling policy, the related schedulability analysis problem is to check whether there exists a reachable state (l, u, q) of the automaton where the task queue q contains a task which misses its given deadline. Such states are called *non-schedulable* states. An automaton is said to be non-schedulable with the given scheduling policy if it may reach a non-schedulable state. Otherwise the automaton is schedulable. As the number of reachable states of an extended automaton is infinite, it is not obvious that the schedulability analysis problem is decidable.

The first decidability result is presented at TACAS 2002 showing that the schedulability checking problem for the optimal scheduling policy i.e. EDF can be solved by reachability analysis on timed automata extended with subtraction on clocks. Consider an automaton A and a scheduling strategy Sch. To check if A is schedulable with Sch, we construct timed automata $E(\mathsf{Sch})$ (the scheduler), and $E(A)$ (the task arrival pattern), and check the reachability of a predefined error state in the product automaton of the two. If the error state is reachable, automaton A is not schedulable with Sch.

The maximal number of clock variables needed in constructing the scheduler automaton is $2n$ where n is the total number of schedulable task instances $\sum_{i \in \mathcal{P}} \lceil D_i/C_i \rceil$ where $\mathcal{P}$ is the set of task types, and C_i, D_i are the computing time and deadline for each task type i.

To construct $E(A)$, the automaton A is annotated with distinct synchronization actions $\mathsf{release}_i$ on all edges leading to locations labeled with the task name P_i (assume that only one task is annotated). The actions will allow the scheduler to observe when a task is released by A for execution. The structure of $E(\mathsf{Sch})$ is shown in Figure 3.

The main idea is to keep track of the task queue, denoted by q on each step of the reachability analysis. Therefore in the encoding $E(\mathsf{Sch})$ there is a transition with the guard $nonschedulable(q)$ from every location where the queue is not empty (i.e. from all locations except Idle) to the error state. In the encoding, the task queue q is represented as a vector containing pairs of clocks (c_i, d_i) for every

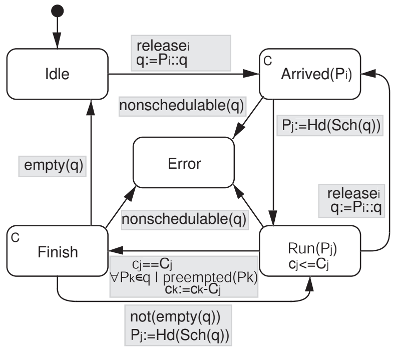


Figure3. Scheduler automaton.

released task instance, called execution time and deadline clock respectively. The intuitive interpretation of the locations in $E(\mathsf{Sch})$ is as follows:

– Idle - the task queue is empty,
– Arrived($\mathsf{P_i}$) - the task instance P_i has arrived,
– Run($\mathsf{P_j}$) - the task instance P_j is running,
– Finished - a task instance has finished,
– Error - the task queue is non-schedulable.

Locations Arrived($\mathsf{P_i}$) and Finished are marked as committed, which means that they are being left directly after entering.

We use the predicate nonschedulable(q) to denote the situation when the task queue becomes non-schedulable and naturally there is a transition labeled with the predicate leading to the error-state. The predicate is encoded as follows: $\exists P_i \in q$ such that $d_i > D_i$.

We use Sch in the encoding as a name holder for a scheduling policy to sort the tasks queue. A given scheduling policy is represented by the predicate: $P_i = \mathsf{Hd(Sch(q))}$. For example, Sch can be:

– Highest priority first (FPS): $P_i \in q, \forall P_k \in q$ $\mathsf{Pri(P_i)} \leq \mathsf{Pri(P_k)}$ where $\mathsf{Pri(P)}$ denotes the fixed priority of P.
– First come first served (FCFS): $P_i \in q, \forall P_k \in q$ $d_i \geq d_k$
– Earliest deadline first (EDF): $P_i \in q, \forall P_k \in q$ $D_i - d_i \leq D_k - d_k$
– Least laxity first (LLF): $P_i \in q, \forall P_k \in q$ $c_i - d_i + D_i - C_i \leq c_k - d_k + D_k - C_k$

For more detailed description of the automaton $E(\mathsf{Sch})$, see [7].

Variant execution times. The analysis for tasks with constant execution times can be extended to deal with interval execution times: $[C_{iB}, C_{iW}]$ for each task P_i (the best case and worst case execution times). The idea is to modify the scheduler automaton as shown in Figure 4. We use c_i to keep track of the lower

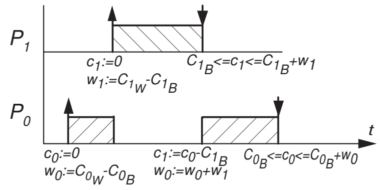


Figure4. Varying execution times.

bound of the accumulated execution time for P_i, and w_i to denote the accumulated difference between best and worst completion time of P_i. Obviously w_i should be set to $C_{iW} - C_{iB}$ in the beginning of task execution. Observe that each preemption will enlarge the difference for the preempted task with lower priority by the difference for the finishing task with higher priority. Accordingly, we modify the scheduler automaton as follows: The guard on edge from location $\mathsf{Run(P_j)}$ to $\mathsf{Finished}$ should be $C_{jB} \leq c_j \leq C_{jB} + w_j$ and variable updating should be $c_k := c_k - C_{jB}, w_k := w_k + w_j$ for all k such that $\mathsf{preempted}(P_k)$. The rest of the scheduler automaton reamins the same as before.

Fixed priority scheduling policy. In a recent work [6], it is shown that the schedulability problem for Fixed Priority Scheduling Policy can be solved efficiently using ordinary timed automata with only two clock variables (in addition to the original clocks used to describe task arrivals). For models with shared data variables (e.g. data dependent control when the values of data variables of a task may influence the release time of task instances), the number of clocks needed in the analysis is $n + 1$ where n is the number of tasks involved in the data sharing. More recently these results are extended to handle precedence and resource constraints [8] and implemented in TIMES.

3.2 Code Generation

The second main function of the tool is code generation. We consider automata extended with tasks as design models. Code generation is to transform a validated design model to executable code whose execution preserves the behaviour of the model. We assume that the generated code will be executed on a platform on which every annotated task in the design model will not take more than the given computing time. Further assume that the platform guarantees the synchronous hypothesis in the sense that the times for handling system functions e.g. collecting external events can be ignored compared with the computing times and deadlines for the annotated tasks. Under these assumptions on the platform, code generation is essentially to resolve non-determinism in the design model.

Deterministic semantics A model can exhibit two types of non-determinism: *time non-determinism*, i.e. that enabled transition can be taken at any time point within the time-zone, and *external non-determinism* i.e. that several actions may be simultaneously present from the environment. To overcome the problems introduced by this we adopt a deterministic semantics that define a subset of the behaviour. External non-determinism is resolved by defining priorities for action transitions in the controller. If several transitions are enabled in a state the one with the highest priority is taken. Time non-determinism is resolved by adopting the so-called maximal-progress assumption [11]. Maximal-progress means that the controller should take all enabled transitions until the system stabilises, i.e. no more action transitions are enabled.

Structure of the generated code TIMES is currently able to generate code for a small generic operating system (brickOS), and code for platform independent execution. The generated code is in C and an optimising compiler is used to compile the final program. For both cases, the control structure of the timed automata is encoded into four tables and two functions. These are used by an event handling procedure which is invoked on events (such as timeouts and arrival of external events) to update the state of the controller. When an action transition has been executed the event handling procedure will continue to execute transitions until a stable state is reached, i.e. it implements the maximal progress or run-to-completion semantics.

Code generation for brickOS brickOS is a small open source operating system designed to run on the Hitachi H8 equipped RCX control brick in the LEGO®Mindstorms system. We consider brickOS to be a reasonable example of a target platform running a small operating system. On this target we let the tasks execute as separate threads which are scheduled by the underlying operating system. Due to limited support for interrupts the event handling procedure is executed every time the OS scheduler is executed (i.e. every 20 ms).

Platform-independent code generation The platform independent target does not rely on any specific operating system, instead it implements its own run-time system based on the scheduler automaton created for schedulability analysis. The run-time system also includes code to handle task release and execution, and an event handler that is invoked periodically to poll for new events. The current implementation of the platform independent code can only handle non-preemptive tasks.

4 Tool Overview

In this section, we present the main features of TIMES, the tool architecture and the main components in the implementation.

4.1 Features

Figure 5 illustrates a design process using TIMES. As shown in the use case, TIMES offers the following main features:

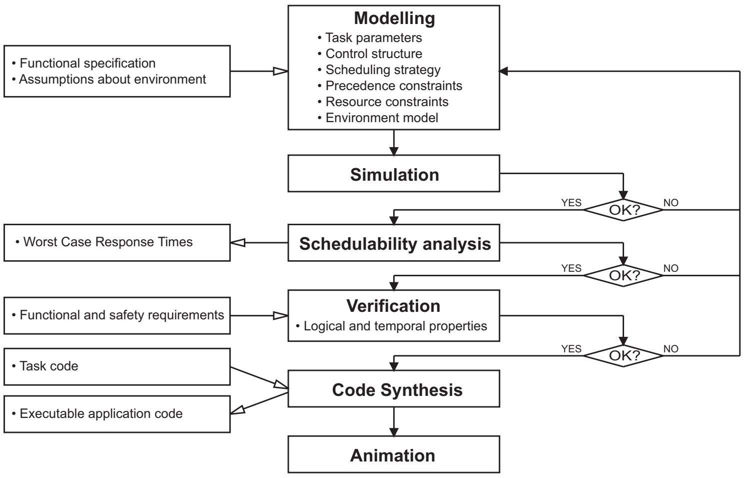


Figure5. The design process using TIMES.

– **Editor** (see Figure 6) to graphically model a system and the abstract behaviour of its environment. A system description consists of a task set and a network of timed automata extended with the tasks.

  A task is described by the task code (in C), its (worst-case) computation time and (relative) deadline, and if applicable optional parameters for priority (for fixed priority scheduling), period (for periodic tasks), and minimal inter-arrival time (for sporadic tasks). It is also possible to specify precedence constraints on the tasks using an editor for AND/OR precedence graphs, and resource access patterns using semaphores.
– **Simulator** (see Figure 7) to visualise the dynamic behaviour of a system model as Gantt charts and message sequence charts. The simulator can be used to randomly generate possible execution traces, or alternatively the user can control the execution by selecting the transitions to be taken. The simulator can also be used to visualise error traces produced in the analysis phase.
– **Analyser** to check that the tasks associated to a system model are guaranteed to always meet their deadline. In case schedulability analysis finds a task that may fail to meet its deadline, a trace is generated and visualised in the simulator. It is also possible to compute the worst-case response times of individual tasks. Recently, an improved schedulability analysis algorithm has been developed for tasks with fixed priorities without dependencies [6]. The schedulabilty analysis has also been extended to handle resource and precedence constraints [8]. In addition to scheduling, it is possible analyse safety and liveness properties specified as temporal logic formulae.
– **Compiler** to generate executable C code from timed automata with tasks. The compiler assumes that the target platform ensures the asynchronous

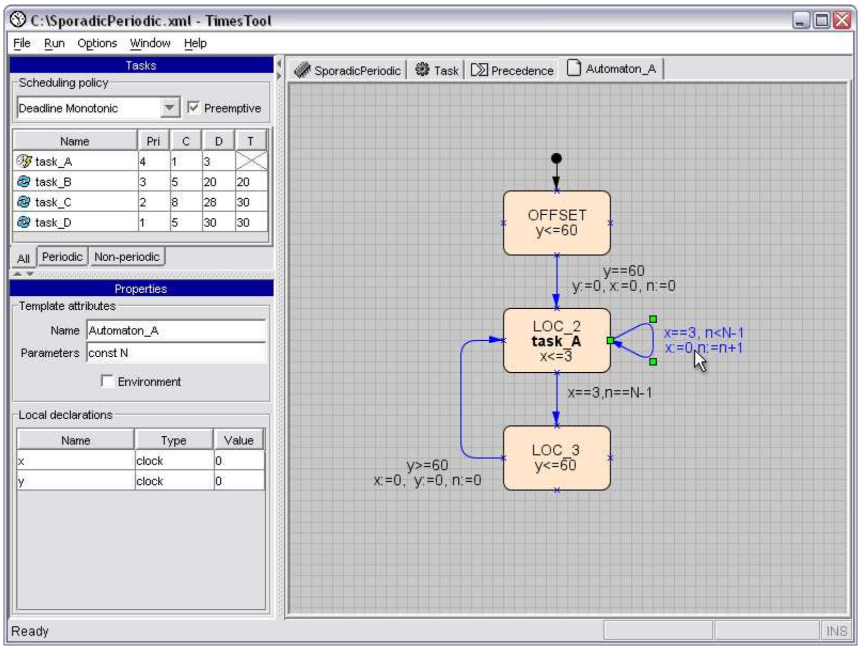


Figure6. The TIMES editor.

hypothesis and that the task code can be executed in the specified computation time. To produce executable code, the compiler relies on a deterministic refinement of the semantics that realise a subset of the behaviour specified in the timed automata of a system model. In this way, the generated code is guaranteed to satisfy analysis results from e.g. schedulabilty analysis when executed on the target platform. The currently implemented compiler supports code generation for: the brickOS operating system (that uses the scheduler in the brickOS runtime system), platform independent code (C code for GNU gcc, including code for a scheduling policy), and code for the Animator of TIMES.

– **Animator** to transform hybrid automata modeling the controlled environment into C code simulating the controlled objects in the environment of the embedded system. The simulated environment enables the designer to experiment with the design prior to implementation.

4.2 Implementation

The architecture of the TIMES tool is illustrated in Figure 8. Logically it is divided in three main parts:

– **Graphical User Interface** consisting of editors, simulator, analyser, and animator, as described above. The graphical user interface is implemented

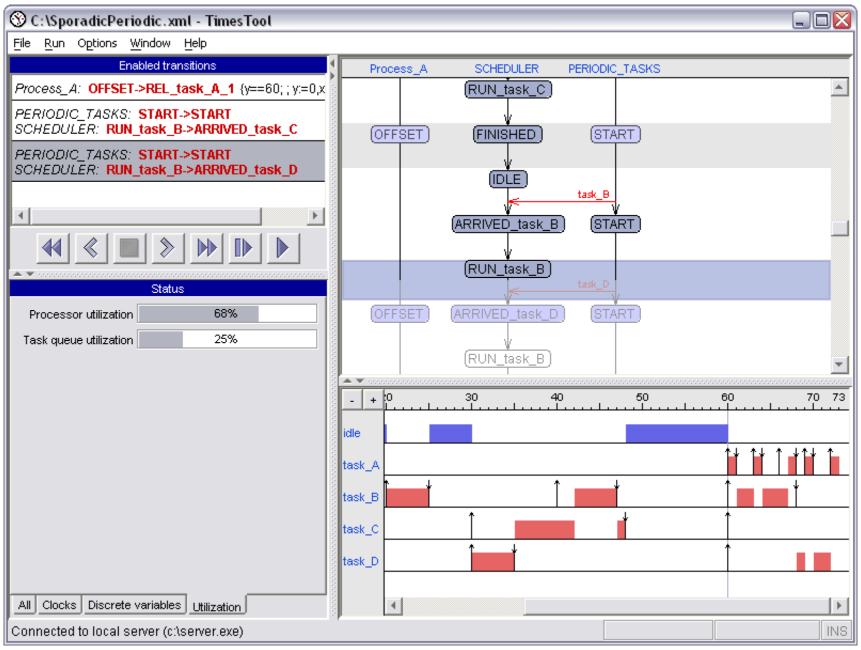


Figure7. The TIMES simulator.

entirely in Java and uses XML to represent the system descriptions both internally and externally (on file).

– **Server** consisting of two parts: a scheduler generator implemented in Java, and a module for schedulabilty analysis based on the Uppaal engine [9] with extensions, like the rest of the Uppaal engine implemented in C++. The scheduler generator produces a scheduler automaton based on input from the editor, which is composed in parallel with an annotated version of the original system automata. The parallel composition is analysed by on-the-fly reachability techniques in the schedulabilty analysis module. Currently supported scheduling policies are: rate monotonic, deadline monotonic, fixed priority scheduling (with user defined priorities), earliest deadline first (EDF), and first come first served (FCFS). All scheduling policies support preemptive or non-preemptive task sets.
– **Compiler** that takes as input the XML system representation from the editor and the task code segments to produce executable code of the application. The generated code consists of three main parts: a set of C-functions (look-up tables) representing the automata of the system representation, a generic part storing and updating the current state according to the look-up tables, and possibly an implementation of the scheduling strategy (in case platform independent code is produced).

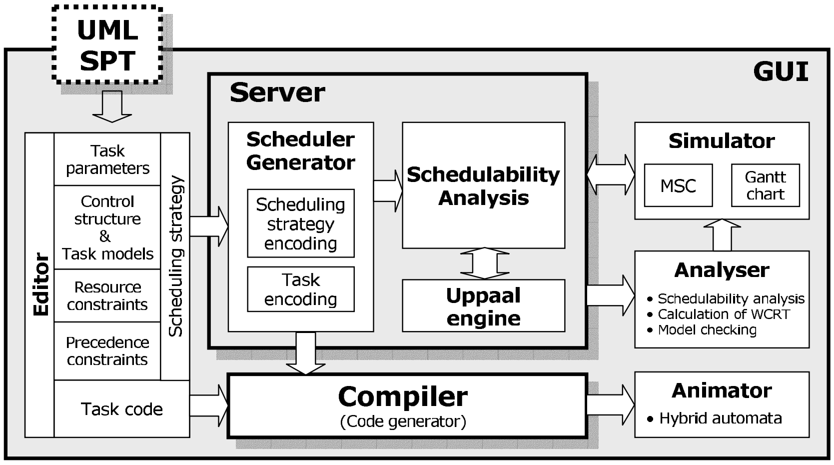


Figure8. The TIMES tool architecture.

5 Applications and Current Development

Case Studies Currently we are in the process of using TIMES to verify reliable message transmission with TTCAN (Timed Triggered CAN). So far, the only non-trivial example using TIMES is the development of the control software of a production cell (a well-studied case in verification), consisting of an industrial robot, a press and two transportation belts to process and move metal plates. The robot controller is designed as a timed automaton annotated with tasks. A complete description of the case study can be found in [2]. It is a non-trivial application involving 12 tasks (task types), 7 automata, 17 integers, 24 booleans and 31 clock variables (7 in the model and 24 in the scheduler). The schedulability (and a number of other requirements) of the system is verified on a machine equipped with two 1.8 GHz AMD processors and 2 GB of main memory, running Mandrake Linux. TIMES consumes 207 MB of memory and terminates in 11 minutes. Using the option for over approximation (based on the convex-hull approximation, the analysis requires only 13 MB and 9 seconds on the same machine.

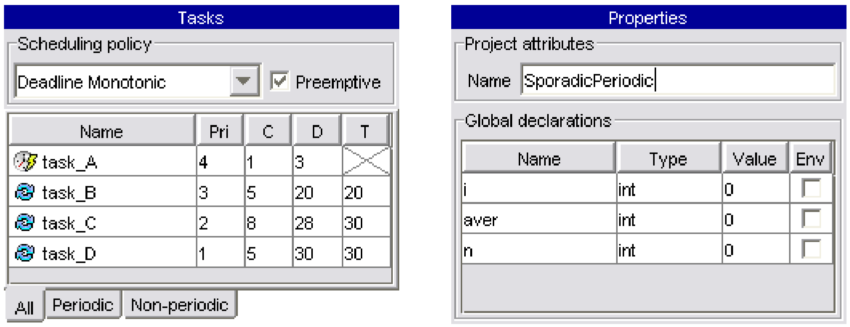
UML SPT profile SPT (Scheduling, Performance, and Time) specification is a UML profile developed recently as an extension of the UML standard to model time and time-related aspects of embedded systems. An ongoing work has been initiated with I-Logix to develop TIMES as a plug-in tool for schedulability analysis of UML diagrams in Rhapsody, annotated with stereotypes, constraints, and tag definitions according to the UML SPT profile.

Appendix: A Brief Tutorial

This section contains a step-by-step tutorial describing how to create and verify a model using the Times tool.

Creating a project: Run the Times tool and it will open an editor with a new empty project. Specify the project name `SporadicPeriodic` in the Name field of the project attribute window. Save the project (File→Save as...) using the name `SporadicPeriodic.xml`.

Adding tasks: Start modelling by creating four tasks and specifying their parameters in the task table as shown in Figure 9(a). In order to add a periodic or non-periodic task to the task table right click inside the table and select the item Add periodic task or Add controlled task respectively. To change a task parameter double click on an appropriate field of the table and type in a new value. Select preemptive Dealine Monotonic scheduling policy as shown in Figure 9(a).



(a) Task table (b) Project properties

Figure9. Times project elements.

A task table can be imported from an external text file using menu command File→Import.... An example file representing the task table from the Figure 9(a) is shown in Figure 10.

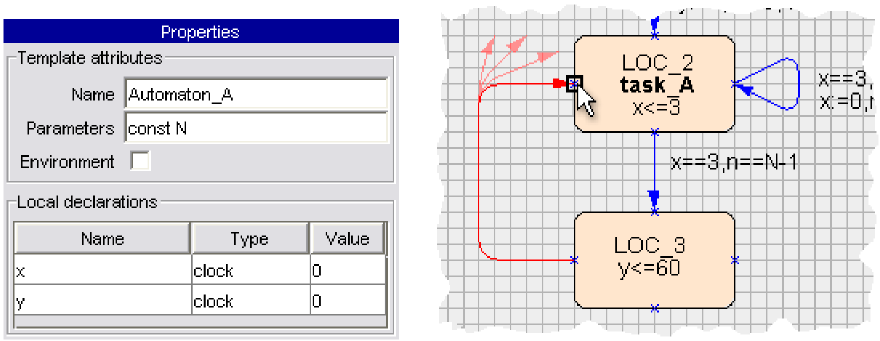
Declaring global variables: Declare three integer variables i, aver and n in the global declarations table of the properties window as shown in Figure 9(b). In order to add a variable right click inside the table and select the item Add declaration from the popup menu. Assign an appropriate name to each variable and set the value of the Type field to int by making a selection from the drop-down menu.

```
[SchedulingPolicy]
DM                  // Deadline Monotonic.
[NonPeriodic]       // Tasks released by
Name      C  D      //     control automaton.
task_A    1  3
[Periodic]          // Periodic tasks.
Name      C  D  T   // Specify parameter names.
task_B    5 20 20
task_C    8 28 30   // Fill in the table
task_D    5 30 30   //   with task parameters.
```

Figure10. Task table specified as a text file.

Creating a control structure: The next step is creating timed automaton representing the release pattern for task A, or rather an automaton template that will be instantiated later forming a runtime release control thread for the task A.

Right click on the project tab and select the item Add template from the popup menu. A new tab named Template1 will be added to the project. Open it and enter `Automaton_A` in the Name field of the properties window. Type in `const N` in the Parameters field and add two local clocks x and y as shown in Figure 11(a).



(a) Template properties (b) Creating a transition

Figure11. Creating an automaton template.

Draw the timed automaton shown in Figure 6 in the drawing area under the template tab. In order to create a location, double click on the drawing area, or right click on it and select Create→Location from the popup menu. Drag a

Task name	Interface expression	Semaphores
A	`aver:=(n==0?i:(aver*n+i)(n+1))`	`res1(0,1)`
B	`i:=i*i`	`res1(4,5),res2(1,2)`
C	`i:=i+1`	`res1(3,7),res2(1,2)`
D	`i:=(i>=24?0:i)`	`res1(4,5)`

Table1. Task interfaces and semaphores

location to arrange its position. In order to modify location arguments double click on it and make changes using an inline editor. In order to mark location as initial, right click on it and select Location→Make initial from the popup menu.

To create a transition from one location to another, double click on a port (blue cross) of the source location, move the pointer to the destination location and click on its port as shown in Figure 11(b). Nails (turning points) can also be added to the transition line by clicking at their positions while creating a transition. The trasition arguments are modified using the same procedure as applied to the location arguments.

Specifying interfaces: After a task has completed its execution it may update some of the system variables. In order to reflect that in the model an interface between the task and the control structure is defined. In the TIMES model an interface is an expression assigned to every task describing how the global variables are changed by the end of its execution. In order to define an interface select a task in the task table, then open the Tasks tab, and enter an expression in the Interface field. Use expressions from Table 5 to specify interfaces for all the tasks.

Instantiating the control structure: The automaton template that we have created on an earlier stage should be instantiated. In order to do that open the project tab and create a process in the drawing area by right clicking on it and selecting Create→Process from the popup menu. Set the name of the process to `Control_A`, the Template argument to `Automaton_A` and pass 5 as a constant parameter.

Introducing semaphores: Let us assume that tasks are using a shared resource. In order to keep it in a consistent state only one party at a time is allowed to access it. To achieve this, a mutual exclusion mechanism between tasks is implemented based on the semaphores associated with the shared resources. The *lock* and *unlock* primitives, which delimit a critical section, are defined by the time points relative to the task execution time. The list of semaphores with lock/unlock time points is specified for each task in the Semaphores field of the Tasks tab in the form:

$$semaphore\_name(lock\_timepoint, unlock\_timepoint), ...$$

For example, an expression $res1(4,5)$ is read "the task has an exclusive access to the resource controlled by a semaphore $res1$ for 1 time unit staring at the moment it has been running for 4 time units". In our example there are two resources guarded by semaphores res1 and res2 respectively. Use expressions from Table 5 to define semaphores for all the tasks.

Creating a precedence graph: Sometimes tasks have to respect certain precedence relations to express, for example, explicit input/output relation between them. A precedence graph allows us to describe these relations inducing a partial order over a task set. Here we show how to specify such order between task D and task C such that task D will always precede task C. Open the Precedence tab, and create two graph nodes using procedure similar to one of creating locations in the automaton template. Set the task argument to `task_D` for one node and `task_C` for another. Create a precedence relation starting at node task_D and ending at node task_C using the technique similar to one for creating transitions in the automaton template. The final result should look as in Figure 12.

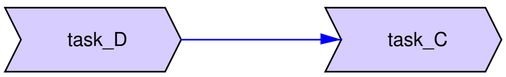


Figure12. The precedence graph

Simulating the system: Simulation is the first step in the model analysis. Run the simulator selecting item Run→Simulation in the main menu. A syntax checking procedure will be run automatically and if the model is syntactically correct, the simulator window will be opened immediately (see Figure 7).

The simulator window is divided in four parts: Enabled transitions, Message sequence chart (MSC), Gantt chart, and Watches. In the first part there is a list of transitions that the system can perform from the current state. The message sequence chart displays processes within the control structure and communications between those. For example, the timed automaton, which we have instantiated with the name `Process_A` is displayed as the leftmost process. Process that controls releases of periodic tasks as well as the scheduler process have been generated automatically by the TIMES tool. Red horizontal arrows in MSC show message passing between the processes. For example, the scheduler process receives notifications about task releases from the task control processes. On the Gantt chart one can see the timeline with the tasks being executed, and the processor idle time. The watches part has a tabbed layout and is intended for tracing the values of variables, clocks, and other readings such as processor utilization, and task queue utilization during simulation. Use the control buttons to start simulation in a step-by-step or random run mode.

Let us check using the simulator that the precedence relation between task D and task C holds, and the shared resource is accessed by the tasks correctly. Indeed, as follows from the Gantt chart shown in Figure 7, task C is preempted twice by task D due to a precedence relation, even though task C has a higher priority. Shared resource is exclusively accessed by the tasks causing blocking of the task B on an interval [40,42], and task A on an interval [66,67].

Analysing schedulability: Schedulability analysis checks, exploring all possible execution traces, whether resources needed to execute tasks are assigned so that all the tasks meet their deadlines. Run the schedulability analysis selecting the item Run→Schedulability analysis from the main menu. The result should be SATISFIED. Once schedulability analysis has been performed and the result is positive, the values of the task worst case response times (WCRT) are available. These values are the longest times needed for a task to complete its execution taken among all its instances. In order to display a list of WCRT values press the Show WCRT button in a schedulability analysis dialog window. Clearly, in a correct system WCRT value of a given task should be greater or equal than its execution time and less or equal than its relative deadline.

Checking safety and liveness properties: Apart from schedulability, other model properties can be checked. Safety properties assert what the system allowed to do, or equivalently, what it may not do, while liveness properties specify what the system must do. Open the verifier dialog window by selecting Run→Verification from the main menu. Enter and check the following properties:

`A[] (aver<10)` (The variable aver is always lower than 10)
`E<> (aver>5)` (The variable aver eventually becomes greater than 5)

After obtaining the result of the verification of the last property press the Show trace button in the verification dialog window. The shortest execution trace satisfying the liveness property will be loaded in the simulator. Use the variable watches to check that the value of the variable aver is greater than 5. Navigate the simulator one step back along the trace and ensure that aver is less than 5.

Generating executable application code: Before generating executable application code out of the model it is necessary to associate appropriate pieces of the code with every task. Close the simulator and open the Tasks tab in the editor. Under this tab there is a text editor where you can create the code and associate it with the task. Enter the code from Table 5 to the corresponding task code editors and save each piece in a separate file on the disk. A reference to the saved file should appear in the Code pointer field in the Tasks tab.

Select Run→Code synthesis in the main menu to run the code generator. This will create C-code in SporadicPeriodic.c and SporadicPeriodic.h files. Additional target specific files will be created in the same directory together with the make

Task A	Task B

`if( n == 0 ) {`
`aver = i;`
`} else {`
`aver = (aver*n+i)/(n+1);`
`}`
	`#include <math.h>`
	
	`i = pow( i , 2 );`

Task C	Task D
`i = i + 1;`	`if ( i >= 24 )` `i = 0;`

Table2. Task code.

file. Build an application by executing `make` in the source directory.

References

1. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times - a tool for modelling and implementation of embedded systems. In *Proc. of TACAS'02*, volume 2280 of *LNCS*, pages 460–464. Springer, 2002.
2. Tobias Amnell, Elena Fersman, Paul Pettersson, Hongyan Sun, and Wang Yi. Code synthesis for timed automata. *Nordic Journal of Computing*, 9(4):269–300, 2002.
3. F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-vincentelli. Scheduling for embedded real-time systems. *IEEE Design & Test of Computers*, 15(1):71–82, 1998.
4. G. C. Buttazzo. *Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications.* Kulwer Academic Publishers, 1997.
5. C. Ericsson, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proceedings of Nordic Workshop on Programming Theory*, 1998.
6. E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis using two clocks. In *Proc. of TACAS'03*, volume 2619 of *LNCS*, pages 224–239. Springer, 2003.
7. E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *Proc. of TACAS'02*, volume 2280 of *LNCS*, pages 67–82. Springer, 2002.
8. Elena Fersman and Wang Yi. A generic approach to schedulability analysis of real-time tasks. Submitted for publication., 2003.
9. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
10. R. Rajkumar, L. Sha, and J.P. Lehoczky. An experimental investigation of synchronisation protocols. In *Proceedings 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 11–17. IEEE Computer Society Press, 1998.
11. Wang Yi. *A Calculus of Real Time Systems.* PhD thesis, Department of Computer Science, Chalmers University of Technology, 1991.