Technical Report CSM-341
Kernel Abstractions

Thorsten Gerdsmeier and Rachel Cardell-Oliver
Department of Computer Science, University of Essex
COLCHESTER CO4 3SQ, UK
gerdst@essex.ac.uk,cardr@essex.ac.uk

This paper presents a method for analyzing timed behavior of concurrent Ada programs using a combination of SPARK, the specification language TLA+ and the model checker UPPAAL. We prove the correctness of Spark procedures according to their pre and postconditions with the SPARK & Spade proof tools. Using the proof contexts, actions in TLA+ are defined. From the resulting TLA+ specification a sequence of abstractions are derived resulting in an abstraction that can be interpreted as a timed automata and used as input to the model checker UPPAAL for analysis of real time properties. The method has been used for scheduling analysis predicting response times of tasks more precise than traditional schedulability tests.

1 Introduction

The programming language Ada95 has been widely and successfully used to implement multi-task, real-time systems. However, existing verification methods for Ada do not provide a way to validate that a concurrent, real-time program satisfies real-time requirements. In order to provide this we need :

1. a computational model which can express both concurrency and timing properties
2. an interpretation of Ada95 programs and of real-time requirements in this model
3. an abstraction relation between implementations and requirements
4. tool support to manage the verification proofs.

We present a method for verifying real-time properties of Ada95 programs using the computational model of timed automata with data variables [2, 15]. In [12] we showed how the specification language of timed automata could be used to specify and verify real-time properties of an ICPP scheduler, for example that every task in a concurrent program meets its deadline. In this paper, we show how to verify the link between programs and timed automata, thus completing a method for proving that an Ada95 program meets its real-time requirements.
Analysis of concurrent Ada programs has been investigated by several authors. In [6] concurrent Ada programs are automatically verified with colored Petri Nets. Verification is untimed and only checks for deadlocks.
In [11] an Ada95 program is translated to a Time Petri Net for deadlock detection. Every program statement is translated to a certain Time Petri Net structure.
In [16] a runtime system using the *delay until* statement is specified in UPPAAL. A scheduler has not been modeled nor is there an Ada implementation which has been abstracted. Automata specifying a delay queue, a periodic task template using *delay until* and a clock automata is modeled in UPPAAL. Properties are verified : when the task is removed from the queue and ready for scheduling, the variable *Time* will always be larger than what the tasks delay time was and also never more than one task can execute at a time.
In [10] a real-time kernel implementing the Ravenscar Profile [8] is specified

in HOL. Properties of the HOL specification like mutual exclusion of tasks and simple properties like : *if a task is executing, its state is ready*, are checked in HOL using the PVS theorem prover.
Our work differs to the previous in the following. We produce smaller timed automata from an Ada implementation by two abstraction steps. The first step consists of defining pre and post-conditions in SPARK [4]. This is an abstraction of sequences and removal of parts we do not need for verification (for example local variables). A second abstraction step is done in TLA by using refinement mapping. This abstraction step includes encoding of variables in locations and abstraction of arrays to simple integer variables. We include time in our abstraction and in addition to deadlock properties, we can check timed safety and response properties. We also analyze the behavior of a set of tasks under scheduling [12].
The main contribution of this paper is a method for verifying real-time properties of concurrent Ada95 programs. We claim that our method is scalable because it is based on the deductive framework of TLA+ for refinement which is itself scalable. We also use automatic verification tools where appropriate: the SPARK analyzer to verify pre- and post- conditions of Ada95 procedures within a single task, and the UPPAAL model-checker to verify that an abstract timed automata model of a program satisfies real-time requirements. Our approach is semi automatic since annotating the Ada code with pre and postconditions and developing a refinement mapping can only be done with human interaction.
The language of TLA+ is used to describe Ada95 programs and to construct correct abstractions into timed automata which can be verified using the UPPAAL model-checker. TLA+ is a general purpose language for modeling processes. It allows us to describe systems on different levels of abstraction. We can describe a system on the implementation level as well as on the more abstract level of UPPAAL timed automata. Most importantly, it has well studied refinement rules which we have used to construct an abstract specification from the program model.
The paper is organized as follows. Section 2 is an overview of our method. In section 3 we briefly describe a scheduling kernel implementation of the Ravenscar Profile [7] with ICPP scheduling. Modeling of tasks in TLA+ is described in section 4. In section 5 we translate a periodic user task in SPARK to a TLA specification. In section 6 we abstract the specification to a timed automaton in several steps. We explain the refinement in abstraction steps in the refinement calculus in TLA . Section 7 introduces the resulting specification that can be interpreted as a timed automaton. Sec-

tion 8 draws some conclusions on the work.

2 Outline of the Method

In order to construct an UPPAAL timed automata specification of an Ada95 implementation, our method translates the Ada implementation into a TLA+ specification, constructs a series of abstractions of that model and then translates the resulting abstract TLA+ specification into an UPPAAL timed automata specification. Real-time requirements on the final UPPAAL specification can be automatically verified using the UPPAAL model-checker.
Model checking suffers under the well known problem of state explosion. The abstractions of the implementation we derive have the purpose to construct a specification that can be implemented in the model checker and they have the purpose to get a specification that can be checked efficiently. For example the CHOOSE operator that might be used in a TLA specification cannot be implemented directly in the language of our model checker. We have to derive an abstraction that does not use it but preserves the important properties of the original specification. Another example is the use of data variables. In the final abstraction we can only use data types that are supported by the input language of the model checker. The state space of the automata of the input to the model checker can be reduced by abstracting or goedelizing variables and by encoding variables in automata locations. The abstractions are tailored to the special tool we use, that is UPPAAL.
In this paper we illustrate our method using an implementation of ICPP scheduling in the Ravenscar kernel. The method is, however, applicable to a wide class of concurrent, real-time programs. In outline, the method is,

1. Extract information from a given concurrent, real-time Ada95 program in order to build a specification of the program in TLA+ as a network of communicating timed automata with data variables:

   (a) Extract task information from the full Ada95 implementation: how many concurrent tasks are there and how do they communicate? Each task will have its own control location variables, its own clock, and will share some data with other tasks. Identify the environment.

(b) For each individual task in the Ada95 program, use SPARK to define and verify pre- and post- conditions of its procedures.

(c) Define calls to the runtime environment of the program using procedure declarations and abstract own variables in SPARK.

(d) Use WCET tool, for example [18], or any available information about the runtime environment to verify worst case execution times for each procedure called. Each action is enabled when the task clock is equal to the action's WCET and taking an action transition resets the task clock.

(e) Extract control flow information for each sequential task of the full system. Model sequential control flow with a digraph whose nodes are control locations and edges are state changing TLA+ actions. The action granularity is determined by the placing of control locations, for example at the beginning and end of each procedure call.

2. Using information from 1.a to 1.d, define a timed automata in TLA+: each Ada95 task is represented by an initial condition, plus a set of actions. In every step one of the defined actions is taken or all stay data unchanged (a stuttering step).

3. Abstract this TLA+ specification into an abstract timed automata specification which can be analyzed in UPPAAL. The following types of refinement for TLA+ can be used:

   (a) simplify the control structure of a task by abstraction of actions

   (b) simplify Ada95 structured data variables by mapping them onto more abstract variables (Goedelizing composite data)

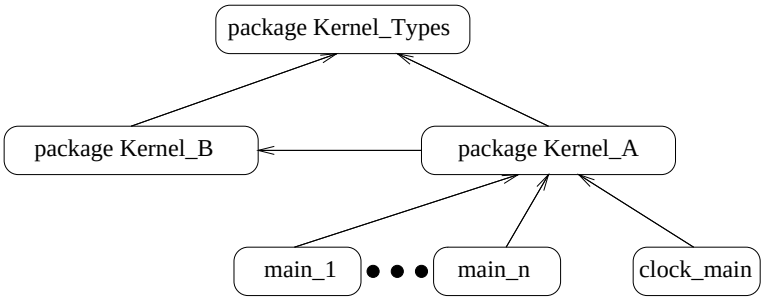
   (c) reduce the data range of clock variables, for example substitute monotonic clocks by periodic ones

4. Translate the abstract TLA specification into UPPAAL's timed automata input language and verify real-time requirements using the UPPAAL model-checker.

3 The Kernel and Environment

We demonstrate our method with a case study of an implementation of a kernel similar to [10] based on the Ravenscar Profile [7] implementing ICPP

scheduling [9].
The kernel operations are implemented in two packages, *Kernel_A* and *Kernel_B* and the complete kernel datastructure is stored in two variables, *Kernel_B.kb* and *Kernel_A.ks*. Package *Kernel_B* defines the procedures that cannot be written in Ada but must be implemented in machine code, for example context switching. *Kernel_B.kb* stores the part of the kernel datastructure that is only accessed and changed by the machine code operations in *Kernel_B*. It represents the run time environment of *Kernel_A*. We declare *Kernel_B.kb* as an abstract own variable [3] and we do not refine it. All kernel operations in package *Kernel_A*, like *delay_until*, *clock_tick* and operations for entering and leaving protected objects, are written in SPARK and call machine code procedures in *Kernel_B*. The SPARK package *Kernel_Types* defines the kernel types accessed in *Kernel_B* and accessed as well as manipulated in *Kernel_A*.
The subprogram *main_clock* is implemented in full Ada. It is part of the environment of our analyzed system and uses the Ada interrupt handler package to connect a clock interrupt with the procedure *Kernel_A.clock_tick*.
A set of user tasks $main_1$ to $main_n$ is written in SPARK and calls operations of *Kernel_A*. The relation of the packages and subprograms is illustrated below.



For simplicity and demonstration purpose we assume each of the tasks $main_i$ have the following form :

```
with Kernel_A,Kernel_B,Kernel_Types;
--# inherit Kernel_A,Kernel_B,Kernel_Types;
--# main_program;
procedure Main_1
--# global in out Kernel_A.ks,Kernel_B.kb;
--# derives Kernel_A.ks from Kernel_A.ks &
--#         Kernel_B.kb from Kernel_A.ks,Kernel_B.kb;
is
Next_Period : Kernel_Types.K_Time := 0;
Period : constant Kernel_Types.K_Time :=10;
Id : constant Kernel_Types.Task_Id := 1;
```

```
begin
  while True loop
   -- statements to be executed
   Next_Period := Next_Period + Period;
   Kernel_A.Delay_until(Next_Period,Id);
  end loop;
end Main_1;
```

The operation `Delay_until` is defined as below. The procedure `Update_state` updates the state of a task to `READY` or `SUSPENDED` depending on the delay time and the local time of the kernel `ks.ltime`.

```
 procedure Delay_until(next : in KT.K_Time; t : in KT.Task_Id)
 is
 curr : KT.Task_Id;
 begin
  Kernel_B.set_kernel_mode;
  curr := ks.run;
  Update_state(next,t);
  Dispatcher;
  Kernel_B.yield(curr,ks.run);
  Kernel_B.set_int_mask(ks.active_priority(ks.run));
  Kernel_B.clear_kernel_mode;
 end delay_until;

procedure Update_state(next : in KT.K_Time; t : in KT.Task_Id)
  --# global in out ks;
  --# derives ks from ks,next,t;
  --# post ( next>ks.ltime -> ((ks.current_state(t) = KT.SUSPENDED) and
  --#                                        (ks.delay_time(t)=next)))
  --# and ( next<=ks.ltime -> ((ks.current_state(t) = KT.READY) and
  --#                           ks~.delay_time(t) = ks.delay_time(t)));
is
begin
 if next > ks.ltime then
    ks.current_state(t)   := KT.SUSPENDED;
    ks.delay_time(t)      := next;
 else
    ks.current_state(t) := KT.READY;
 end if;
end Update_state;
```

The dispatcher chooses the task with highest active priority as task that is running next.

```
procedure Dispatcher
  --# global in out ks;
  --# derives ks from ks;
  --# pre ks.current_state(0) = KT.READY and ks.active_priority(0)=0;
  --# post (for all  I in KT.Task_Id =>
  --#         (for all J in KT.Task_Id =>
  --#             (((ks.active_priority(I) >= ks.active_priority(J)) or
  --#               (ks.current_state(J)=KT.SUSPENDED))
  --#               -> (ks.run=I))));
```

4 Modeling Tasks in TLA+

The computational model of TLA+ is a state machine specified in terms of an initial condition and a set of actions. At each step, an enabled action is taken and state variables are updated, or no action is taken and the variables of the system stay unchanged. We add timed behavior to this framework, as in the model of timed automata, using clock variables which are incremented as time passes, and time constraints in actions which govern when transitions may, or must, be taken. At the implementation level, state variables can use any of the data structures modeled in SPARK Ada. Using the model checker UPPAAL for verifying a TLA specification means restricting data types to those supported in UPPAAL timed automata. In practice the data are mapped on the UPPAAL data types integers and arrays of integers. Procedures in SPARK correspond to TLA actions, for example the action *disp* corresponds to procedure `Dispatcher`, action *upd* merges procedure `Update_state` and the `executed_statements`.

5 TLA Specification 1

We translate $main_i$ to the TLA specification module *Task1* below. An action in TLA is a conjunction of a precondition and a postcondition. The precondition is a predicate when the action is enabled. The postcondition describes how all variables of the specification change in the next state. A primed variable (for example a') means the value of a variable in the next state, an unprimed variable means the value the current state. The values of variables that do not change their value in the next state usually are described with UNCHANGED .
For example action $s1 - s2$ in specification *Task1* can be taken when (or has the precondition) $pc[i]$ = "s1" and $ks.run = i$. The postcondition defines the value of $pc[i]$ in the next state as "s2" and the values of the variables

$ks, clock_i$ as unchanged. The notation : $pc' = [pc \text{ EXCEPT } !.i = e]$ means $pc'[i] = e \wedge \forall x \neq i : pc'[x] = pc[x]$.
The predicate *urgent* is used in action *clock_tick* and defines when the action cannot be taken, and the clock variables $clock_i$ and $ks.ltime$ cannot be increased by one. The predicate formalizes our assumption of the timing behavior of the specification.
Control flow of an Ada program is implicit, it is convention that the statements are executed in the order they are written in the program. In the TLA specification we have to make this control flow explicit by introducing a control variable. In our specification we use a variable *pc[i]* for every user task i that ranges over control locations "s1", "s2" and "s3".
Execution time information is also implicit in the Ada program that may depend on the hardware used. We assume worst case execution times have been calculated, ideally with a supporting tool. Time is made explicit by using the concept of explicit clock variables to represent time [1]. For every task i one local clock variable $clock_i$ is added. When entering a new control location corresponding to a state where a task is executing, the clock is reset : $clock_i' = 0$.
We use a precondition $clock_i = wcet_i$ in a TLA action representing an Ada statement with worst case execution time $wcet_i$.
The pre and postconditions of every SPARK procedure [13, 3, 5], are translated to one TLA action. In our example the pre and postconditions in SPARK are translated to TLA on a pure syntactical basis.
In the TLA specification we also make all information about variables explicit that are inherited from the packages *Kernel_Types* and *Kernel_A* by $main_i$.
TLA is an untyped logic. Type information can be expressed as an invariant of the specification. An example of a type invariant is $\Box pc[i] \in \{$"s1", "s2", "s3"$\}$ that means $pc[i]$ always has one of the values "s1","s2" or "s3".

module *Task1*

EXTENDS *Naturals*

parameters

$ks, clock_i, pc$: VARIABLE
$MAX\_TASK, MAX\_TIME, MAX\_ID, MAX\_PRI,$
$NULL\_TASK, IDLE\_TASK$: CONSTANT
$i, period_i, wcet_i$: CONSTANT

$MAX\_TASK, MAX\_TIME, MAX\_ID, MAX\_PRI \in Nat$
$NULL\_TASK \notin 0..MAX\_TASK, IDLE\_TASK = 0, period_i > 0$

predicate

$$
\begin{array}{lcl}
\mathit{Init} & \triangleq & \land\ \mathit{clock}_i = 0 \\
& & \land\ \forall i \in 0..\mathit{MAX\_TASK}\ :\ \mathit{pc}[i] = \text{“s1”} \\
& & \land\ \mathit{ks.run} = \mathit{MAX\_TASK} \\
& & \land\ \mathit{ks.ltime} = 0 \\
& & \land\ \forall i \in 0..\mathit{MAX\_TASK}\ :\ \land\ \mathit{ks.current\_state}[i] = \text{“READY”} \\
& & \qquad\qquad\qquad\qquad\ \land\ \mathit{ks.active\_priority}[i] = i \\
& & \qquad\qquad\qquad\qquad\ \land\ \mathit{ks.delay\_time}[i] = 0 \\
\mathit{urgent} & \triangleq & \lor\ \mathit{pc}[i] = \text{“s1”} \land \mathit{ks.run} = i \\
& & \lor\ \mathit{pc}[i] = \text{“s2”} \land \mathit{clock}_i = \mathit{wcet}_i \\
& & \lor\ \mathit{pc}[i] = \text{“s3”} \\
\mathit{goto}_i(a, e) & \triangleq & \mathit{pc}[i] = a \land \mathit{pc}' = [\mathit{pc}\ \text{EXCEPT}\ !.i = e]
\end{array}
$$

actions

$$
\begin{array}{lcl}
s1 - s2 & \triangleq & \land\ \mathit{goto}_i(\text{“s1”}, \text{“s2”}) \land \mathit{ks.run} = i \\
& & \land\ \text{UNCHANGED}\ \langle \mathit{ks}, \mathit{clock}_i \rangle \\
\mathit{upd} & \triangleq & \land\ \mathit{goto}_i(\text{“s2”}, \text{“s3”}) \land \mathit{clock}_i = \mathit{wcet}_i \\
& & \land\ \textbf{if}\ \mathit{ks.delay\_time}[i] + \mathit{period}_i > \mathit{ks.ltime} \\
& & \quad \textbf{then}\ \land\ \mathit{ks.current\_state}' = [\mathit{ks.current\_state}\ \text{EXCEPT}\ !.\mathit{ks.run} = \text{“SUSPENDED”}] \\
& & \qquad\ \land\ \mathit{ks.delay\_time}' = [\mathit{ks.delay\_time}\ \text{EXCEPT}\ !.i = !.i + \mathit{period}_i] \\
& & \quad \textbf{else}\ \land\ \mathit{ks.current\_state}' = \mathit{ks.current\_state} \\
& & \qquad\ \land\ \mathit{ks.delay\_time}' = \mathit{ks.delay\_time} \\
& & \land\ \text{UNCHANGED}\ \langle \mathit{ks.run}, \mathit{ks.ltime}, \mathit{ks.active\_priority}, \mathit{clock}_i \rangle \\
s2 - s1 & \triangleq & \land\ \mathit{goto}_i(\text{“s2”}, \text{“s1”}) \land \mathit{clock}_i < \mathit{wcet}_i \land \mathit{ks.run} \neq i \\
& & \land\ \text{UNCHANGED}\ \langle \mathit{ks}, \mathit{clock}_i \rangle \\
\mathit{disp} & \triangleq & \land\ \mathit{goto}_i(\text{“s3”}, \text{“s1”}) \\
& & \land\ \mathit{ks.run}' = \text{CHOOSE}\ i\ :\ \land\ i \in 0..\mathit{MAX\_TASK} \\
& & \qquad\qquad\qquad\qquad\ \land\ \mathit{ks.current\_state}[i] = \text{“READY”} \\
& & \qquad\qquad\qquad\qquad\ \land\ \forall j \in 0..\mathit{MAX\_TASK}\ : \\
& & \qquad\qquad\qquad\qquad\quad \lor\ \mathit{ks.current\_state}[j] = \text{“SUSPENDED”} \\
& & \qquad\qquad\qquad\qquad\quad \lor\ \land\ \mathit{ks.current\_state}[j] = \text{“READY”} \\
& & \qquad\qquad\qquad\qquad\qquad \land\ \mathit{ks.active\_priority}[j] \leq \mathit{ks.active\_priority}[i] \\
& & \land\ \mathit{clock}_i' = 0 \\
& & \land\ \text{UNCHANGED}\ \langle \mathit{ks.current\_state}, \mathit{ks.delay\_time}, \\
& & \qquad\qquad\qquad\qquad \mathit{ks.active\_priority}, \mathit{ks.ltime} \rangle
\end{array}
$$

$$
\begin{array}{lll}
clock\_tick & \triangleq & \land\ \neg urgent \\
 & & \land\ ks.ltime' = ks.ltime + 1 \\
 & & \land\ \textbf{if}\ pc[i] = \text{“s2”}\ \ \textbf{then}\ clock'_i = clock_i + 1 \\
 & & \qquad\qquad\qquad\ \ \textbf{else}\ \ clock'_i = clock_i \\
 & & \land\ \forall i \in 1..MAX\_TASK\ : \\
 & & \quad \textbf{if}\ (ks.ltime + 1) = ks.delay\_time[i] \\
 & & \qquad \textbf{then}\ ks.current\_state' = [ks.current\_state\ \text{EXCEPT}\ !.i = \text{“READY”}] \\
 & & \qquad \textbf{else}\ \ ks.current\_state' = ks.current\_state \\
 & & \land\ ks.run' = \text{CHOOSE}\ i\ :\ \land\ i \in 0..MAX\_TASK \\
 & & \qquad\qquad\qquad\qquad\qquad\ \land\ ks.delay\_time[i] \leq ks.ltime + 1 \\
 & & \qquad\qquad\qquad\qquad\qquad\ \land\ \forall j \in 0..MAX\_TASK\ : \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \lor\ \land\ ks.delay\_time[j] \leq ks.ltime + 1 \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\ \land\ ks.active\_priority[j] \leq ks.active\_priority[i] \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \lor\ \land\ ks.delay\_time[j] > ks.ltime + 1 \\
 & & \land\ \text{UNCHANGED}\ \langle ks.delay\_time, \\
 & & \qquad\qquad\qquad\ ks.active\_priority, ks.current\_state\rangle
\end{array}
$$

$$
action\_step \triangleq s1 - s2 \lor upd \lor s2 - s1 \lor disp
$$

temporal

$$
Spec \triangleq Init \land \Box[action\_step \lor clock\_tick]_{\langle clock_i, pc, ks\rangle}
$$

Now we formalize the relation between the values of $ks.delay\_time$, $ks.ltime$ and the control locations of $pc[i]$. This will help us to encode the relation $ks.delay\_time$ and $ks.ltime$ in control locations in the next abstraction. The invariants below hold on specification *Task1*.

module *InvTask1*

$$
\begin{array}{lll}
Inv1 & \triangleq & \Box\land\ pc[i] = \text{“s1”} \\
 & & \quad \land\ ks.delay\_time[i] \leq ks.ltime \\
 & & \quad \Rightarrow ks.current\_state[i] = \text{“READY”} \\
Inv2 & \triangleq & \Box\land\ pc[i] = \text{“s1”} \\
 & & \quad \land\ ks.delay\_time[i] > ks.ltime \\
 & & \quad \Rightarrow ks.current\_state[i] = \text{“SUSPENDED”} \\
Inv3 & \triangleq & \Box pc[i] = \text{“s2”} \\
 & & \quad \Rightarrow ks.current\_state[i] = \text{“READY”} \\
Inv4 & \triangleq & \Box pc[i] = \text{“s3”} \land ks.delay\_time[i] > ks.ltime \\
 & & \quad \Rightarrow ks.current\_state[i] = \text{“SUSPENDED”} \\
Inv5 & \triangleq & \Box pc[i] = \text{“s3”} \land ks.delay\_time[i] \leq ks.ltime \\
 & & \quad \Rightarrow ks.current\_state[i] = \text{“READY”}
\end{array}
$$

$$
\begin{array}{lcl}
Inv6 & \triangleq & \Box pc[i] = \text{“s2”} \\
 & & \quad \Rightarrow ks.delay\_time[i] \leq ks.ltime \\
Inv7 & \triangleq & \Box pc[i] = \text{“s3”} \\
 & & \quad \Rightarrow ks.delay\_time[i] \leq ks.ltime \vee ks.delay\_time[i] > ks.ltime \\
Inv8 & \triangleq & \Box pc[i] = \text{“s1”} \\
 & & \quad \Rightarrow ks.delay\_time[i] \leq ks.ltime \vee ks.delay\_time[i] > ks.ltime \\
Inv & \triangleq & Inv1 \wedge Inv2 \wedge Inv3 \wedge Inv4 \wedge Inv5 \wedge Inv6 \wedge Inv7 \wedge Inv8
\end{array}
$$

$InvTask1.Inv6$,$InvTask1.Inv7$ and $InvTask1.Inv8$ directly follow from invariants $InvTask1.Inv1$ to $InvTask1.Inv5$, because $ks.delay\_time[i] \leq ks.ltime$ always corresponds to a state with $ks.current\_state[i]$ = “READY” and $ks.delay\_time[i] > ks.ltime$ always corresponds to a state with $ks.current\_state[i]$ = “SUSPENDED”. The invariants make explicit that there is no one to one relation between the locations “s1”, “s2” and “s3” and the value of $ks.current\_state[i]$.

6 Abstraction 2

In specification *Task2* we encode the state of $ks.current\_state[i]$ in the locations. This will make the variable $ks.current\_state$ redundant.
In specification *Task2* we encode the values of $ks.current\_state$ in locations by introducing two new locations “s0” and “s4”. The invariants below hold on specification *Task2*.

module *InvTask2*

$$
\begin{array}{lcl}
Inv1 & \triangleq & \Box pc[i] = \text{“s1”} \\
 & & \quad \Rightarrow ks.ltime \geq ks.delay\_time[i] \\
Inv2 & \triangleq & \Box pc[i] = \text{“s2”} \\
 & & \quad \Rightarrow ks.ltime \geq ks.delay\_time[i] \\
Inv3 & \triangleq & \Box pc[i] = \text{“s3”} \\
 & & \quad \Rightarrow ks.ltime < ks.delay\_time[i] \\
Inv4 & \triangleq & \Box pc[i] = \text{“s4”} \\
 & & \quad \Rightarrow ks.ltime \geq ks.delay\_time[i] \\
Inv5 & \triangleq & \Box pc[i] = \text{“s0”} \\
 & & \quad \Rightarrow ks.ltime < ks.delay\_time[i] \\
Inv & \triangleq & Inv1 \wedge Inv2 \wedge Inv3 \wedge Inv4 \wedge Inv5
\end{array}
$$

Because $ks.ltime \geq ks.delay\_time[i]$ corresponds to a state where the task is ready to start running, and $ks.ltime < ks.delay\_time[i]$ corresponds to a state where the task is suspended e.g *not* ready to execute, the variable *ks.current_state* does not add any extra information to the specification.
In specification *Task1* there is only one transition from "s2" to "s3", defined by the TLA action $Task1.upd$. This transition is replaced by two transitions in *Task2*, one transition from "s2" to "s3" and another transition from "s2" to "s4", defined by the TLA action $Task2.upd$.
While in specification *Task1* there is a direct transition from location $pc[i] =$ "s3" to location $pc[i] =$ "s1", defined by action $Task1.disp$, in specification *Task2* there is only a transition from $pc[i] =$ "s3" to $pc[i] =$ "s0" to $pc[i] =$ "s1", defined by the actions $Task2.disp$ and $Task2.clock\_tick$. (The condition in action clock_tick, $ks.ltime + 1 = ks.delay\_time[i]$ can only happen when $pc[i] =$ "s0").
An informal graphical description of specification *Task2* is shown in figure 1.

module *Task2*

EXTENDS *Naturals*

parameters

ks : VARIABLE
$clock_i, pc$: VARIABLE
$MAX\_TASK, MAX\_TIME, MAX\_ID, MAX\_PRI,$
$NULL\_TASK, IDLE\_TASK$: CONSTANT
$i, period_i, wcet_i$: CONSTANT

$MAX\_TASK, MAX\_TIME, MAX\_ID, MAX\_PRI \in Nat$
$NULL\_TASK \notin 0..MAX\_TASK, IDLE\_TASK = 0, period_i > 0$

predicate

$$
\begin{array}{lcl}
Init & \triangleq & \land\ clock_i = 0 \\
 & & \land\ \forall i \in 0..MAX\_TASK\ :\ pc[i] = \text{"s1"} \\
 & & \land\ ks.run = MAX\_TASK \\
 & & \land\ ks.ltime = 0 \\
 & & \land\ \forall i \in 0..MAX\_TASK\ :\ \land\ ks.active\_priority[i] = i \\
 & & \qquad\qquad\qquad\qquad\ \land\ ks.delay\_time[i] = 0 \\
urgent & \triangleq & \lor\ pc[i] = \text{"s1"} \land ks.run = i \\
 & & \lor\ pc[i] = \text{"s2"} \land clock_i = wcet_i \\
 & & \lor\ pc[i] = \text{"s3"} \\
 & & \lor\ pc[i] = \text{"s4"} \\
goto_i(a, e) & \triangleq & pc[i] = a \land pc' = [pc \text{ EXCEPT } !.i = e]
\end{array}
$$

actions

$s1 - s2 \triangleq \land\ goto_i(\text{“s1”}, \text{“s2”}) \land ks.run = i$
$\land$ UNCHANGED $\langle ks, clock_i \rangle$

$upd \triangleq \land\ pc[i] = \text{“s2”} \land clock_i = wcet_i$
$\land$ **if** $ks.delay\_time[i] + period_i > ks.ltime$
then $\land\ ks.delay\_time' = [ks.delay\_time$ EXCEPT $!.i = .!\,i + period_i]$
$\land\ pc' = [pc$ EXCEPT $!.i = \text{“s3”}]$
else $\land\ ks.delay\_time' = ks.delay\_time$
$\land\ pc' = [pc$ EXCEPT $!.i = \text{“s4”}]$
$\land$ UNCHANGED $\langle ks.run, ks.ltime, ks.active\_priority, clock_i \rangle$

$s2 - s1 \triangleq \land\ goto_i(\text{“s2”}, \text{“s1”}) \land ks.run \neq i \land clock_i < wcet_i$
$\land$ UNCHANGED $\langle ks, clock_i \rangle$

$disp \triangleq \land\ goto_i(\text{“s3”}, \text{“s0”}) \lor goto_i(\text{“s4”}, \text{“s1”})$
$\land\ ks.run' =$
CHOOSE $i : \land\ i \in 0..MAX\_TASK$
$\land\ pc[j] = \text{“s1”} \lor pc[j] = \text{“s2”} \lor pc[j] = \text{“s4”}$
$\land\ \forall j \in 0..MAX\_TASK :$
$\lor \land\ pc[j] = \text{“s1”} \lor pc[j] = \text{“s2”} \lor pc[j] = \text{“s4”}$
$\land\ ks.active\_priority[j] \leq ks.active\_priority[i]$
$\lor\ pc[j] = \text{“s3”} \lor pc[j] = \text{“s0”}$
$\land\ clock_i' = 0$
$\land$ UNCHANGED $\langle ks.delay\_time, ks.active\_priority, ks.ltime \rangle$

$clock\_tick \triangleq \land\ \neg urgent$
$\land\ ks.ltime' = ks.ltime + 1$
$\land\ \forall i \in 0..MAX\_TASK :$ **if** $pc[i] = \text{“s2”}$ **then** $clock_i' = clock_i + 1$
else $clock_i' = clock_i$
$\land\ \forall i \in 0..MAX\_TASK :$
if $(ks.ltime + 1) = ks.delay\_time[i]$
then $pc' = [pc$ EXCEPT $!.i = \text{“s1”}]$
else $pc' = pc$
$\land\ ks.run' =$ CHOOSE $i : \land\ i \in 0..MAX\_TASK$
$\land\ ks.delay\_time[i] \leq ks.ltime + 1$
$\land\ \forall j \in 0..MAX\_TASK :$
$\lor \land\ ks.delay\_time[j] \leq ks.ltime + 1$
$\land\ ks.active\_priority[j] \leq ks.active\_priority[i]$
$\lor \land\ ks.delay\_time[i] > ks.ltime + 1$
$\land$ UNCHANGED $\langle ks.active\_priority, ks.delay\_time \rangle$

$action\_step \triangleq s1 - s2 \lor upd \lor s2 - s1 \lor disp$

temporal

$Spec \triangleq Init \land \Box[action\_step \lor clock\_tick]_{\langle clock_i, pc, ks \rangle}$

$$
\begin{array}{lll}
Update - running - task & \triangleq & ks.run' = \\
 & & \text{CHOOSE } i : \wedge\ i \in 0..MAX\_TASK \\
 & & \qquad \wedge\ pc[j] = \text{“s1”} \vee pc[j] = \text{“s2”} \vee pc[j] = \text{“s4”} \\
 & & \qquad \wedge\ \forall j \in 0..MAX\_TASK\ : \\
 & & \qquad\quad \vee \wedge\ pc[j] = \text{“s1”} \vee pc[j] = \text{“s2”} \vee pc[j] = \text{“s4”} \\
 & & \qquad\quad\quad \wedge\ ks.active\_priority[j] \leq ks.active\_priority[i] \\
 & & \qquad\quad \vee\ pc[j] = \text{“s3”} \vee pc[j] = \text{“s0”} \\
 & & clock'_i = 0
\end{array}
$$

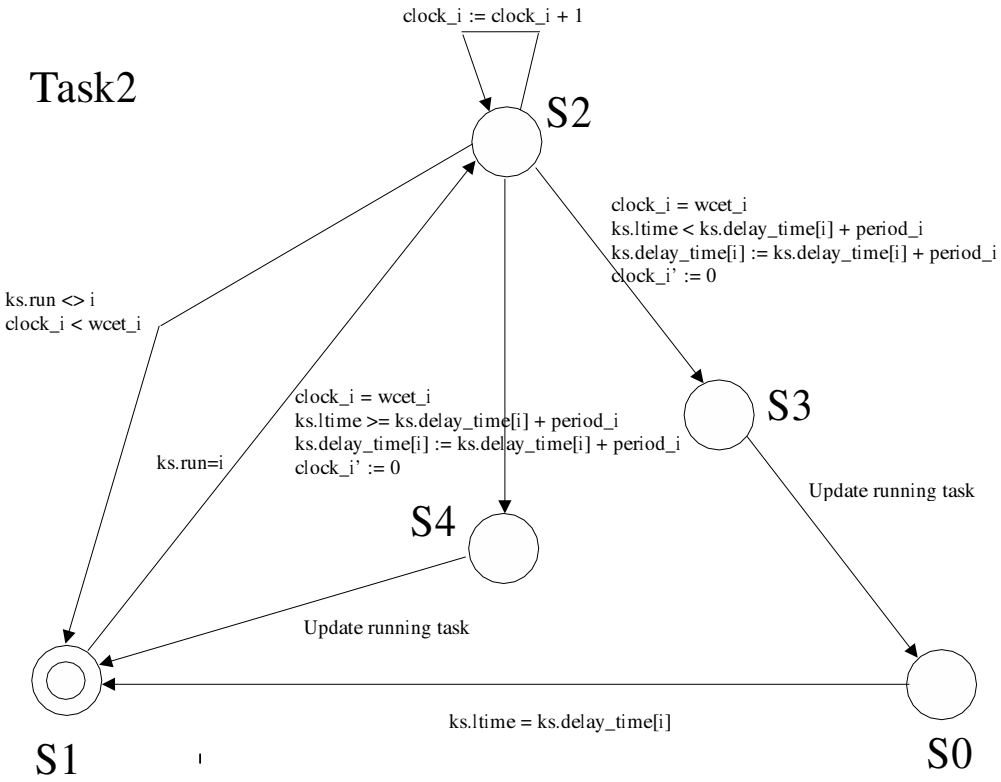


Figure 1: Task2

module *Task1-implements-Task2*

$I \triangleq$ **instance** *Task*1 **with** $I.ks \leftarrow ks, I.pc \leftarrow pc$
$A \triangleq$ **instance** *Task*2 **with** $A.ks \leftarrow ks, A.pc \leftarrow pc$
Refinement Mapping:

$$\begin{array}{lll} \overline{A.pc[i]} \triangleq \textbf{case} & I.pc[i] = \text{“s1”} \wedge ks.ltime \geq I.ks.delay\_time[i] & \rightarrow \text{“s1”}\Box \\ & I.pc[i] = \text{“s1”} \wedge ks.ltime < I.ks.delay\_time[i] & \rightarrow \text{“s0”}\Box \\ & I.pc[i] = \text{“s2”} & \rightarrow \text{“s2”}\Box \\ & I.pc[i] = \text{“s3”} \wedge ks.ltime < I.ks.delay\_time[i] & \rightarrow \text{“s3”}\Box \\ & I.pc[i] = \text{“s3”} \wedge ks.ltime \geq I.ks.delay\_time[i] & \rightarrow \text{“s4”} \end{array}$$

PROOF SKETCH*::*
$\langle 0 \rangle 1.\ I.Init \Rightarrow \overline{A.Init}$
This is a direct inclusion.
$\langle 0 \rangle 2.\ I.s1 - s2 \Rightarrow \overline{A.s1 - s2}$
This is a direct inclusion.
$\langle 0 \rangle 3.\ I.upd \Rightarrow \overline{A.upd}$
This inclusion follows directly by using the refinement mapping and substituting $pc[i]$ *with* $\overline{A.pc[i]}$.
$\langle 0 \rangle 4.\ I.s2 - s1 \wedge InvTask1.Inv \wedge InvTask2.Inv \Rightarrow \overline{A.s2 - s1}$
This is a direct inclusion.
$\langle 0 \rangle 5.\ I.disp \Rightarrow \overline{A.disp}$
This inclusion follows from the refinement mapping of $pc[i]$. *If* $pc[i] = \text{“s1”} \vee pc[i] = \text{“s2”} \vee pc[i] = \text{“s4”}$ *in spec* Task2 *then* $ks.current\_state[i] = \text{“READY”}$ *in spec Task1. If* $pc[i] = \text{“s0”}$ *then* $ks.current\_state[i] = \text{“SUSPENDED”}$.
$\langle 0 \rangle 6.\ I.clock\_tick \wedge InvTask1.Inv \wedge InvTask2.Inv \Rightarrow \overline{A.clock\_tick}$
The condition $ks.ltime + 1 = ks.delay\_time[i]$ *in action* $Task1.clock\_tick$ *can only be taken if* $pc[i] = \text{“s1”}$. *With the refinement mapping of pc it follow that in this state of spec* Task1 *the next state of* Task2 *is defined bypc'*$[i] = \text{“s1”}$. *The state* $ks.current\_state[i] = \text{“READY”}$ *in specification* Task1 *is equivalent to* $ks.ltime \geq ks.delay\_time[i]$ *and corresponds to* $pc[i] = \text{“s1”} \vee pc[i] = \text{“s2”} \vee pc[i] = \text{“s4”}$ *in specification* Task2.

7 Abstraction 4

In specification *Task3* we eliminate the variable *ks.run* and introduce the variable w. The value of w is always given by the invariant $InvTask3.Inv6$. This allows us to eliminate the the CHOOSE operator in the actions *clock_tick* and *disp*. (Usually this operator is not available in the input language of a model checker.) The following holds : $(ks.run = i) \equiv w \leq 2.2^i - 1$, $(ks.run \neq i) \equiv w > 2.2^i - 1$ as we will prove below.

module *w-property*

Definition

$\forall i : 1 \leq i \leq Max\_Task : s[i] \triangleq$ **case** $pc[i] = \text{“s0”} \vee pc[i] = \text{“s3”} \rightarrow 0\Box$
$pc[i] = \text{“s1”} \vee pc[i] = \text{“s2”} \vee pc[i] = \text{“s4”} \rightarrow 1$

$s[0] \triangleq 1$

$w \triangleq \sum_{0 \leq i \leq Max\_Task} 2^i.s[i]$

$\langle 1 \rangle 1.\ ks.run = i \Rightarrow w \leq 2.2^i - 1$

$\langle 2 \rangle 1.\ \wedge\ ks.run = i$
$\wedge\ w = \sum_{0 \leq k \leq Max\_Task} 2^k.s[k]$
$\Rightarrow$
$\wedge\ \forall k \in i+1 \ldots Max\_Task : pc[k] = \text{“s3”} \vee pc[k] = \text{“s0”}$
$\wedge\ w = \sum_{0 \leq k \leq Max\_Task} 2^k.s[k]$

$\langle 2 \rangle 2.\ \wedge\ \forall k \in i+1 \ldots Max\_Task : pc[k] = \text{“s3”} \vee pc[k] = \text{“s0”}$
$\wedge\ w = \sum_{0 \leq k \leq Max\_Task} 2^k.s[k]$
$\Rightarrow$
$w = \sum_{0 \leq k \leq i} 2^k.s[k]$

$\langle 2 \rangle 3.\ w = \sum_{0 \leq k \leq i} 2^k.s[k]$
$\Rightarrow$
$w \leq 2.2^i - 1$

The abstraction of *Task2* has the following properties :

module *InvTask3*

$Inv1 \triangleq \Box pc[i] = \text{“s1”} \Rightarrow ks.ltime \geq ks.delay\_time[i]$

$Inv2 \triangleq \Box pc[i] = \text{“s2”} \Rightarrow ks.ltime \geq ks.delay\_time[i]$

$Inv3 \triangleq \Box pc[i] = \text{“s3”} \Rightarrow ks.ltime < ks.delay\_time[i]$

$Inv4 \triangleq \Box pc[i] = \text{“s4”} \Rightarrow ks.ltime \geq ks.delay\_time[i]$

$Inv5 \triangleq \Box pc[i] = \text{“s0”} \Rightarrow ks.ltime < ks.delay\_time[i]$

$Inv6 = \Box w = \sum_{i=0}^{MAX\_TASK} 2^i.$**if** $\vee\ pc[i] = \text{“s1”}$
$\vee\ pc[i] = \text{“s2”}$
$\vee\ pc[i] = \text{“s4”}$
then 1
else 0

$Inv \triangleq Inv1 \land Inv2 \land Inv3 \land Inv4 \land Inv5 \land Inv6$

module *Task3*

EXTENDS *Naturals*

parameters

$ks, w, clock_i, pc$: VARIABLE
$MAX\_TASK, MAX\_TIME, MAX\_ID, MAX\_PRI,$
$NULL\_TASK, IDLE\_TASK$: CONSTANT
$i, period_i, wcet_i$: CONSTANT

$MAX\_TASK, MAX\_TIME, MAX\_ID, MAX\_PRI \in Nat$
$NULL\_TASK \notin 0..MAX\_TASK, IDLE\_TASK = 0$

predicate

$$
\begin{array}{lcl}
Init & \triangleq & \land\ clock_i = 0 \\
 & & \land\ \forall i \in 0..MAX\_TASK\ :\ pc[i] = \text{“s1”} \\
 & & \land\ ks.ltime = 0 \\
 & & \land\ \forall i \in 0..MAX\_TASK\ :\ \land\ ks.active\_priority[i] = i \\
 & & \qquad\qquad\ \land\ ks.delay\_time[i] = 0 \\
 & & \land\ w = \sum_{i=0}^{MAX\_TASK} 2^i \\
urgent & \triangleq & \lor\ pc[i] = \text{“s1”} \land w \leq 2.2^i - 1 \\
 & & \lor\ pc[i] = \text{“s2 ”} \land clock_i = wcet_i \\
 & & \lor\ pc[i] = \text{“s3”} \\
 & & \lor\ pc[i] = \text{“s4”} \\
goto_i(a, e) & \triangleq & pc[i] = a \land pc' = [pc \text{ EXCEPT } !.i = e]
\end{array}
$$

actions

$$
\begin{array}{lcl}
s1 - s2 & \triangleq & \land\ goto_i(\text{“s1”}, \text{“s2”}) \land w \leq 2.2^i - 1 \\
 & & \land\ \text{UNCHANGED } \langle clock_i, ks, w \rangle \\
upd & \triangleq & \land\ pc[i] = \text{“s2”} \land clock_i = wcet_i \\
 & & \land\ \textbf{if } ks.ltime < next\_period_i + period_i \\
 & & \quad \textbf{then } \land\ ks.delay\_time' = [ks.delay\_time \text{ EXCEPT } !.i = !.i + period_i] \\
 & & \qquad\qquad\ \land\ pc' = [pc \text{ EXCEPT } !.i = \text{“s3”}] \\
 & & \qquad\qquad\ \land\ w' = w - 2^i \\
 & & \quad \textbf{else } \land\ ks.delay\_time' = ks.delay\_time \\
 & & \qquad\qquad\ \land\ pc' = [pc \text{ EXCEPT } !.i = \text{“s4”}] \\
 & & \qquad\qquad\ \land\ w' = w \\
 & & \land\ clock_i' = 0 \\
 & & \land\ \text{UNCHANGED } \langle ks.ltime, ks.active\_priority \rangle
\end{array}
$$

$s2 - s1 \triangleq \land\ goto_i(\text{“s2”}, \text{“s1”}) \land w > 2.2^i - 1 \land clock_i < wcet_i$
$\land$ UNCHANGED $\langle ks, clock_i, w \rangle$

$disp \triangleq \land\ goto_i(\text{“s3”}, \text{“s0”}) \lor goto_i(\text{“s4”}, \text{“s1”})$
$\land$ UNCHANGED $\langle ks, clock_i \rangle$

$clock\_tick \triangleq \land \neg urgent$
$\land\ ks.ltime' = ks.ltime + 1$
$\land\ \forall i \in 1..MAX\_TASK\ :$ **if** $pc[i] = \text{“s2”}$ **then** $clock'_i = clock_i + 1$
else $clock'_i = clock_i$
$\land\ \forall i \in 1..MAX\_TASK\ :$
if $(ks.ltime + 1) = ks.delay\_time[i]$
then $\land\ pc' = [pc$ EXCEPT $!.i = \text{“s1”}]$
else $\land\ pc' = pc$
$\land\ w' = \sum_{i=0}^{MAX\_TASK} 2^i$.**if** $\lor\ pc'[i] = \text{“s1”}$ **then** 1
$\lor\ pc'[i] = \text{“s2”}$
$\lor\ pc'[i] = \text{“s4”}$
else 0
$\land$ UNCHANGED $\langle ks.active\_priority, ks.delay\_time \rangle$

$action\_step \triangleq s1 - s2 \lor upd \lor s2 - s1 \lor disp$

temporal

$Spec \triangleq Init \land \Box[action\_step \lor clock\_tick]_{\langle clock_i, pc, w, ks \rangle}$

module *Task2-implements-Task3*

PROOF SKETCH*:*
$\langle 0 \rangle 1$. *predicate* urgent
PROOF SKETCH*:Inclusion follows with* $ks.run = i \Rightarrow w \leq 2.2^i - 1$
$\langle 0 \rangle 2$. $I.Init \Rightarrow \overline{A.Init}$
PROOF SKETCH*:Inclusion follows with* $ks.run = i \Rightarrow w \leq 2.2^i - 1$
$\langle 0 \rangle 3$. $I.s1 - s2 \Rightarrow \overline{A.s1 - s2}$
PROOF SKETCH*:Inclusion follows with* $ks.run = i \Rightarrow w \leq 2.2^i - 1$
$\langle 0 \rangle 4$. $I.upd \Rightarrow \overline{A.upd}$
PROOF SKETCH*:Inclusion follows with InvTask*3.*Inv*6.
$\langle 0 \rangle 5$. $I.s2 - s1 \Rightarrow \overline{A.s2 - s1}$
PROOF SKETCH*:Inclusions follows with* $ks.run \neq i \Rightarrow w > 2.2^i - 1$
$\langle 0 \rangle 6$. $I.clock\_tick \Rightarrow \overline{A.clock\_tick}$
PROOF SKETCH*:Inclusion follows with InvTask*3.*Inv*6.

8 Abstraction 4

In the abstraction step from specification *Task3* to *Task4* we substitute monotonic clocks by periodic clocks. Model checking of specifications with variables of large data ranges like the Ada system clock is very expensive. The purpose of the monotonic clock *ks.ltime* is to define the delay time of a task in location "s3". To get a specification with less states that preserves the property of the delay time, we eliminate the monotonic clock variable *ks.ltime* and introduce a clock variable $periodic\_clock_i$ for every task i. The clock variable *periodic_clock* has values in $0..period_i$ and is reset periodically. As a consequence of substituting *ks.ltime* we do not need the array *delay_time* any more.
The resulting specification *Task4* is given below.

module *Task4*

EXTENDS *Naturals*

parameters

$ks, w, clock_i, periodic\_clock_i, pc$: VARIABLE
$MAX\_TASK, MAX\_TIME, MAX\_ID, MAX\_PRI,$
$NULL\_TASK, IDLE\_TASK$: CONSTANT
$i, period_i, wcet_i$: CONSTANT

$MAX\_TASK, MAX\_TIME, MAX\_ID, MAX\_PRI \in Nat$
$NULL\_TASK \notin 0..MAX\_TASK, IDLE\_TASK = 0$

predicate

$$\begin{aligned}
Init \triangleq\ & \wedge\ clock_i = 0 \wedge periodic_clock_i = 0 \\
& \wedge\ \forall i \in 0..MAX_TASK\ :\ pc[i] = \text{“s1”} \\
& \wedge\ \forall i \in 0..MAX_TASK\ :\ ks.active_priority[i] = i \\
urgent \triangleq\ & \vee\ pc[i] = \text{“s1”} \wedge w \leq 2.2^i - 1 \\
& \vee\ pc[i] = \text{“s2”} \wedge clock_i = wcet_i \\
& \vee\ pc[i] = \text{“s3”} \\
& \vee\ pc[i] = \text{“s4”} \\
goto_i(a, e) \triangleq\ & pc[i] = a \wedge pc' = [pc \text{ EXCEPT } !.i = e]
\end{aligned}$$

actions

$$\begin{aligned}
s1 - s2 \triangleq\ & \wedge\ goto_i(\text{“s1”}, \text{“s2”}) \wedge w \leq 2.2^i - 1 \\
& \wedge\ \text{UNCHANGED } \langle clock_i, periodic_clock_i, w, ks.active_priority \rangle
\end{aligned}$$

$$
\begin{array}{lcl}
upd & \triangleq & \land\ pc[i] = \text{“s2”} \land clock_i = wcet_i \\
& & \land\ \textbf{if}\ periodic\_clock_i < period_i \\
& & \quad \textbf{then} \land\ pc' = [pc\ \text{EXCEPT}\ !.i = \text{“s3”}] \\
& & \qquad \land\ w' = w - 2^i \\
& & \qquad \land\ periodic\_clock'_i = periodic\_clock_i \\
& & \quad \textbf{else}\ \land\ pc' = [pc\ \text{EXCEPT}\ !.i = \text{“s4”}] \\
& & \qquad \land\ w' = w \\
& & \qquad \land\ periodic\_clock'_i = periodic\_clock_i \% period_i \\
& & \land\ clock'_i = 0 \\
& & \land\ \text{UNCHANGED}\ \langle ks.active\_priority \rangle \\
s2 - s1 & \triangleq & \land\ goto_i(\text{“s2”}, \text{“s1”}) \land w > 2.2^i - 1 \land clock_i < wcet_i \\
& & \land\ \text{UNCHANGED}\ \langle clock_i, periodic\_clock_i, w, ks.active\_priority \rangle \\
disp & \triangleq & \land\ goto_i(\text{“s3”}, \text{“s0”}) \lor goto_i(\text{“s4”}, \text{“s1”}) \\
& & \land\ \text{UNCHANGED}\ \langle clock_i, periodic\_clock_i, ks.active\_priority \rangle \\
clock\_tick & \triangleq & \land\ \neg urgent \\
& & \land\ \forall i \in 1..MAX\_TASK\ : \\
& & \quad \textbf{if}\ pc[i] = \text{“s2”}\ \textbf{then}\ clock'_i = clock_i + 1 \\
& & \qquad\qquad \textbf{else}\ \ clock'_i = clock_i \\
& & \land\ \textbf{if}\ periodic\_clock_i = period_i \\
& & \quad \textbf{then} \land\ pc' = [pc\ \text{EXCEPT}\ !.i = \text{“s1”}] \\
& & \qquad \land\ periodic\_clock'_i = 0 \\
& & \qquad \land\ w' = \sum_{i=0}^{MAX\_TASK} 2^i.\textbf{if} \lor\ pc'[i] = \text{“s1”}\ \textbf{then}\ 1 \\
& & \qquad\qquad\qquad\qquad\qquad\qquad \lor\ pc'[i] = \text{“s2”} \\
& & \qquad\qquad\qquad\qquad\qquad\qquad \lor\ pc'[i] = \text{“s4”} \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ 0 \\
& & \quad \textbf{else}\ \ \land\ pc' = pc \\
& & \qquad \land\ periodic\_clock'_i = periodic\_clock_i + 1 \\
& & \qquad \land\ w' = \sum_{i=0}^{MAX\_TASK} 2^i.\textbf{if} \lor\ pc'[i] = \text{“s1”}\ \textbf{then}\ 1 \\
& & \qquad\qquad\qquad\qquad\qquad\qquad \lor\ pc'[i] = \text{“s2”} \\
& & \qquad\qquad\qquad\qquad\qquad\qquad \lor\ pc'[i] = \text{“s4”} \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ 0 \\
& & \land\ \text{UNCHANGED}\ \langle ks.active\_priority, clock_i \rangle
\end{array}
$$

$$
action\_step \triangleq s1 - s2 \lor upd \lor s2 - s1 \lor disp
$$

temporal

$$
Spec \triangleq Init \land \Box[action\_step \lor clock\_tick]_{\langle periodic\_clock_i, clock_i, w, ks, pc \rangle}
$$

Comment to *clock_tick* : $periodic\_clock_i > period_i$ can never happen when $pc[i] =$ “s0”. $periodic\_clock_i$ only increases in locations “s1” and “s2”. $clock_i$ only increases in location “s2”.

The introduction of a periodic clock in specification *Task4* should preserve the same periodic invocation of the task as in *Task3*. This means in location

$pc[i]$ = "s3", both tasks in specification *Task3* and *Task4* should be delayed for the same time.
Now we proof that this property is preserved. We assume the constant $period_i$ has the same value in specification *Task3* and *Task4*.

module *periodic-monotonic-clock*

Induction start
In the initial state $delay\_time[i] - ks.ltime + period_i$ *is equal to* $period_i - periodic\_clock_i$ *because* $delay\_time[i] = 0$, $periodic\_clock_i = 0$ *and* $ks.ltime = 0$.
When we enter location "s3" *in* Task3 $delay\_time[i]$ *is increased by* $period_i$. *When we enter* "s3" *in* Task4 $periodic\_clock_i$ *stays unchanged. This means in this state* $delay\_time[i] - ks.ltime$ *is equal to* $A.period_i - periodic\_clock_i$.
Proof : *When we enter* $pc[i]$ = "s3" *for the first time* $periodic\_clock_i$ *and* $ks.ltime$ *have never been reset,* $ks.ltime$ *is equal to* $periodic\_clock_i$ *and* $delay\_time[i]$ *has the value* $period_i$. *Therefore* $delay\_time[i] - ks.ltime$ *is equal to* $period_i - periodic\_clock_i$.
Whenever we leave location "s3" *in* Task3 *for the first time and enter location* "s1" *from* "s3", $delay\_time[i]$ *is unchanged and* $ks.ltime$ *has the value of* $delay\_time[i]$. *Whenever we leave location* "s3" *in* Task4 *for the first time and enter location* "s1" *from* "s3", $periodic\_clock_i$ *is reset to zero (that is a decrease by* $period_i$*). Therefore when we enter* "s0" *via* "s3" *for the first time,* $delay\_time[i] - ks.ltime + period_i$ *is equal to* $period_i - periodic\_clock_i$. *This is the property of the initial state again.*
In all other actions $ks.ltime$ *and* $periodic\_clock_i$ *increase with the same speed and* $delay\_time[i]$ *is not changed. Furthermore whenever we enter location* "s3" *in* Task3, $delay\_time[i] - ks.ltime \leq period_i$.

9 UPPAAL Timed Automata

The specification *Task4* can be interpreted as a timed automata. The translation of the specification to the input language of the model checker UPPAAL has to consider the specific features of the tool.
The UPPAAL timed automaton is a finite state automaton equipped with real valued clocks, integer valued data variables and synchronisation actions. Transitions have guards and reset operations on clock and data variables. At any time an automaton can change its location by following a transition, provided the current clock and data variables satisfy the enabling guard. Transitions occur instantaneously and time only passes in locations. The

values of all clocks increase synchronously with time. An invariant is associated with every location which determines valid clock values for the location. Automata synchronize with each other via channels and shared variables. The rules of UPPAAL timed automata which govern change of location have three different components. Any of the three rule components may be empty.

guard the constraints on clock values and the values of global variables under which the transition is taken. For example, $clock_i \triangleq 1, w \leq pri_i * 2 - 1$ states the value of my clock is 1 and that the value of global variable w is at most $pri_i * 2 - 1$.

synchronization causes two different tasks to synchronize over a named channel. For example, synchronization channels are used to ensure that the actions occur simultaneously. *urgent* channels ensure that a synchronized transition is taken as soon as it is enabled.

assignments reset clock variables and update global variables. For example $clock_i := 0, w := w + pri_i$ describes the addition of a task to the set of waiting tasks and sets the task's clock to 0 ready to measure the timing of the next action.

committed locations have to be left immediately

In recent versions of UPPAAL [14], an automata template can be defined using parameters for any of its constants, clocks, data variables and synchronization channels. A network of automata can then be constructed by instantiating parameterized automata with actual parameters. Process parameters can include arrays of data variables.

10 Translating the Final Specification to UPPAAL

The steps to translate specification *Task4* to the input language of UPPAAL are described below.

Translating Integrator Clocks to UPPAAL

The clock $clock_i$ in specification *Task4* only increases in location $pc[i] =$ "s2" and is stopped in all other locations. In UPPAAL we do not have integrator

clocks that can be switched off. We simulate an integrator clock by introducing an integer variable $clock_i$ and an UPPAAL clock gc. In location "s2" gc is used as a trigger, always set to 0 when it has value 1. $clock_i$ is increased by one in "s2" whenever the clock gc is one.

Translating action *clock_tick*

UPPAAL Timed automata are finite state machines with clocks. In every step either a transition of the automata is taken or the the value of the clocks are increased. The increase of the clocks is not specified explicitly but it is implicit in the specification. In our TLA specification we specify the increase of the clock values explictly in action *clock_tick*. We translate this action by declaring a clock variable $periodc\_clock_i$ in UPPAAL, by adding corresponding location invariants to the locations of the UPPAAL automaton ($periodic\_clock_i \leq period_i$), guards on the transitions ($periodic\_clock_i = period_i$) and assignments to variables on the transitions ($periodic\_clock_i := 0$).
In action *clock_tick* we define w in the next state as

$$\begin{aligned} w' = \textstyle\sum_{i=0}^{MAX\_TASK} 2^i . \textbf{if} \; & \vee \; pc'[i] = \text{“s1”} \;\; \textbf{then} \; 1 \\ & \vee \; pc'[i] = \text{“s2”} \\ & \vee \; pc'[i] = \text{“s4”} \\ & \qquad\qquad\qquad\qquad \textbf{else} \;\; 0 \end{aligned}$$

This cannot be implemented in UPPAAL directly. The state change of w is implemented by decreasing and increasing w by 2^i when automaton i enters location "s3" and leaves "s4". We can check that the property

$$\begin{aligned} w = \textstyle\sum_{i=0}^{MAX\_TASK} 2^i . \textbf{if} \; & \vee \; pc[i] = \text{“s1”} \;\; \textbf{then} \; 1 \\ & \vee \; pc[i] = \text{“s2”} \\ & \vee \; pc[i] = \text{“s4”} \\ & \qquad\qquad\qquad\qquad \textbf{else} \;\; 0 \end{aligned}$$

is preserved. The assignment of $clock_i$ in action *clock_tick* is translated as described before in *Translating Integrator Clocks*.

Translation of the predicate *urgent*

The predicate *urgent* defines assumptions on the timed behavior and is used in action *clock_tick* :

$$
\begin{aligned}
urgent \triangleq\ & \vee\ pc[i] = \text{“s1”} \wedge w \leq 2.2^i - 1 \\
& \vee\ pc[i] = \text{“s2”} \wedge clock_i = wcet_i \\
& \vee\ pc[i] = \text{“s3”} \\
& \vee\ pc[i] = \text{“s4”}
\end{aligned}
$$

The predicate has to be implemented as automaton location invariants and guards on transitions. $pc[i] = \text{“s2”} \wedge clock_i = wcet_i$ is translated to a location invariant $clock_i \leq wcet_i$ and the guard $clock_i = wcet_i$ on the transition from “s2” to “s3”. $pc[i] = \text{“s3”}$ is translated to a committed location.
$pc[i] = \text{“s1”} \wedge w \leq 2.2^i - 1$ is translated with an urgent channel *change* on the transition from “s1” to “s2”, and a second automata that always offers a synchronization with a matching channel. This is a way to force the transition to be taken if it is enabled.

Ignoring location *s4*

The transition from “s2” to “s4” is taken when a task did not finish its execution in its period. If we are only interested in checking whether a task always finishes its execution within its period after invocation, the “s4” corresponds to a state where a task did not meet its deadline. If we do not model “s4” in UPPAAL, a deadlock during model checking indicates a task does not finish within its period because there is no transition that can be taken if $periodic\_clock_i \geq period_i \wedge clock_i = wcet_i$. Ignoring “s4” reduces the state space further.

An example of a mine pump implemented in UPPAAL is :

```
clock c1,c2,c3,c4,c5,c6,gc;
int  [0,63] w:=0;
urgent chan change;

Methane_Monitor :=TaskICPP(c1, 200, 32, 58);
Air_Monitor :=TaskICPP(c2, 300, 16, 37);
CO_Monitor :=TaskICPP(c3, 300,  8, 37);
Safety_Checker :=TaskICPP(c4, 350,  4, 39);
Low_Sensor :=TaskICPP(c5, 1000, 2, 33);
High_Sensor :=TaskICPP(c6, 1000, 1, 33);

system  Methane_Monitor, Air_Monitor, CO_Monitor,
        Safety_Checker, High_Sensor, Low_Sensor, Idle, Change-sync;
```

The worst case response time of ICPP can be found by selecting a `R` for every `TaskX` so that:

```
A[] (TaskX.s3) imply clock_i <= R-1     fails
A[] (TaskX.s3) imply clock_i <= R succeeds
```

11 Conclusions and Future Work

This paper presented a new method for verifying a real-time, concurrent Ada95 program and demonstrated its feasibility by verifying a Ravenscar profile scheduler. Our method is also applicable to other real-time programming languages although the SPARK analyzer makes Ada95 particularly convenient. TLA+ proved to be sufficiently general to describe the different timed automata models we needed for verification and also to provide a framework for scalable refinement proofs.
Where possible we used existing verification tools to support our method. The SPARK analyzer was used to verify pre- and postconditions of the procedures of single Ada95 tasks, from which a timed automata process model of that task could be constructed. In this example, we estimated worst case execution times for procedures, but a SPARK extension tool such as [18] could be used to calculate accurate timing values. We used the UPPAAL model-checker to verify requirements of timed automata specifications after abstraction.
The TLA+ refinement steps have been done by hand here, but tool support as in [19] or using a general purpose theorem prover such as Isabelle [17] will be considered in future work.
Our model of Ada95 implementations includes assumptions about the execution environment of a set of tasks, such as context switching, and clock properties. In future work, we plan to further explore our model of such behaviors.

References

[1] Martin Abadi and Leslie Lamport. An Old Fashioned Recipe for Real-Time. In *Proceedings of REX Workshop Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 1–27. Springer Verlag, 1994.

[2] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] Peter Amey and Dave Allen. *INFORMED Design Method for SPARK.* Praxis Critical Systems, 2000.

[4] John Barnes. *High Integrity Ada-The SPARK Approach.* Addison-Wesley, 1997.

[5] John Barnes. *programming in Ada95.* Addison-Wesley, second edition 1998.

[6] Eric Bruneton and Jean-Francois Pradat-Peyre. Automatic Verification of Concurrent Ada Programs. In *Reliable Software Technologies-Ada Europe '99*, volume 1622 of *Lecture Notes in Computer Science*, pages 146–157. Springer Verlag, 1999.

[7] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar Tasking Profile for High Integrity Real-time Programs. In *Reliable Software Technologies-Ada-Europe '98*, volume 1411 of *Lecture Notes in Computer Science*, pages 263–275. Springer Verlag, 1998.

[8] Alan Burns. The Ravenscar Profile. Technical report, University of York, UK, 1999.

[9] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley, 1997.

[10] Simon Fowler and Andy Wellings. Formal Analysis of a Real-Time Kernel Specification. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135, pages 440–458. Springer Verlag, 1996.

[11] F. J. Garcia and J.L. Villaroel. Translating Time Petri Net Structures into Ada95 Statements. In *Reliable Software Technologies - Ada-Europe '99*, volume 1622 of *Lecture Notes in Computer Science*, pages 158–169. Springer Verlag, 1999.

[12] Thorsten Gerdsmeier and Rachel Cardell-Oliver. Analysis of Scheduling Behaviour with Timed Automata. In *7th Computing : The Australasian Theory Symposium (CATS 2001)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2001.

[13] Jonathan Hammond and Peter Amey. *Generation of VCs for SPARK Programs*. Praxis Critical Systems, 2000.

[14] Kim G. Larsen and Paul Pettersson. Uppaal2k. Newsletter 10, BRICS, October 1999.

[15] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[16] K. Lundqvist and L. Asplund. A Formal Model of the Ada Ravenscar Tasking Profile; Delay Until. In *Proc. ACM SIGAda Annual International Conference (SIGAda'99)*, pages 15–21, 1999.

[17] Lawrence C. Paulson. *Isabelle A Generic Theorem Prover*, volume 828. Springer Verlag, 1994.

[18] A. Burns R. Chapman and A. J. Wellings. Combining Static Worst-Case Timing Analysis and Program Proof. *Real-Time Systems*, 1996.

[19] Peter Groenning Urban Engberg and Leslie Lamport. Mechanical Verification of Concurrent Systems with TLA. In *Proceedings of the Fourth International Conference CAV'92*, volume 663, pages 44–55, 1992.

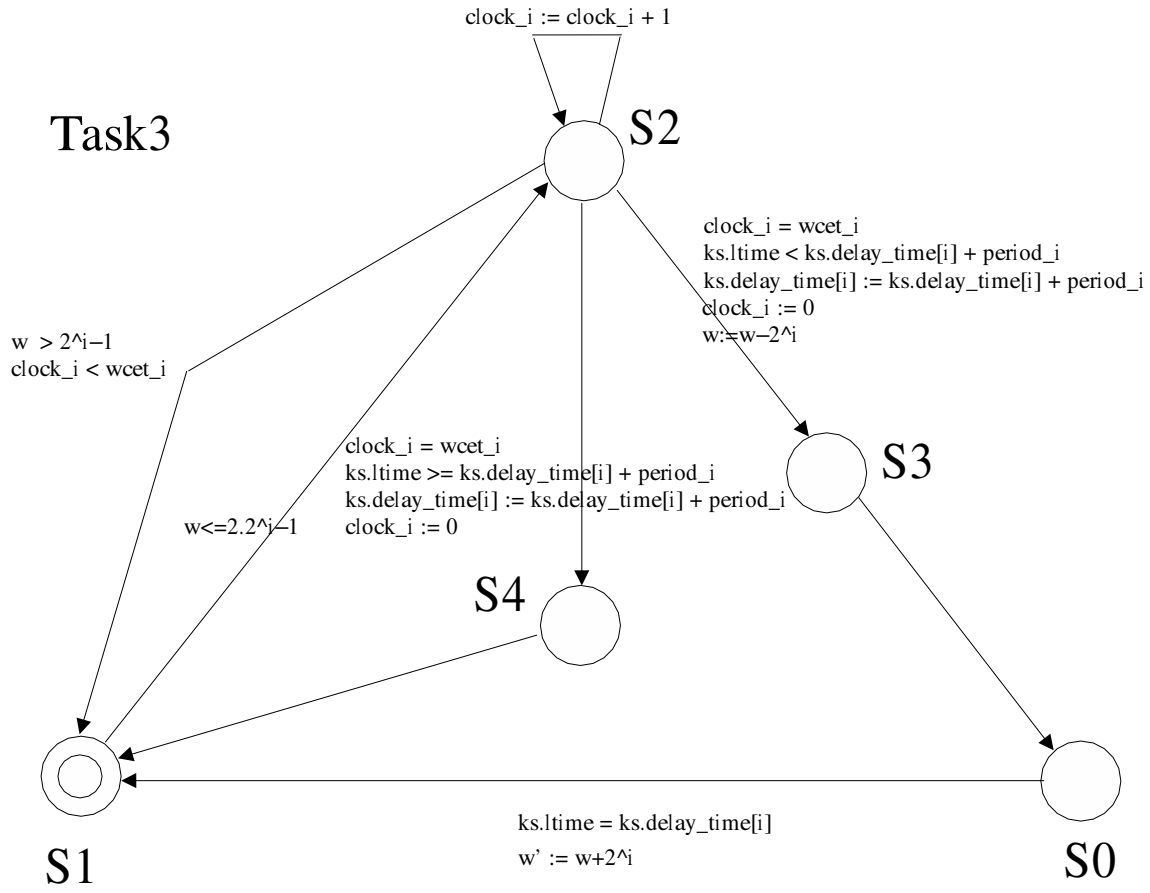


Figure 2: Task3

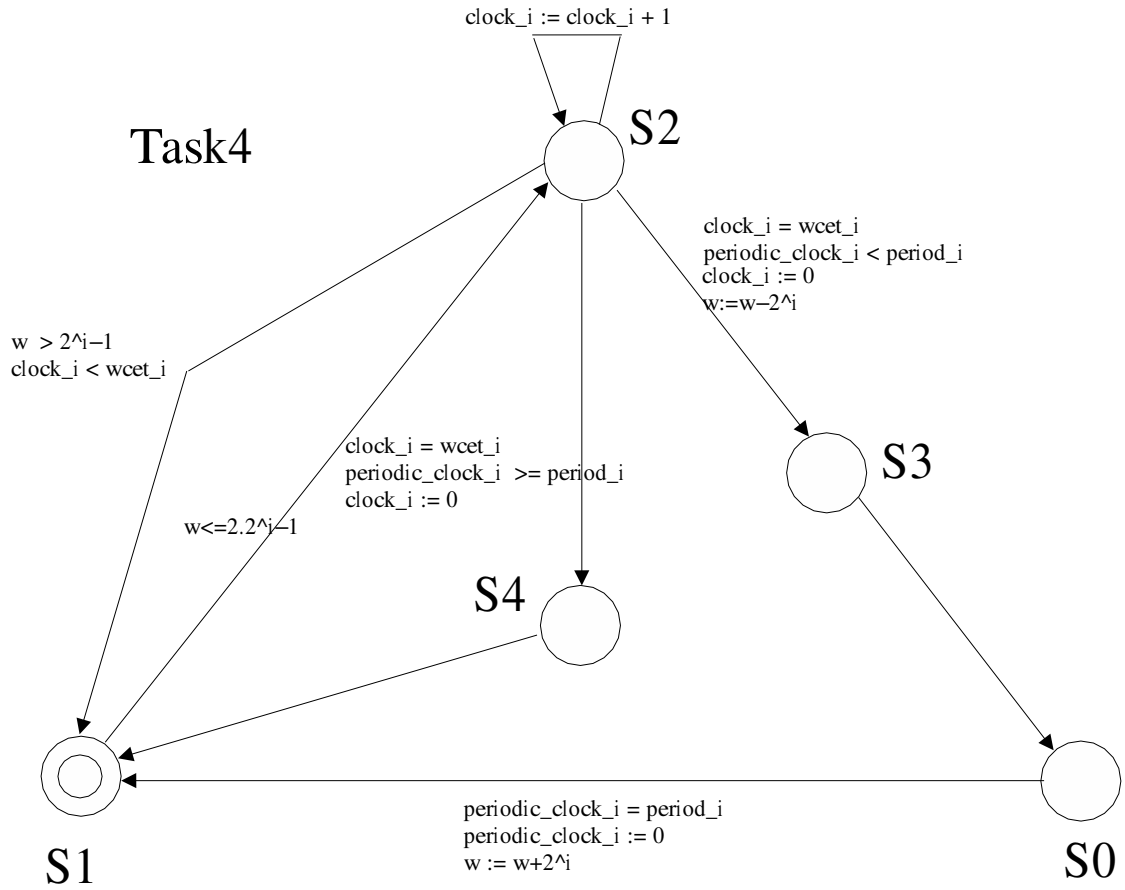


Figure 3: Task4

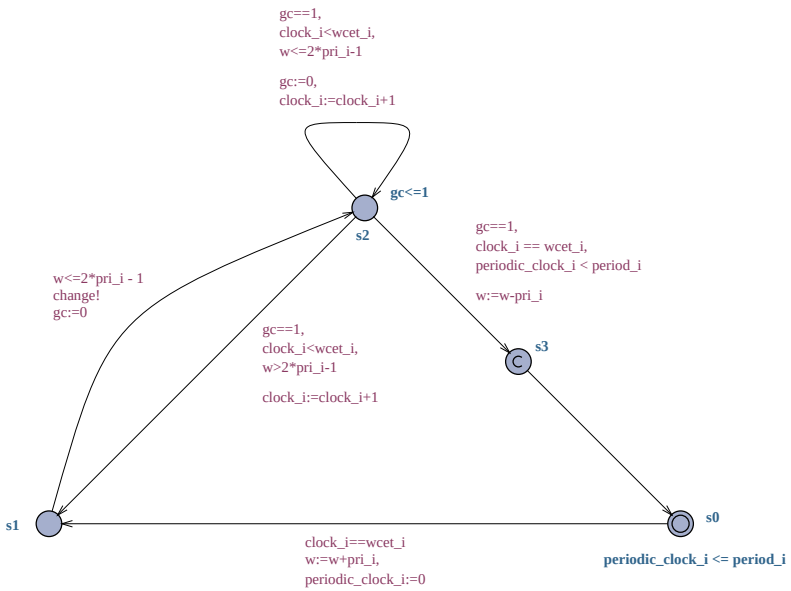


Figure 4: User Task Template

```
process Task4 (clock periodic_clock_i; int clock_i;
               const period_i, pri_i, wcet_i)
```

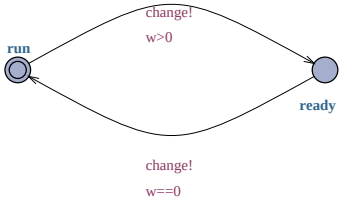


Figure 5: Idle Task Template

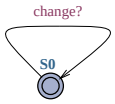


Figure 6: Abstraction method