Model-Checking Temporal Properties of Real-Time HTL Programs

André Carvalho[2], Joel Carvalho[1], Jorge Sousa Pinto[2], and Simão Melo de Sousa[1]

[1] Departamento de Informática, Universidade da Beira Interior, Portugal, and LIACC, Universidade do Porto, Portugal
[2] Departamento de Informática / CCTC Universidade do Minho, Braga, Portugal

Abstract. This paper describes a tool-supported method for the formal verification of timed properties of HTL programs, supported by the automated translation tool HTL2XTA, which extracts from a HTL program (i) an Uppaal model and (ii) a set of properties that state the compliance of the model with certain automatically inferred temporal constraints. These can be manually extended with other temporal properties provided by the user. The paper introduces the details of the proposed mechanisms as well as the results of our experimental validation.

1 Introduction

New requirements arise from the continuous evolution of computer systems. Processing power alone is not sufficient to satisfy all the industrial requirements. For instance in the context of critical systems, the safety and reliability aspects are fundamental [14]: it is not sufficient to merely provide the technical means for a set of tasks to be executed; it is also required that the system (as a whole) can correctly execute all of the tasks in due time. The focus of this paper is precisely on the reliability of safety-critical systems. Such systems are usually real-time systems [11] that add to traditional reliability requirements the intrinsic need to ensure that tasks are executed within a well-established time scope. For such systems, missing these timing requirements corresponds to a system failure.

Our study considers the Hierarchical Timing Language (HTL) [5, 6, 10] as a basis for real-time system development, and addresses the issue of the (automated) formal verification of timing requirements. Since HTL is a coordination language [4] for which schedulability analysis is decidable, our focus here is on the verification of complementary timing properties. The verification framework we propose relies on model checking based on timed automata and timed temporal logic. The contribution of this paper is a detailed description of the methodology and its underlying tool-supported verification mechanism.

Our tool takes as input a HTL program and extracts from it an Uppaal model and a set of proof obligations that correspond to certain expected timed temporal properties. The resulting model can be used to run a timed simulation of the program execution, and the properties can be checked using the proof facilities provided by the Uppaal tool. With the help of these mechanisms, the development team can audit the program against the expected temporal behaviour.

Motivation and Related Work. The HTL language is derived from Giotto [9]. Giotto-based languages share the important feature that they allow one to statically determine the schedulability of programs. Although academic, these languages have a number of interesting properties that cannot be found in languages currently used in industry, including efficient reuse of code; theoretical ease of adaptation of a program to several platforms; hierarquical construction of programs; and the use of functional features of languages without limitations.

HTL introduces several improvements with respect to Giotto, but the HTL platform still lacks verification mechanisms to complement schedulability analysis, in order to allow the language to compete with other tools more widely used in industry. Bearing in mind this aspect, we propose to complement the verification of temporal HTL with model checking [3]. While the static analysis performed by the HTL compiler enforces the schedulability (seen as a safety property) of the set of tasks in a program, a model checker allows the system designer to perform a temporal analysis of the tasks' behaviour from the specified timing requirements – an aspect that is ignored by the HTL tools.

The verification methodology proposed in this paper is inspired by [13], but uses a different abstraction based on the *logical execution time* of each task. Unlike [13], a key point of our tool chain is that the verification is fully automatic. [12] proposes the use of Uppaal with a related goal: the verification of a Ravenscar-compliant scheduler for Ada applications.

HTL. The Hierarchical Timing Language [5–7, 10] is a coordination language [4] for real-time critical systems with periodic tasks, which allows for the static verification of the schedulability of the implemented tasks. The aim of coordination languages is the combination and manipulation of programs written in heterogeneous programming languages. A system may be implemented by providing a set of tasks written in possibly different programming languages, together with a HTL layer, and additionally specifying how the tasks interact. This favours a clear separation, in the system design, between the functional layer and the concurrent and temporal aspects. The HTL toolchain provides code generators that translate the HTL layer into executable code of the target execution platform.

A fundamental aspect of HTL is the *Logical Execution Time* (LET), that provides an abstraction for the physical execution of tasks. The LET of a task considers a time scope in which the task can be executed regardless of how the operating system assigns resources to this task. The LET of a periodic task implementing a *read data; process; write data* cycle begins in the instant when the last variable is read and ends when the first variable is written.

For illustration purposes, we give in Listing 1.1 an excerpt of a HTL program (based on the 3TS_Simulink case study, see Section 5). A HTL program is composed by a number of main commands which allow programmers to describe the desired behaviour of almost any program. These commands are *communicator*, *module*, *task*, *port*, *mode*, *invoke* and *switch*. Briefly, a communicator is a typed variable which can be accessed any time during the execution; modules have to be declared after communicators and their bodies are composed by ports, tasks and modes. At least one (initial) mode must be declared. The task command,

```
module IO start readWrite{
  task t_read
      input() state()
      output(c_double p_h1, c_double p_h2,c_bool p_V1, c_bool p_V2)
      function f_read;
  (...)

  mode readWrite period 500{
      invoke t_read
            input()
            output((h1,3), (h2,3), (v1,1), (v2,1));
      (...)
  }
}
```

Listing 1.1. 3TS Simulink code snippet

as the name indicates, is used to declare tasks, taking as arguments possible input/output ports and a Worst Case Execution Time (WCET) estimation. Similarly to a communicator, a port is a typed variable accessed during program execution, but in this case declared inside a module. The set of modes declared inside a module defines the module's behaviour. Through the modes declaration it is possible to know which tasks will be executed, and at which moment. The invocations are responsible for dictating when the tasks should be executed, and define the LET of each task. Finally, the switch command, which takes as input a condition and a mode identifier, is used to change the current execution mode.

HTL favours a layered approach to the development of programs. Tasks can be organized in refinements that allow programmers to provide details gradually, and also allow for a more finely grained task structure. A concrete task refines an abstract task if it has the same frequency as the abstract task and it is able to provide a time behaviour that is at least as good as the behaviour of the abstract task. The notion of refinement correctness is then expressed in terms of *time safety*. The refined task must be time-indistinguishable from the abstract task; a concrete HTL program is schedulable if it contains only time-safe refinements of the tasks of a schedulable abstract HTL program.

Uppaal. The Uppaal tool is a modelling application developed at the universities of **Upp**sala and **Aal**borg, based on networks of timed automata [2]. The tool offers simulation and verification functionality based on model checking of formulas of a subset of the TCTL logic [1]. Uppaal is particularly suitable for modeling and analysing the timed behaviour of a set of tasks; properties like *two given tasks t_1, t_2 do not reach the states A and B simultaneously* are typical of the kind of analyses that can be performed with Uppaal.

Since the model checking engine is independent from the GUI, both visual and textual representations of timed automata can be used for the verification tasks. This is particularly interesting when Uppaal is used in cooperation with other tools. Timing requirements (target properties to be checked) can be specified using the editing facilities of the GUI, or separately in a file. This last approach is used by the toolchain introduced in this paper.

2 The HTL2XTA Toolchain

The purpose of the verification methodology proposed in this paper is to extend the verification capabilities provided by the HTL platform. Given a HTL program and the schedulability analysis provided by the regular HTL toolchain [7], the methodology consists in the following two steps:

1. From a HTL program, the HTL2XTA translator produces two files: one (.xta) contains a model of the program (timed automata); the other (.q) a set of automatically inferred properties (timed temporal logic formulas). The translation algorithm has a recursive structure and requires only two depth-first traversals of the AST: the first one produces the model and the second one infers the properties.
2. Both these files are fed to the Uppaal model checker; the GUI or the model checker engine (*verifyta*) can be used to check if the properties are satisfied.

We remark that the automatically generated properties correspond to relatively simple timing requirements; formulas for more complex requirements, such as "task X must not execute at the same time as task Y", or "if task X executes, then after T time units task Y must also execute" are not automatically generated, but can of course be manually incorporated in the .q file after the first step above. Writing the appropriate TCTL formulas must of course take into consideration the requirements and the generated model. We now turn to an exploration of the involved translation mechanisms, which will be detailed in the next two sections.

Model Translation. With the classic state space explosion limitation of model checking [3] in mind, and given the central role of the models in the verification process, it was decided to avoid translation schemes that would result in the construction of very complex models. Therefore, and given that the HTL platform already performs a scheduling analysis, the translation abstracts away from the physical execution of tasks, unlike, say, the approach described in [13]. As such, we consider that the notion of LET is sufficient to allow the remaining interesting timing properties to be checked. A network of timed automata is then obtained from a HTL program as follows:

– Each task is modeled as a single automaton with its own LET, calculated from the concrete ports and the communicators given in the task's declaration. The lower bound of the LET corresponds to the instant in which the last variable reading is performed, and the upper bound to the instant in which the first variable writing is performed.
– For each module in the HTL program a timed automaton is created. Note that each mode in a module represents the execution of a set of tasks, and that, at any moment, each module can only be in one operation mode, thus there is no need to have more than one automaton for each module. Whenever the (module) automaton performs an execution cycle, it will synchronize

```
/* Deadlock Free -> true */
A[] not deadlock

/* P1 mode readWrite period 500 @ Line 19 -> true */
A[] sP_3TS_IO.readWrite imply ((not sP_3TS_IO.t>500) && (not sP_3TS_IO.t
     <0))

/* P2 mode readWrite period 500 @ Line 19 -> true */
sP_3TS_IO.readWrite --> (sP_3TS_IO.Ready && (sP_3TS_IO.t==0  ||
     sP_3TS_IO.t==500))

/* P1 Let of t_write = [400;500] @ Line 21 -> true */
A[] (IO_readWrite_t_write.Let imply (not IO_readWrite_t_write.tt<400 &&
     not IO_readWrite_t_write.tt>500))
```

Listing 1.2. Example of annotated properties

with the automata representing the tasks invoked in the specified mode. The level of abstraction adopted completely ignores the type of communicator as well as the initialization driver.

Since HTL is a hierarchical coordination language, a very relevant aspect is the number of refinements in the program (directly related to program hierarchies), which can naturally increase the complexity of the model substantially. By default, the translation process reflects faithfully the refinement present in the HTL programs. However in some cases this could make the model exploration impracticable, and for this reason the translator allows for the construction of models to take the desired level of refinement as input.

Inference of Properties. Listing 1.2 shows examples of automatically produced timing properties. The automatically inferred properties are all related with some HTL feature, like the modes' periods, the LET of each task, the tasks invoked in each mode, and the program refinement. To allow for traceability, each property is annotated with a textual description of the feature to check, a reference to the position of the respective feature in the HTL file, and the expected verification result. The inferred properties can and should be manually complemented with information extracted from the established temporal requirements. The automata corresponding to a given module and tasks, as well as the states corresponding to task invocations and LETs, are identified by clearly defined labels, which facilitates writing properties manually.

3 Model Translation

Some aspects of HTL are purely ignored by the translation process, either because they do not bring any relevant information, or because the abstraction level of the model is not sufficient to cope with it. The translation process is syntax-oriented and based on the abstract syntax tree (AST) of the HTL language, which was built using a HTL grammar. It supports all of the HTL language, however there is information that is not analysed or translated by the tool.

Let us consider the definition of the function T that takes as input a HTL program and returns a network of timed automata (NTA). Naturally, this function is defined recursively over the structure of the AST. An auxiliary function A is used for task invocation analysis, that takes as argument a HTL program and returns relevant information to build the NTA.

Translation of Mode_Switch. Consider the abstract representation of a switch instruction as the tuple (n, s, p), where n is the name of the mode for which the change of execution is pretended, s the name of the function (in the functional code) that evaluates whether the change should take place, and p the declaration position in the HTL file. Let $Prog$ denote the set of all programs, then we have $\forall switch \in Prog, T_{switch}(n, s, p) = \emptyset$. Note that the non-determinism of Uppaal will be important to guarantee that the modes are alternated during the execution. The translation itself is not affected by any mode switches.

Translation of Types and Initialization Drivers. Let ct be a type and ci the declaration of the initialization driver. We have $\forall dt \in Prog, T_{dt}(ct, ci) = \emptyset$. Neither the type nor the initial value (initialization driver) of a declaration have any impact on the application of the translation process. This information does not contribute to the temporal analysis.

Translation of Task Declarations. Consider the abstract representation of a task as the tuple (n, ip, s, op, f, w, p), where n is the name of the task, ip the list of *input ports*, s the list of internal states, op the list of *output ports*, f the name of the function which implements the task, w the task's WCET, and finally p the task declaration position in the HTL file. We have $\forall task \in Prog, T_{task}(n, ip, s, op, f, w, p) = \emptyset$. Analogously to the previous situations, task declarations do not have any impact on the translation.

Translation of Communicator Declarations. Consider the abstract representation of a communicator as the tuple (n, dt, pd, p), where n is the communicator's name, dt the communicator's type with ct, ci as initialization driver, pd the communicator's period and p the communicator's declaration position in the HTL file, then $\forall communicator \in Prog, T_{communicator}(n, dt, pd, p) = \emptyset$.

Once more the translator ignores the declaration. In order to evaluate the LET (see below) the following clause is defined for the auxiliary function A: $\forall communicator \in Prog,\ A_{communicator}(com, dt, pd, p) = pd$, even if the communicator com does not have a direct representation in the model given the abstraction level adopted.

Translation of Ports Declaration. Let the abstract representation of a port be the tuple (n, dt, p), where n is the ports's name, dt the port's type with ct, ci the initialization driver and p the port's declaration position in the HTL file, then $\forall port \in Prog, T_{port}(n, dt, p) = \emptyset$. The port's declaration is ignored in the translation, and moreover no task invocation analysis is performed. In task invocations ports are just names.

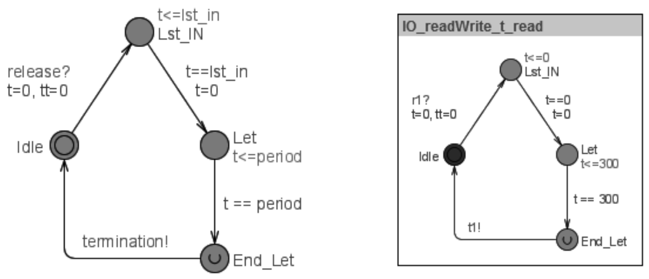


Fig. 1. *taskTA* automata on the left and instantiation on the right

LET Transposition

This translation is based on an implementation of the concept of LET, based on the timed automata *taskTA*, *taskTA_S*, *taskTA_R* and *taskTA_SR*. These four automata result from the use of concrete ports in the task invocations: *taskTA* represents the task invocations where only communicators are used, *taskTA_S* (S for *send*) those where a single concrete port is used as output, *taskTA_R* (R for *receive*) those where a single concrete port is used as input, and finally *taskTA_SR* where two concrete ports are used, as input and as output.

In the following, a task invocation will be seen in abstract terms as the tuple (n, ip, op, s, pos) where n is the invoked task's name, ip is the input port's (variables) mapping, op the output port's (variable) mapping, s the name of the task's parent, and finally pos is the task's declaration position in the HTL file.

TaskTA. Let *Port* be the set of all concrete ports, *cp* be one concrete port, and $taskTA(r, t, p, li)$ be a timed automaton where r is a *release* urgent synchronization, t is a *termination* urgent synchronization, p the task's LET period and li the exact moment where the last variable is read. Then we have

$$\begin{aligned} &\forall cp \in Port, \forall invoke \in Prog, cp \notin ip, cp \notin op, \Rightarrow \\ &\qquad\qquad T_{invoke}(n, ip, op, s, pos) = taskTA(r, t, p, li) \end{aligned}$$

Each task invocation in which no concrete ports are used either in the input or in the output variables, gives rise to an automaton *taskTA* (see Figure 1). The urgent synchronization channels r and t are calculated in the system declaration. For each task instantiation the channel r has an unique name, produced by an enumeration $r_1, r_2, r_3, \ldots$ Similarly, the channel t has an unique name for each set of mode automata, produced by an enumeration $t_1, t_2, t_3, \ldots$

The instant at which the last input variable li is read is calculated as a product of the maximum value of each instance of input communicator and the period (in the case of non-existence of input variable, this instant is considered to be zero). The LET's period p is the subtraction between the instant where the

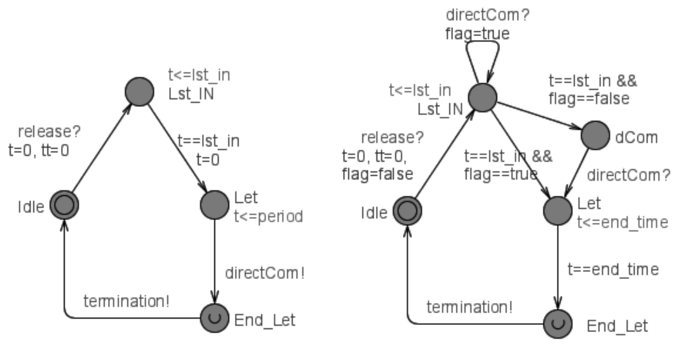


Fig. 2. *taskTA_S* automaton on the left and *taskTA_R* on the right

first output port is written (in case of non-existence, the value is the respective mode's period) and li.

TaskTA_S. Let $taskTA\_S(r, t, dc, p, li)$ be a timed automaton where r is the *release* urgent synchronization, t a *termination* urgent synchronization, dc the urgent synchronization of a direct communication (*directCom*), p the task's LET period and li the instant where the last input variable is read, then

$$\forall cp \in Port, \forall invoke \in Prog, cp \notin ip, cp \in op, \Rightarrow \\ T_{invoke}(n, ip, op, s, pos) = taskTA\_S(r, t, dc, p, li)$$

For each task invocation, the existence of a concrete port in the set of output variables and non-existence in the set of input variables originates the instantiation of a *taskTA_S* automaton (Figure 2, left). This automaton is very similar to *taskTA* – the difference is just the inclusion of direct communication.

TaskTA_R. Let $taskTA\_R(r, t, dc, p, li)$ be a timed automaton with r, t, dc, p, and li the same as in the previous case, then

$$\forall cp \in Port, \forall invoke \in Prog, cp \in ip, cp \notin op, \Rightarrow \\ T_{invoke}(n, ip, op, s, pos) = taskTA\_R(r, t, dc, p, li)$$

For each task invocation, the existence of a concrete port in the set of input variables and non-existence in the set of output variables originates the instantiation of *taskTA_R* automaton (Figure 2, right). This automaton is slightly more complex than *taskTA_S* since it considers two alternative paths for the initialization of the task's LET. The first one encodes the direct communication done before the reading of the last communicator (with no impact on the LET's start) and the later encodes awaiting of the port reading after all communicators have been read (the LET's start becomes dynamic). In this last case, the start of the LET depends on a direct communication with another task in the same mode.

Modules and Modes. Consider a module abstracted as a tuple (n, h, mi, bm, pos), where n is the module's name, h a list of hosts, mi the initial mode, bmu the module's body and pos the module's declaration position in the HTL file.

Let $moduleTA(ref, rl, tl)$ be a timed automaton with ref the refinement's urgent synchronization channel (if it exists), rl the set of all release urgent synchronization channels coming from the invocations of module tasks, and finally tl the set of all termination urgent synchronization channels coming from the invocations of module tasks, then

$$\forall module \in Prog, T_{module}(n, h, mi, bm, pos) = moduleTA(ref, rl, tl)$$

For each module a timed automaton is dynamically created. Unlike tasks automata, where the instantiation of the different default automata is done by just matching the input parameters, here a single automaton is attributed to each module, instantiated by passing as parameters the synchronization channels used by the module's task invocations.

Consider now a mode abstracted as the tuple $(n, p, refP, bmo, pos)$, where n is the mode's name, p is the period, $refP$ is the refinement program for that mode (if it exists), bmo the mode's body, and pos the mode's declaration position in the HTL file. Let $subModule(e, t)$ be a subset of the timed automaton's $moduleTA$ declaration where e is the set of states (with invariants) and t the set of transitions (with guards, updates and synchronizations), then we have

$$\begin{aligned}\forall mode \in module, \exists subModule(e, t) \in moduleTA,\\ T_{mode}(n, p, refP, bmo, pos) = subModule(e, t)\end{aligned}$$

4 Inference of Properties

This section presents the definition of a function P which accepts a HTL program and returns the specification of properties to verify. Naturally, this function is again defined recursively over the AST structure of the HTL language.

Absence of Block. Let $Prog$ be the set of all programs and df be the absence of blocking property description, then we have $P_{Prog} = df$. The application of this method to any program always produces the same absence of blocking property ($A \Box$ *not deadlock*).

Modes Period. Let the tuple $(n, p, refP, bmo, pos)$ be the abstraction of a mode, where n is the mode's name, p the period, $refP$ the refinement program for that mode (if it exists), bmo the mode's body and pos the mode's declaration position in the HTL file. In the following vm denotes the property specifications of a mode's period. We have

$$\forall mode \in Prog, P_{mode}(n, p, refP, bmo, pos) = vm(p1, p2)$$

We also have, with *moduleTA* a module automaton and *NTA* a set of timed automata,

$$\begin{aligned}
&\forall mode \in Prog, \exists moduleTA \in NTA,\\
&\quad p1 = A\,\Box\, moduleTA.n \Rightarrow ((\neg\ moduleTA.t > p) \wedge\ (\neg\ moduleTA.t < 0)),\\
&\quad p2 = moduleTA.n \Rightarrow (moduleTA.Ready\\
&\qquad\quad \wedge\ (moduleTA.t == 0\ \vee\ moduleTA.t == p)
\end{aligned}$$

The first property $p1$ states that whenever the control state is the mode state, the module's (automaton) local clock is lower than the mode's period, and not negative. The second property $p2$ on the other hand states that whenever the mode's state is reached, the state *Ready* is also reached, which implies that the local clock is either zero or exactly equal to the period's value. The combination of both properties allows the restriction of a mode's period to the interval $[0, p]$, and guarantees that the period's maximum value is reached.

Task Invocations. Let (n, ip, op, s, pos) be a task invocation, where n is the task's name, *ip* the input port's (variables) mapping, *op* the output port's (variable) mapping, s the name of the parent task and finally *pos* the task's declaration position in the HTL file. In the following *vi* denotes the specification of properties in a mode's task invocation. We have

$$\forall invoke \in Prog, P_{invoke}(n, ip, op, s, pos) = vi(p1, p2)$$

Let $taskTA_i$ be the automaton of task i, *taskTA* the set of task automata, $taskState_i$ the task i invocation's state, *modeState* the mode's state where the invocation is done, *moduleTA* a module automaton and *NTA* a set of timed automata, then

$$\begin{aligned}
&\forall i, \exists moduleTA \in NTA, \exists taskTA_i \in TaskTA,\\
&\quad p1 = A\,\Box\,(moduleTA.taskStaste_i \Rightarrow (\neg\ taskTA_i.Idle))\\
&\qquad\quad \wedge\ (moduleTA.Ready \Rightarrow taskTA_i.Idle),\\
&\quad p2 = A\,\Box\,(taskTA_i.Let\ \wedge\ taskTA.tt! = 0) \Rightarrow moduleTA.modeState
\end{aligned}$$

The property $p1$ states that for all executions, every time an invocation's state is equal to a control state, that task's automaton cannot be in the *Idle* state. Moreover, when the respective *moduleTA*'s control state is equal to *Ready*, the task's automaton must be in the *Idle* state. The second property specifies that whenever a task's automaton *Let* state is the control state and the local clock *tt* is different from zero, the execution of the module's automaton must be in the state representing the mode in which the tasks are invoked.

Tasks LET. Considering a task invocation *vlet* in a correct mode, its properties are specified as

$$\begin{aligned}
&\forall invoke \in Prog, P_{invoke}(n, ip, op, s, pos) = vlet(p1, p2, p3), \forall i, \exists moduleTA \in NTA,\\
&\quad p1 = A\,\Box\,(taskTA_i.Let \Rightarrow (\neg\ taskTA_i.tt < 0\ \wedge\ \neg\ taskTA_i.tt > p)),\\
&\quad p2 = A\,\Diamond\, moduleTA.modeState \Rightarrow (taskTA_i.Lst\_IN\ \wedge\ taskTA_i.tt == 0),\\
&\quad p3 = A\,\Diamond\, moduleTA.modeState \Rightarrow (taskTA_i.Let\ \wedge\ taskTA_i.tt == p)
\end{aligned}$$

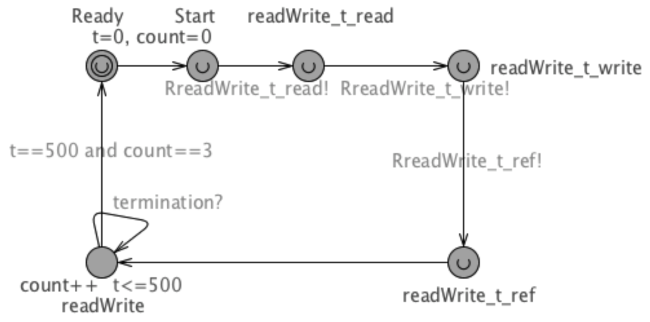


Fig. 3. `P_3TS_IO` automaton (automatic instantiation of `taskTA`)

The LET's validation is done via three distinct properties. Property $p1$ specifies that whenever a task's *Let* is reached, the automaton's local clock tt must lie between 0 and the period. Property $p2$ specifies that every time the mode's state is reached, the Lst_IN state is also reached, necessarily with the local clock tt set to zero. Finally, property $p3$ specifies that every time the mode's state is reached, the *Let* state is inevitably reached, with the clock tt set to the maximum value of the task's period.

5 Case Studies

We consider here the main case study used for illustration purposes by the HTL team: the *three-tank system.* A HTL program implements the controller of a physical system that includes three interconnected tanks with two pumps (for tanks 1 and 3), three taps (one for each tank) and two interconnection taps. The controller supervises the taps in order to maintain the liquid at a specific level.

Description of the Problem. The controller is implemented as a program that contains three modules; two of them (T1 and T2) specify the timing for the controllers of tanks T1 and T2, and the third module specifies the timing for the communications (IO) controller. Each controller module contains one mode which invokes one task and which is refined by a program into a P or PI controller. We assume that in addition to height measuring sensors there exist also sensors that detect perturbation in a tank (this determines the switch between P and PI). The IO module contains one mode named `readWrite` and invokes three tasks: `t_read` reads sensor values and updates communicators `h1`, `h2`, `v1` and `v2`; `t_write` reads communicators `u1` and `u2` and sends commands to the pumps; `t_ref` reads target values and updates communicators `h1_ref` and `h2_ref`.

Generated Model (excerpt). The HTL program is translated into a network consisting of nine timed automata, of which four are default automata that are instantiated by each task invocation depending on the modules and ports declared;

File	Levels	HTL	Model	Verifications	States
3TS-simulink.htl	0	75	263	62/62	7'566
	1	75	199	30/30	666
3TS.htl	0	90	271	72/72	18'731
	1	90	207	40/40	1'123
3TS-FE2.htl	0	134	336	106/106	280'997
	1	134	208	42/42	1'580
3TS-PhD.htl	0	111	329	98/98	172'531
	1	111	201	34/34	1'096
steer-by-wire.htl	0	873	1043	617/0	N/A
	1	873	690	394/0	N/A
flatten_3TS.htl	0	60	203	31/31	411

Table 1. Results

the remaining five represent the modules and respective execution modes. Taking as example the IO module, in which three tasks (`t_read`, `t_write` and `t_ref`) and no ports are used, it is translated as three timed automata, with the default `taskTA` automaton instantiated for each task. An example is shown below, extracted from the (.xta) file produced by the tool. Task invocations are represented by signals RreadWrite_t_read, RreadWrite_t_write and RreadWrite_t_ref.

Properties. In abstract terms the automatically inferred properties can be seen as divided in four classes: *Absence of Block*, *Modes Period*, *Task Invocations* and *Tasks' LET*. We give below an excerpt from the (.q) file generated for the 3TS_Simulink program, that shows two classes of properties : *Absence of Block* for the first property shown, and *Modes Period* for the second. The properties are annotated with a descriptive string and the expected verification result.

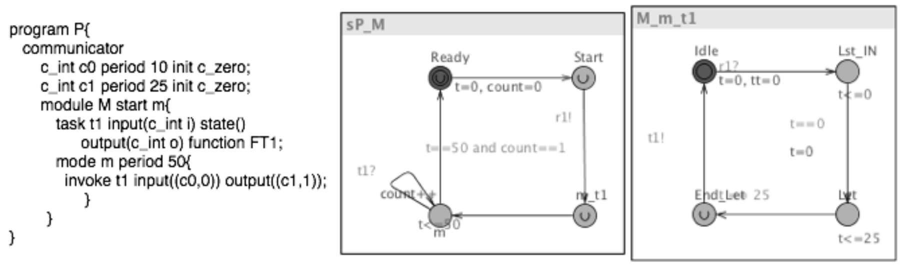
```
//Deadlock Free -> true
A[] not deadlock

//P1 mode readWrite period 500 @ Line 19 -> true
A[] sP_3TS_IO.readWrite imply ((not sP_3TS_IO.t>500) && (not sP_3TS_IO.t
    <0))
```

In some situations, small modifications in the code can have serious effects in the program and affect the verification of properties. In these cases the solution is to analyse and manually specify properties appropriate to each scenario.

Verification. In this case study the HTL2XTA translator for all levels of refinement (switch -L 0) has generated 62 properties automatically, which were all successfully checked (using *verifyta* version 4.0.10). 291794 states were explored and the maximum number of states consumed by a single property was 7566. Some properties were trivially verified. These numbers contribute to an increased confidence degree on the 3TS_Simulink's HTL specification. In spite of the large number of properties and states, this goal is achieved in reasonable time.

Other Case Studies. Using the current version of the translator it was possible to successfully generate models and properties for several HTL programs from [6, 10], and the HTL website. Table 1 summarizes relevant information



```
program P{
  communicator
     c_int c0 period 10 init c_zero;
     c_int c1 period 25 init c_zero;
     module M start m{
        task t1 input(c_int i) state()
             output(c_int o) function FT1;
        mode m period 50{
          invoke t1 input((c0,0)) output((c1,1));
             }
        }
}
```

Fig. 4. A misbehaved HTL program and the corresponding Uppaal automata

about the results, specifically the number of applied levels (0=all, 1=main program), the number of lines in the HTL file, the number of lines in the model's specification file, the number of specified properties versus the number of properties successfully verified, and the number of states explored. The values in the table concern the Uppaal verification, given several different models translated from HTL to XTA.

The proposed toolchain was able to cope with all except one program, the more complex *steer-by-wire* example, for which the verification process does not terminate in reasonable time. Clearly, this is due to the use of all the advanced features of HTL (including a large and complex coordination layer).

6 Towards Correctness

The correctness of the proposed approach has not yet been established; we give here some preliminary remarks. The desired correctness property can be formulated as follows, where p is a HTL program, and MC corresponds to execution of the Model Checker.

If $MC(T(p)) = Error$ then there exists an execution that derives to a timed error execution, following the operational semantics of HTL [5].

Although we have not proven such a correctness result, we give here an example to give the reader an intuition of why the approach should in principle be correct.

The example is shown in Figure 4. The period of the last task's (t_1) output is 25 and the first input is 0; the mode's period is 50, so it is trivial to conclude that this system is schedulable. As such, this program is validated by the HTL toolchain. However, this is not satisfactory, since with these values the LET of this task is specified as [0;25]. Due to the period of the communicator c_0 this task must not execute between instants 0 and 9, and the standard HTL toolchain contains no mechanism to specify or prove situations like this.

It is obvious that in such a small example this problem could be easily detected and corrected by simply changing the instant when the c_0 communicator is used from 0 to 1. But in more complex systems it is hard to obtain any insight about this kind of temporal behaviours.

Considering again the above example, it is straightforward to see that the HTL2XTA translator preserves the bad temporal requirement in the timed automata model. Checking the property $A[]M\_m\_t_1.Let\ imply\ (not(M\_m\_t_1.tt < 10))$, which can be manually inserted in Uppaal and specifies that task t_1 must never occur in an instant inferior to 10, will produce a counter-example.

7 Conclusion and Future Work

The HTL language was created in an academic context, and its transfer to the industrial context remains a challenge. This work is a contribution towards that goal. The tool is available online[3] and runs only on the Linux platform. The HTL2XTA translator was developed in Ocaml, following the traditional compiler design process (but we rely on the HTL compiler for type checking).

We envision two natural improvements of our current methodology. First, the translation methodology has not yet been formally verified (i.e. it has not been proved that the translation preserves the timed semantics of HTL programs). The proof of the theorem sketched in Section 6 is a heavyweight task that must be carefully carried out.

Secondly, the current version of the translator is unable to deal with large-scale HTL programs, and moreover there are still some features of HTL syntax that are not covered by the current version. The translation of the currently covered HTL features can be improved in order to lower the size of the resulting NTA. As future work we plan to analyse these possibilities, and also to extend HTL with *annotations* to introduce supplementary behaviour rules. For instance this may provide insight about the behaviour of programs in the presence of *switch cases*. The impact of such annotations in the model and their influence on the design of the translator will of course have to be carefully considered.

Moreover, in the short term, the toolchain could be improved with a script that provides an automatic analysis of the logfile generated by Uppaal. Such a script could establish conveniently which timing requirements have been checked and which have not, and create a final report based on this information.

Finally, we are interested in transferring our work to the context of the SPARK/Ada language, widely used in the development of safety-critical systems. The *Giotto in Ada* [8] initiative should make this process quite straightforward. We also believe that our translation mechanisms may in principle be applied to other (more exploratory) concurrency models, but this remains an open issue.

Acknowledgment. This work was partially supported by the projects Rescue (PTDC/EIA/65862/2006) and FAVAS (PTDC/EIA-CCO/105034/2008), and by LIACC-UP through the Programa de Financiamento Plurianual, all funded by Fundação para a Ciência e Tecnologia (FCT).

[3] http://sourceforge.net/projects/htl2xta/

References

1. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.
2. Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools, 2004.
3. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
4. David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.
5. Arkadeb Ghosal. *A Hierarchical Coordination Language for Reliable Real-Time Tasks*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2008.
6. Arkadeb Ghosal, Thomas A. Henzinger, Daniel Iercan, Christoph Kirsch, and Alberto L. Sangiovanni-Vincentelli. Hierarchical timing language. Technical Report UCB/EECS-2006-79, EECS Department, University of California, Berkeley, May 2006.
7. Arkadeb Ghosal, Alberto Sangiovanni-Vincentelli, Christoph M. Kirsch, Thomas A. Henzinger, and Daniel Iercan. A hierarchical coordination language for interacting real-time tasks. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 132–141, New York, NY, USA, 2006. ACM.
8. Helge Hagenauer, Norbert Martinek, and Werner Pohlmann. Ada meets giotto. In Albert Llamosí and Alfred Strohmeier, editors, *Ada-Europe*, volume 3063 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 2004.
9. Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 166–184, London, UK, 2001. Springer-Verlag.
10. Daniel Iercan. *Contribuitions to the Development of Real-Time Programming Techniques and Technologies*. PhD thesis, EECS Department, University of California, Berkeley, Set 2008.
11. Shem-Tov Levi and Ashok K. Agrawala. *Real-time system design*. McGraw-Hill, Inc., New York, NY, USA, 1990.
12. Kristina Lundqvist and Lars Asplund. A ravenscar-compliant run-time kernel for safety-critical systems*. *Real-Time Syst.*, 24(1):29–54, 2003.
13. Rajiv Kumar Poddar and Purandar Bhaduri. Verification of giotto based embedded control systems. *Nordic J. of Computing*, 13(4):266–293, 2006.
14. John Rushby. Formal methods and their role in the certification of critical systems. Technical report, Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop, 1995.