Nordic Journal of Computing

VERIFICATION OF GIOTTO BASED EMBEDDED CONTROL SYSTEMS

RAJIV KUMAR PODDAR
PURANDAR BHADURI
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati, Guwahati - 781039, India
{rajiv|pbhaduri}@iitg.ernet.in

Abstract. An implementation of a control system design may not preserve the functional and timing requirements of the application. Our goal is to verify that an implementation meets the high-level timing and functional specifications of a control application. We take Giotto as the implementation model, and verify Giotto models using **UPPAAL**, a tool box for modelling, simulation and verification of timed automata. We present a translation scheme for building timed automata in **UPPAAL** for real-time systems written in Giotto. When translating Giotto to timed automata, we consider timing constraints imposed by the control application, as well as the characteristics of the implementation platform. These timing constraints take into account execution times of atomic tasks, worst case execution times, worst case communication times and jitters. The timed models obtained in this manner are analysed and the corresponding system's functional and timing properties are verified using **UPPAAL** .

We develop the translation scheme in two phases. The first is applicable to basic Giotto models; the latter considers Giotto models with annotations providing information on scheduling and resource allocation. We demonstrate both phases of the scheme by applying it to two Giotto models – an elevator control and a hovercraft control system. The two systems vary in their complexity, their functional and non-functional requirements. We report on the results of our verification of the Giotto models.

ACM CCS Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification – *model checking*; I.6.4 [Simulation and Modeling]: Model Validation and Analysis; I.6.5 [Simulation and Modeling]: Model Development – *modeling methodologies*; C.3 Special-Purpose and Application Based Systems – *real-time and embedded systems*

Key words: embedded control systems, real-time verification, guided model translation

1. Introduction

A *high-level* model of a control system is based on abstractions about the behaviour and the timing requirements of its computer based implementation. Such a model involves a description of the plant behaviour and the control inputs, using differential or difference equations. The control design abstractions include issues such as concurrent execution, instantaneous computation, zero delay, and perfect communication between components or between components and the external en-

Received July 7, 2006.

vironment, among others. These abstractions enhance the platform independence, software portability and reuse and reduce the system development complexity.

When a high-level model for the control design is completed, it is implemented on a particular platform (the word "platform" stands for a hardware configuration together with a real-time operating system). A number of platform dependent issues, not considered at design level, are resolved during implementation; these issues include resource allocation (e.g., the distribution of tasks to different processors and the scheduling policy), task communication and synchronisation (e.g., using shared memory, semaphores or message queues) and achieving the desired degree of fault tolerance through replication and error correction. These issues vary from one platform to another.

Since an implementation involves making design decisions that often break the abstractions, it is possible that the low-level model of computation may not preserve the functional and timing requirements of its high-level counterpart. The correctness of the implemented system depends upon the degree to which the implementation platform meets these assumptions of the high level model. In the design of a real-time control system, the implementation may inadvertently introduce new undesirable properties into the system or destroy desirable ones and therefore, the implementation needs to be verified against the high-level behavioural requirements of the system. We call this *functional verification*. This includes both the functionality and timing aspects of the control application, and is independent of any implementation platform. In contrast, the implementation platform introduces certain design choices which determine whether all the computation tasks involved in the control application meet their deadlines. Task scheduling is a major concern in implementing a real-time control application. We verify whether the set of tasks in the control application is schedulable under the given platform constraints using non-preemptive scheduling with first-come-first-served and fixed-priority policies. We call this *scheduling verification*.

The challenge, therefore, is to *verify that the implementation of a control design meets the high-level timing and functional specifications of a control application*. In this paper, we present a method to carry out these verification tasks by building timed models of the real-time system (the implementation). As the implementation model, we take real-time systems written in Giotto [Henzinger *et al.* 2001, 2003]. Giotto provides a programming abstraction for hard real-time applications that exhibit time-periodic and multi modal behaviour, as in automotive, aerospace and manufacturing control. Fig. 1 explains the embedded control systems development process with Giotto [see Henzinger *et al.* 2001]. The control application development starts with the control engineer modelling the plant and designing the control laws using a tool such as Matlab. The control design is handed over to a software engineer, who implements the functionality and timing of the control design in Giotto. The Giotto program is then mapped to a given platform (hardware and RTOS) by the Giotto compiler, using optional annotations in the Giotto program which specify hardware characteristics, mapping of tasks to CPU's and the scheduling policy. The Giotto compiler actually generates code for a virtual embedded machine called E code. After linking, the E code supervises the execution of functionality code to guarantee the timing behaviour specified by the Giotto

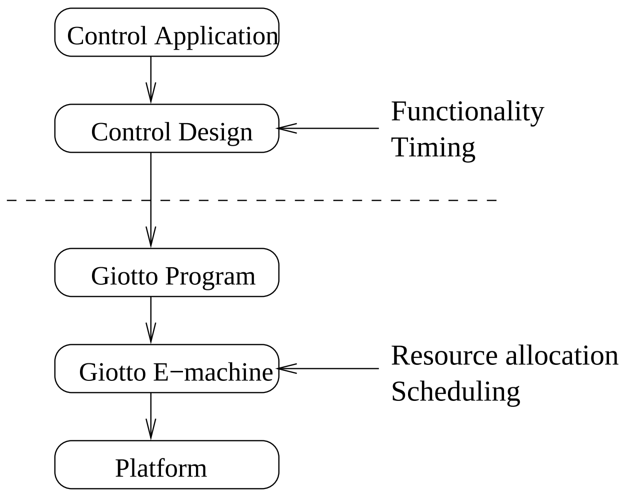


Fig. 1: Embedded control systems development with Giotto.

program. As can be seen from the figure, the goal of Giotto is to separate the functional and timing issues from the platform related issues. The programmer needs to specify only the functional details and provide the timing parameters like task frequency. The platform dependent issues like resource allocation and task scheduling are taken care of by the Giotto compiler using optional annotations provided by the software engineer. This enhances the flexibility in the design process by achieving platform independence, software portability, and re-use and reduces the system development complexity.

Although Giotto facilitates enhanced platform-independence, reusability and portability, it does not offer any guarantees that the functional and timing properties of the control design are preserved in the Giotto model or the generated code for a platform. The functionality implemented by the programmer is directly compiled to e-code, without verifying whether this actually implements the behaviour intended at the design time. Our first goal is to verify that the implementation satisfies the high-level functional and timing properties. We propose a scheme which translates a Giotto program into a network of timed automata, which can be analysed using UPPAAL [Behrmann *et al.* 2004]. With UPPAAL, we can verify the Giotto model for its functional and timing correctness. The proposed translation scheme can be used as an aid to debugging the system and locating the source of errors. Our second goal is to check for schedulability of the Giotto model on a given platform. To this end we extend the above scheme to verify the task scheduling under given platform constraints. In this paper we consider only non-preemptive scheduling with first-come-first-served and fixed-priority policies. For preemptive scheduling we can use the task automaton approach of Fersman *et al.* [2002, 2004], but that is beyond the scope of this paper.

Scheduling verification is performed by checking whether the timed automata model of the system can reach a *fail* state. In case all of the tasks cannot be scheduled, at least one of the tasks would be executing beyond its deadline. This situation will lead the system to enter the *fail* state. We try to detect whether the *fail* state is reached for all possible runs of the system. A run of the system which does not encounter the *fail* state gives a way to schedule the system in which all the tasks meet their deadlines.

Among related work, we cite the work on the TAXYS tool [see Bertin *et al.* 2001, Sifakis *et al.* 2003]. The aim of the TAXYS project is the design and validation of embedded real-time code for telecommunication applications. TAXYS builds a formal model of the entire real-time application by constructing timed automata for the system components as well as the environment. The inputs to TAXYS are the application and the environment specifications written in the synchronous programming language ESTEREL [Berry and Gonthier 1992] with additional real-time annotations. TAXYS translates the annotations to timed automata and uses the temporal model checker KRONOS to verify that the tasks corresponding to the external ESTEREL functions can be scheduled on a given platform. It is assumed that the execution time of the tasks have been instrumented via a profiling tool, and this data is part of the annotations.

Our work is very similar in spirit and objective to the TAXYS project. Instead of ESTEREL, we use Giotto models for verifying timing behaviour. The time-triggered nature of Giotto is more suitable for automotive, aerospace and industrial manufacturing applications. Both ESTEREL and Giotto involve timing abstractions – the *synchrony hypothesis i.e.*, zero execution time for functional code, in the case of ESTEREL, and *fixed logical execution time* with zero delay for reading sensors and writing actuators in the case of Giotto – that need to be verified for a given environment and execution platform. We do not model the environment explicitly, and assume that tasks are released periodically with a known frequency. Our verification has two distinct goals – the first is to check the functional and timing correctness of the Giotto model with respect to the requirements of the control application; the second is to verify the schedulability of the Giotto model on a given platform. The platform characteristics, such as WCET, form part of the Giotto specifications in the form of annotations, just as in the TAXYS tool. The objective of the TAXYS project is identical to our second goal. However, TAXYS does not have anything similar to our first objective of checking the functionality and timing of a real-time application model against the corresponding control requirements in its purview. Unlike TAXYS we do not have an automated tool to perform the translation to timed automata at present; it is part of our future work.

The rest of the paper is organised as follows. In Section 2, we describe Giotto and its modelling language. We present the basic translation scheme for functionality and timing verification in Section 3, the extended scheme for first-come-first-served and fixed-priority non-preemptive schedulability analysis in Section 4. Sections 5 and 6 describe applications of the schemes to the Giotto models for two applications. The paper concludes with the different perspectives of the scheme and possible extensions in Section 7.

2. Giotto

Giotto provides an abstract model and a tool-supported methodology for implementing complex real-time embedded systems on distributed platforms [Henzinger *et al.* 2001, 2003]. It is suitable for embedded control systems with multi-modal behaviour, as found in avionics, automotive and industrial control. It is based on the view that time-triggered task invocations and time-triggered mode-switches form “an abstract essence of programming real-time control systems”. The time-triggered nature of task invocations and communications implies that all these actions are triggered by the tick of a global clock. This assumption leads to timing predictability, which is essential for hard real-time control systems.

A Giotto model comprises both functionality and timing specifications of the system. A Giotto model can also be annotated with the target platform specification, to guide the Giotto E-machine to compile the system for a particular platform. The Giotto programmer need not specify how, when and where the tasks are scheduled. The Giotto compiler handles these implementation details using optional annotations provided by the programmer. Thus the Giotto methodology separates out the platform independent issues such as functionality and timing from the platform specific concerns such as resource allocation, scheduling and communication topology. This adds flexibility in the programming and enhances the portability of the system.

The key features of Giotto are periodic task invocations and time-triggered mode switches. A Giotto model specifies a set of modes. A mode depicts a high level state of the system. Each mode executes a set of tasks whose combined effect determines the behaviour of that mode. A task in the current mode is invoked at a constant frequency, as long as the program remains in the same mode. A mode switch has an associated frequency, the frequency at which a condition for the switch is evaluated. If the condition is true, the mode switch is executed. This brings a new set of tasks with their associated periods into execution.

A Giotto program also specifies sensor readings and actuator updates. The different constructs of a Giotto model communicate with each other by *drivers*. Drivers check the pre-conditions for the task executions, actuator updates and mode switches and accordingly execute or skip the tasks, updates and mode switches, respectively. Drivers also transport the values among ports. According to the Giotto semantics, a driver execution can not be interrupted as it is executed essentially instantaneously on the system level. However, a task, representing an application level computation, consumes a non negligible amount of time. Thus, the Giotto abstraction integrates scheduled computation (tasks) and instantaneous communication (drivers).

2.1 The Giotto language

The formal definition of the Giotto language is described in [Henzinger *et al.* 2003]. In practice, Giotto programs can be written in a concrete, C-like syntax. A *Giotto program* consists of the following components:

Ports Data communication among modes, tasks and other constructs of Giotto is done using ports. A port is nothing but a typed variable which is persistent in the sense that the port variable retains its value over time, unless it is updated. The ports in Giotto are partitioned into mutually disjoint sets of sensor ports, actuator ports, and task ports. The sensor ports are updated by the environment; all other ports are updated by the Giotto program. The task ports are further divided into task input ports, task output ports and task private ports. Giotto also has a set of mode ports (subset of task output ports) which are used to transfer data from one mode to the next: these are assigned a value every time the mode is entered. The task private ports determine the state of the task at any given point of time; these are accessible only within a task.

A port declaration $(p, Type, init)$ consists of a port name p, a type *Type*, and an initial value $init \in Type$.

Tasks and task invocations A task t in a Giotto program has a set of input ports and a set of output ports. Output ports can be shared by tasks as long as they are not invoked in the same mode but every task has to use its own set of input ports. A task declaration $(t, In, Out, Priv, f)$ consists of a task name t, a set $In \subseteq InPorts$ of *input ports*, a set $Out \subseteq OutPorts$ of *output ports*, a set $Priv \subseteq PrivPorts$ of *private ports*, and a *task function* $f : Vals[In \cup Priv] \to Vals[Out \cup Priv]$. $Vals[P]$ stands for the set of valuations for P. The task function f is implemented by a sequential program. Giotto supports C and JAVA for implementing the task function.

Giotto tasks are periodic. A task invocation specifies a task frequency (a non-zero natural number), a task and a driver. The task driver reads the sensor and mode ports and provides the values for the task input ports. The driver is guarded, the guard of a driver being a boolean formula on sensor and mode ports. The invoked task is executed only if the driver guard evaluates to true. The task function computes the values for the task output ports. The Giotto semantics specifies two phases of a task invocation – a communication phase, *i.e.*, the driver execution, which takes logically zero time, and a computation phase, *i.e.*, the task function execution, which actually consumes time. According to Giotto semantics, the output values of the invocation of a task must not be available before its logical execution period expires. This behaviour of Giotto is referred as Fixed Logical Execution Time (FLET) concept.

Modes On top of all constructs, a Giotto program has a set of modes. Each mode has a fixed set of tasks which are invoked periodically. A Giotto program is in one mode at a time. A mode is repeated after a fixed period of time i.e. a mode execution is also periodic. The possible transitions from one mode to other modes are specified by mode switches. A task may be invoked in more than one mode, but its period remains the same in each mode.

Formally, a mode consists of a mode period, a set of mode ports, a set of task invocations and a set of actuator updates. Whenever a mode is entered for the first time, its mode ports are initialised by the mode switch driver function. The actuators in a mode are periodically updated. We assume that a mode switch does not logically interrupt the task invocation *i.e.* a task must not be running when a mode switch occurs.

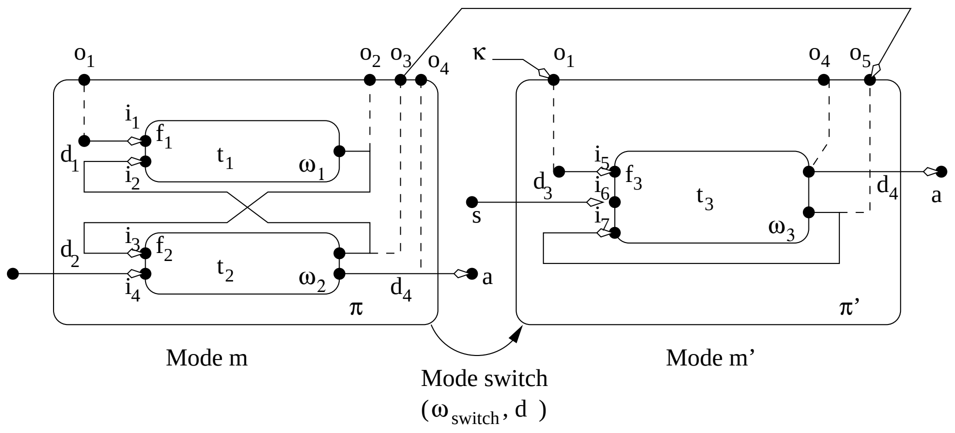


Fig. 2: An example Giotto model.

2.2 An example Giotto model

Fig. 2 shows an example Giotto model. It has two modes of operation m and m'. The mode m consists of two tasks t_1 and t_2. It has four mode ports o_1, o_2, o_3 and o_4. A task driver reads the mode and sensor ports and initialises the input ports of a task. In the given model, the task driver d_1 reads the mode ports o_1, o_3 and initialises the input ports i_1 and i_2 of the task t_1. The task driver d_2 reads the mode port o_2 and sensor port s and initialises input ports i_3 and i_4 of the task t_2. The two tasks are implemented by the functions f_1 and f_2 and have frequencies ω_1 and ω_2. The mode m has a mode period of π ms. The mode m' has one task t_3 and a mode period of π' ms. This system has one actuator a which is updated by actuator update driver d_4 in both modes.

There is a mode switch from the mode m to the mode m', specified by mode switch driver d_5. The driver d_5 initialises the mode ports o_1, o_4 and o_5 of target mode m'. The mode port o_1 is initialised by a constant (denoted by κ), o_5 is initialised by the value of the mode port o_3 and mode port o_4 retains its value.

Every driver d (whether a task driver, an actuator update driver or a mode switch driver) has a driver guard g_d which is a boolean formula representing the precondition for the corresponding action, and a driver function f_d which generally updates the values of the target ports if g_d evaluates to *true*. If g_d is *false*, the corresponding action is ignored.

3. From Giotto to UPPAAL: functionality and timing

In this section, we propose a translation scheme from Giotto to UPPAAL for the verification of Giotto models. According to the translation scheme every component of a Giotto model (e.g. task invocation, driver, mode-switch) is translated to a network of timed automata in UPPAAL.

UPPAAL [Behrmann *et al.* 2004] is a toolbox for modelling, simulation and verification of real-time systems jointly developed by Uppsala University and Aalborg University. It has been applied successfully in case studies ranging from communication protocols to multimedia applications [Hune *et al.* 2000, Lindahl *et al.* 2001, Iversen *et al.* 2000, David and Yi. 2000]. UPPAAL is based on the theory of timed automata [Alur and Dill 1990, 1994]. The timed automata in UPPAAL are equipped with bounded integer variables, structured data types, channel synchronisation, channel and process priorities. A system is modelled in UPPAAL as a network of timed automata; UPPAAL uses the CCS parallel composition operator [Milner 1989] to compute the product automaton on-the-fly during verification. The query language of UPPAAL, used to specify properties to be checked, is a subset of CTL (computation tree logic) [Clarke *et al.* 1999].

Before we explain the translation scheme, we show a UPPAAL network of timed automata for the Giotto model given in Fig. 2 resulting from the application of the translation scheme. We have the following timed automata.

- We translate the modes and mode switches into one timed automaton as shown in Fig. 3. The given model has two modes and one mode switch from mode m to the mode m'. The corresponding timed automaton shows two locations, one for each mode. It has one edge from mode m to mode m' representing the mode switch. The locations have self loops indicating the fact that the system may stay in the current mode at the end of the mode period. The mode switch is enabled if the mode switch driver guard g_{d_5} of the driver d_5 evaluates to *true*. A mode switch driver guard represents the exit condition of the mode. It is a boolean-valued condition on the sensor ports and the mode ports of the mode. The mode switch occurs only when the exit condition evaluates to *true*. The mode switch can be enabled only when one mode period is finished, as represented by the clock guard $x = \pi$ and the invariant $x \leq \pi$. If the mode switch edge is taken, the driver function f_{d_5} is executed; this initialises the mode ports of the target mode m'. The clock x is reset, the flags (which are explained below) used in the target mode are reset to *zero*, and the variable *currmode* is set to target mode m'.
- Fig. 4 and Fig. 5 show the timed automata for the task driver d_1 and the task function f_1 of the task t_1. We have similar automata for the other tasks as well. Every task in the Giotto model is translated into two timed automata – one for the task driver and the other for the task function. The automaton for

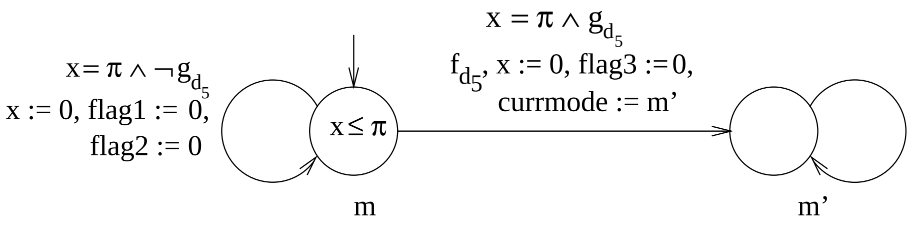


Fig. 3: Timed automaton for Giotto mode switch.

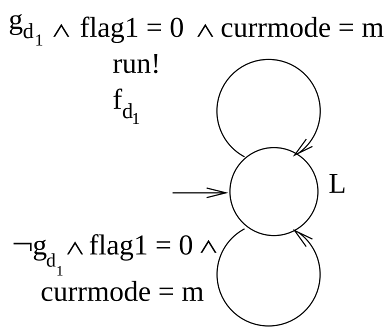

$g_{d_1} \wedge flag1 = 0 \wedge currmode = m$
run!
f_{d_1}
L
$\neg g_{d_1} \wedge flag1 = 0 \wedge$
currmode = m

Fig. 4: Timed automaton for Giotto task driver.

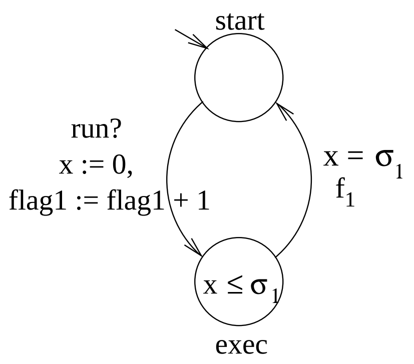


Fig. 5: Timed automaton for Giotto task function.

the task driver has one location and two edges – one edge is enabled (referred to as positive edge) when the task driver guard g_{d_1} for the task driver d_1 is *true*, otherwise the other edge (referred to as negative edge) will be taken. The guard on the edge checks if this task is to be executed in the current mode m ($currmode = m$).

The invariant on location *exec* of the task function automaton and the clock guard on the outgoing edge forces the task execution to last for its logical execution period (σ_1 is the logical execution period for task t_1). The clock x is reset and task function f_1 provides the output values. The synchronisation channel *run*! calls the task function automaton.

We use the integer variables *flag*i to control the task executions. Consider a scenario in which a mode has a mode period of 100ms. This mode consists of one task whose task period is also 100ms and an actuator whose update frequency is 1. A task is translated to two automata – one for the task driver and other for the task function. The two automata are connected through a synchronisation channel in which the task driver is at the calling end and the task function is at the callee end. For the above specification of the system, at time 100ms, four processes are enabled – (a) a task function execution, (b) an actuator update, (c) a mode switch and (d) a task driver execution for the next mode period (a mode repeats execution of tasks in every mode period). Now, according to the Giotto semantics, process (a) has the highest priority and it must take place first, followed by process (b), then by process(c) and finally by process (d). We specify the same priority order in UPPAAL for the above processes. But, since the task function and the task driver automata are connected through a synchronisation, after the task function executes, UPPAAL gives preference to the task driver over the actuator update and mode switch. Hence the actual priority order is violated. To overcome this problem, we use one integer variable *flag*i for each task, which is incremented by the task function automaton as soon as it starts executing. The task driver execution begins only when the *flag* is *zero*. This is again set to *zero* by the mode switch automaton, hence forcing the task driver execution only after the mode switch to maintain normal priority order.

- Other constructs of Giotto model – actuator updates and sensor readings – are also translated into timed automata. These are explained later on.

The Giotto semantics prescribes the priorities among the various actions. Task functions have the highest priority. Next comes the actuator update, followed by the sensor readings. This is followed by a mode switch, which has precedence over a task driver execution. These priority orders can be specified for different timed automata in the system definition part of the UPPAAL specification.

We develop the translation scheme from Giotto to UPPAAL in two phases. The first considers the basic constituents of Giotto model and translates the detailed computation steps such as task functions and various driver functions. The modelling of detailed computation steps is essential for verification of the functional and timing specification. To do this, the complete system behaviour has to be embedded in the resulting system of timed automata. The second goal of our verification is to analyse schedulability with respect to a particular platform using non-preemptive scheduling with first-come-first-served and fixed-priority policies. This requires appropriate modifications to the former scheme in order to model features of the platform. The translation schemes are based on the syntax of Giotto components.

We divide the all Giotto constructs into four sets:

- $\mathbb{MS}$: set of mode switches.
- $\mathbb{TI}$: set of task invocations.
- $\mathbb{AU}$: set of actuator updates.
- $\mathbb{SR}$: set of sensor readings.

All ports declared in the Giotto program are denoted by typed variables in the UPPAAL specification. The translation scheme is defined as a set of four functions, where each function maps a Giotto construct to one or more timed automata. In the following $\mathbb{TA}$ denotes the set of timed automata.

- *Mode translation function* $\Phi_{MS} : \wp(\mathbb{MS}) \to \mathbb{TA}$, *i.e.*, a set of mode switches is translated to a single timed automaton.
- *Task invocation function* $\Phi_{TI} : \mathbb{TI} \to \mathbb{TA} \times \mathbb{TA}$.
- *Actuator update function* $\Phi_{AU} : \mathbb{AU} \to \mathbb{TA}$.
- *Sensor reading function* $\Phi_{SR} : \mathbb{SR} \to \mathbb{TA} \times \mathbb{TA}$.

In the remaining part of this section, we illustrate these functions. We use the following notations in the rest of the paper:

A clock guard g_c is a conjunctive formula of atomic constraints of the form $x \sim n$ or $x - y \sim n$ for $x, y \in C$, $\sim \in \{\leq, < . =, >, \geq\}$ and $n \in \mathbb{N}$. C is a finite set of real-valued variables standing for clocks. We use $\mathcal{B}(C)$ to denote the set of clock constraints.

A mode guard g_m is a disjunction of boolean expressions of the form '*currmode = MODE*', *MODE* being the mode in which the given task occurs.

A driver guard in Giotto is a boolean function which is a precondition for the corresponding function execution. We have three driver guards – *mode switch driver guard* g_{md}, *task driver guard* g_t and *actuator update driver guard* g_a.

A clock reset r_c resets one or more clocks to zero.

Flag reset(s) flag$_r$ resets certain flags to zero. The flags are the integer variables used for controlling the task invocations in a mode as explained earlier in this section.

Actuator update function f_a updates the actuator ports.

Driver function f_d is executed whenever a driver guard evaluates to *true* and generally initialises the target ports.

Task function f_t implements the task.

The variable *currmode* is of type bounded integer and is used to keep track of the current mode. The symbol $\top$ denotes the boolean *true* and we use *none* for an empty action. Clocks are represented by letters *x, y, z etc.*.

The following subsections will describe each function. We will explain the resulting timed automaton. In UPPAAL a timed automaton over an action alphabet Σ and a set of clocks C is defined as a tuple $\langle N, l_0, E, I\rangle$ where

- N is a finite set of locations (or nodes),
- $l_0 \in N$ is the initial location,
- $E \subseteq N \times \mathcal{B}(C) \times \Sigma \times 2^{C} \times N$ is the set of edges and
- $I : N \longrightarrow \mathcal{B}(C)$ assigns invariants to locations.

We shall denote an edge by a tuple $\langle l, g, a, r, l'\rangle$ where

l	is the starting location;
g	is a guard;
$a \in \Sigma$	is an synchronisation action;
r	is a set of clock resets, extended with assignments and updates; and
l'	is the ending location.

3.1 *Mode translation function* Φ_{MS}

At the top level of a Giotto specification, is a mode which consists of task invocations, actuator updates and mode switches. These constituents of a mode are declared in other parts of the specification. We shall take mode switches into consideration in this part of the translation. Task invocations and actuator updates will be dealt with in the following sections. We shall translate the set of mode declarations (mode switches in particular) into a single timed automaton; an example is shown in Fig. 6. The configuration of the system being translated, at any point of time, can be determined by this timed automaton.

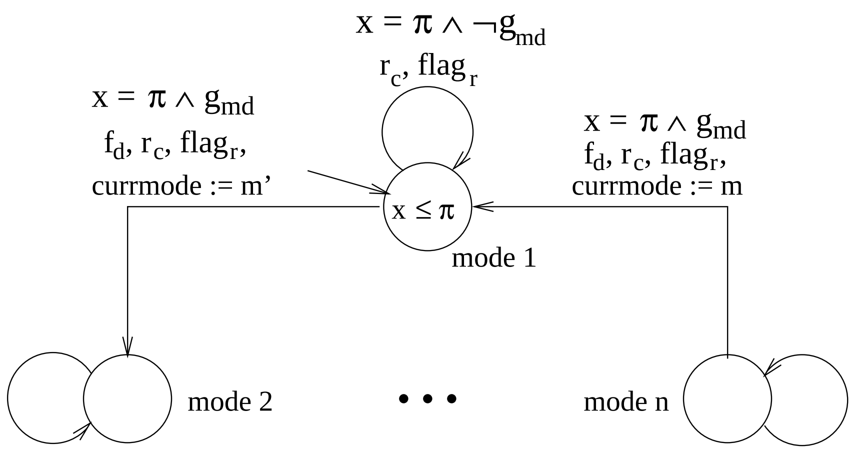


Fig. 6: Mode switch.

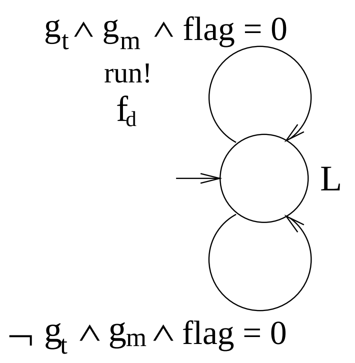
Each mode switch (in Giotto) $(\omega_{switch}, m', d) \in Switches[m]$ consists of a *mode-switch frequency* $\omega_{switch} \in \mathbb{N}$, a *target mode* $m' \in Modes$, and a *mode driver* $d \in Drivers$.

The timed automaton $\langle N,\ l_0,\ E,\ I\rangle$, which results from applying Φ_{MS} to a given collection of mode switches, is defined by

- Each location in N corresponds to a mode in the system.
- The initial location l_0 corresponds to the initial mode.
- There are two types of edges in E – one edge for each mode switch and one self loop on every location i.e. mode.
  (1) An edge corresponding to a mode switch takes the control to target mode and is of the form
  $\langle m,\ (g_c\ \wedge\ g_{md}),\ none,\ r,\ m'\rangle$
  where the set r includes clock resets r_c, driver function f_d, flag resets $flag_r$ and an assignment $currmode := m'$, which sets the *currmode* to the target mode. The flag reset $flag_r$ resets all flags used for controlling the task invocations in a mode to *zero*. This automaton does not have any synchronisation action.
  (2) A self loop edge is taken to keep the control in the current mode, when a mode switch is not enabled. This edge is of the form
  $\langle m,\ (g_c\ \wedge \neg g_{md}),\ none,\ (flag_r,\ r_c),\ m\rangle.$
- There is one clock invariant $I(l)$ on each location l of the form *clock <= mode period*. This, coupled with the clock guard g_c on edges, ensures that a mode switch occurs, if at all, exactly at the end point of a *mode period*.

3.2 Task invocation function Φ_{TI}

A mode specifies the invocation of the tasks it contains. A task invocation (ω_{task}, t, d) consists of a task frequency ω_{task}, a task t and a driver d. We use σ to denote



$g_t \wedge g_m \wedge flag = 0$
run!
f_d
L
$\neg g_t \wedge g_m \wedge flag = 0$

Fig. 7: Task driver.

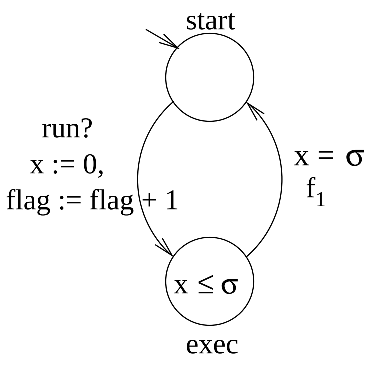


Fig. 8: Task function.

task period (= mode period/ω_{task}). We have two timed automata (shown in Fig. 7 and Fig. 8) for two parts of a task invocation – task driver and task function.

The timed automaton $\mathbb{A}_{td}$ for the *task driver* has

- one location L,
- the initial location is L,
- two edges:

  (1) $\langle L,\ (g_t \wedge g_m \wedge flag = 0),\ run!,\ f_d,\ L\rangle$, the synchronisation action *run* calls the corresponding task function automaton.

  (2) $\langle L,\ (\neg g_t \wedge g_m \wedge flag = 0),\ none,\ \emptyset,\ L\rangle$.

  The integer variable *flag* controls the execution of task driver at the start of a mode period. The first edge is taken when driver guard evaluates to *true*. In this case, this edge is synchronised with the first edge of the task function automaton.

- an empty invariant (*i.e.* $\top$) in every location.

The timed automaton $\mathbb{A}_{tf}$ for the *task function* has the following components:

- two locations *start* and *exec*,
- the initial location *start*,
- two edges:

  (1) $\langle start,\ \top,\ run?,\ (r_c,\ flag := flag + 1),\ exec\rangle$: this edge is synchronised with the edge of the task driver automaton.

  (2) $\langle exec,\ g_c,\ none,\ f_t,\ start\rangle$.

  Here f_t is the task function associated with the task. The task function is performed on the second edge to match with the FLET concept of Giotto specification – the output values of the function should not be available before its logical execution time finishes. The variable *flag* is incremented on the first edge if task frequency is *one*, otherwise it will be incremented on the second edge.

- The automaton has one invariant on location *exec* to ensure the task execution for the corresponding task period.

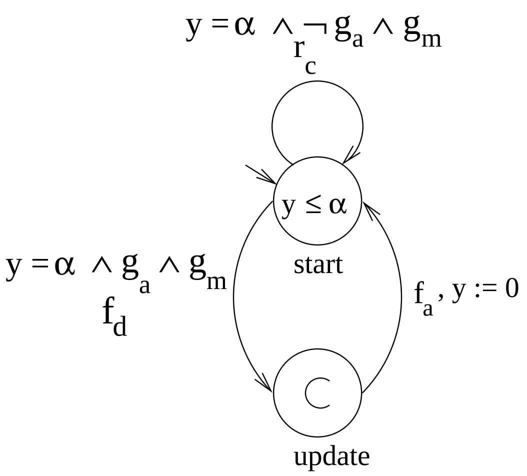


Fig. 9: Actuator update.

3.3 *Actuator update function* Φ_{AU}

An actuator update is similar to a task invocation except that the former takes logically zero time. An actuator update (ω_{act}, d) consists of an update frequency $\omega_{act} \in \mathbb{N}$ and a driver d. We denote the interval between two consecutive actuator updates by α (= mode period/update frequency). The function which updates the actuator is specified within the actuator declaration part of the Giotto specification. The timed automaton for an actuator update (see Fig. 9) consists of

- two locations *start* and *update*,
- the initial location *start*,
- three edges:
  (1) $\langle start,\ (g_c \wedge\ g_a \wedge g_m),\ none,\ f_d,\ update \rangle$.
  (2) $\langle update,\ \top,\ none,\ r,\ start \rangle$, the set r includes clock resets r_c and actuator update function f_a. The location *update* is committed, so no waiting is allowed in this location.
  (3) $\langle start,\ (g_c\ \wedge\ \neg g_a \wedge g_m),\ none,\ r_c,\ start \rangle$, this edge is taken when actuator driver guard evaluates to false and actuator is not be updated.
- an invariant in location *start* to ensure actuator update at the right point of time.

3.4 *Sensor reading function* Φ_{SR}

We will have two timed automata for each sensor – one for the environment of the sensor and another for the sensor's controller. Whenever there is any value for the sensor, the environment automaton will send the value to the controller automaton which, in turn, will save the sensor value in variables to be used later on. If the sensor is of periodic time triggered nature, the environment automaton will have only one edge. If the sensor reading is non-periodic in nature, the environment automaton will have more than one edge. Every edge of this automaton is synchronised with the only edge of the controller automaton.

4. From Giotto to UPPAAL: task scheduling

In this section, we extend the translation scheme to Giotto implementations on a given platform, which requires us to consider the resources available on the platform in performing scheduling verification. Specifically, we look into the non-preemptive scheduling of tasks using first-come-first-served and fixed-priority schemes on a particular platform. Given a Giotto model, the worst case execution time (WCET) of tasks and the resources available (this is what we call a platform), we translate them to a network of timed automata and verify whether the given system is schedulable under the given platform constraints. If the answer to the verification question is yes, we get a possible schedule and if not, we can locate the source of error.

In this translation scheme, functions which actually implement the tasks, are not translated, as here we are only interested in their timing properties like task frequency and WCET. Further, we don't require sensor readings any more, for we are not concerned about functionality. Instead, we have timed automata for resource management. We explain the scheme with one type of resource. It can be extended for multiple resource types by having one timed automaton for each resource type. We omit all details about driver guards, driver functions and task functions from the automata in the following discussion.

We define a function Φ_{res}, mapping a set of resources to a set of timed automata, one automaton for one type of resource. The set of all resource types is denoted by $\mathbb{RES}$. This part of the translation scheme is defined as a set of four functions, where each function maps a Giotto construct to one or more timed automata.

- *Mode translation function* $\Phi_{MS} : \wp(\mathbb{MS}) \rightarrow \mathbb{TA}$.
- *Task invocation function* $\Phi_{TI} : \mathbb{TI} \rightarrow \mathbb{TA} \times \mathbb{TA}$.
- *Actuator update function* $\Phi_{AU} : \mathbb{AU} \rightarrow \mathbb{TA}$.
- *Resource management function* $\Phi_{res} : \mathbb{RES} \rightarrow \mathbb{TA}$.

4.1 Resource management function Φ_{res}

The corresponding timed automaton for one resource type is shown in Fig. 10. It consists of

- four locations $\{L, A, B, C\}$,
- the initial location L, with all locations other than L being committed,
- E, the following list of edges:

  (1) $\langle L,\ \top,\ \textit{request}?,\ \emptyset,\ A\rangle$
  (2) $\langle A,\ \textit{isFree}(),\ \textit{ack}?,\ \textit{alloc}(),\ L\rangle$
  (3) $\langle A,\ \neg\textit{isFree}(),\ \textit{wait}?,\ \textit{add2wait}(),\ L\rangle$
  (4) $\langle L,\ \top,\ \textit{release}?,\ \emptyset,\ B\rangle$

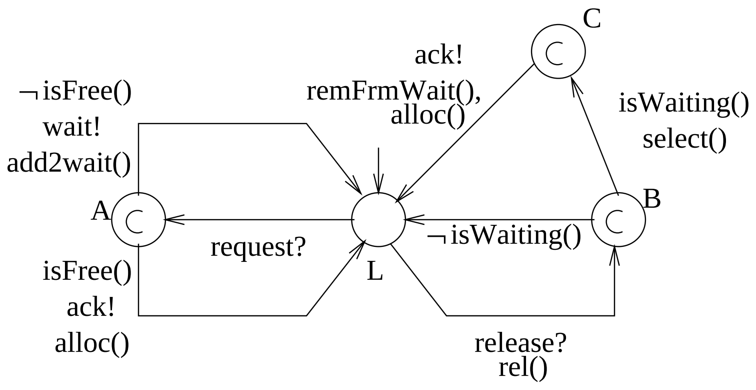


Fig. 10: Resource management.

(5) $\langle B,\ \neg isWaiting(),\ none,\ \emptyset,\ L\rangle$
(6) $\langle B,\ isWaiting,\ none,\ select(),\ C\rangle$
(7) $\langle C,\ \top,\ none,\ (remFrmWait(), alloc()),\ L\rangle$

The edge 1 is taken when a request for the resource arrives, leading to location *A*. The function *isFree*() checks if any resource is available. If it is available, it is allocated through edge 2 using function *alloc*(). The function *alloc*() updates an array *resToProcess*[] of size equal to the number of resource instances. When a resource is allocated, the corresponding process's id is entered into this array. If the resource is unavailable, the requesting task is added to the wait list (edge 3) by function *add2wait*(). This function updates an array which has the id's of the waiting processes. Now, when any task releases the resource, edge 4 is taken. The function *rel*() removes the process id from the array *resToProcess*[]. The function *isWaiting*() checks if any task is waiting for the resource by looking in the waiting list array. If a task is waiting, edge 6 selects one of the waiting tasks and edge 7 allocates the freed resource to this task. If no task is waiting for the resource, edge 5 simply returns the control to initial location.

- This automaton has no invariant.

This automaton maintains a variable *nop* for the number of tasks to which resources are allocated, and a variable *wp* denoting the number of processes waiting for the resources.

The resources are allocated on first-come-first-served basis or by a fixed-priority policy. If several processes are waiting for the resource, we can select one using any strategy. Once a resource is allocated, it is released only after the process finishes its execution (as is shown in the task function automaton later in this section). Thus the scheme captures a non-preemptive scheduling strategy.

4.2 Mode translation function Φ_{MS}

This function is similar to the one described in Section 3. A variable *notk* is used here to denote the number of task instances in one mode period. Here, by a task instance we mean an execution of a task in one mode period. After every mode period, this is set to number of task instances in the target mode. The variable *notk* is decremented in the task automaton. As shown in Fig. 11, if after a mode period, *nop* or *notk* is non-zero, the control will lead to the *fail* location from where control cannot escape. This non-zero value shows that some tasks are running while the mode is being switched and this violates our assumption that the mode period should be an integer multiple of the LCM of all task periods in that mode. This leads to the conclusion that the system is not schedulable under the given platform constraints. The timed automaton has the following components:

- The locations correspond to the modes of the system. In addition there is a *fail* location.
- The initial location corresponds to the initial mode.
- There are three types of edges in this set – one edge for each mode switch, one self loop on every location *i.e.* mode, and one edge leading to the *fail* location from every mode.
  1. The first kind of edge takes the control to the target mode and is defined by

     $$\langle m,\ (g_c\ \wedge\ nop = 0\ \wedge\ notk = 0),\ none,\ r,\ m'\rangle$$

     where the set r consists of clock resets r_c, flag resets $flag_r$ and assignments $currmode := m'$ and $notk := n[m']$, where $n[m]$ denotes the number of task instances in mode m.

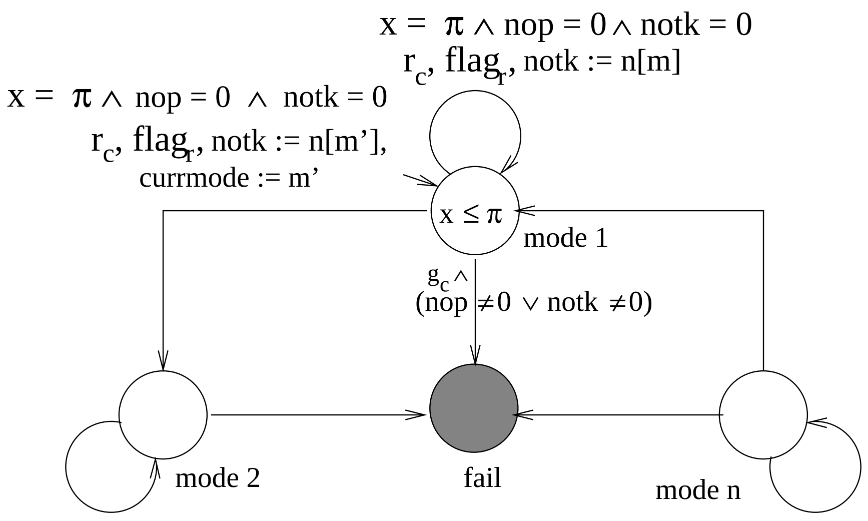

x = π ∧ nop = 0 ∧ notk = 0
r_c, flag_r, notk := n[m]
x = π ∧ nop = 0 ∧ notk = 0
r_c, flag_r, notk := n[m'],
currmode := m'
x ≤ π
mode 1
g_c ∧
(nop ≠0 ∨ notk ≠0)
mode 2
fail
mode n

Fig. 11: Mode switch.

(2) The self loop edge is taken to keep control in the current mode, when a mode switch is not enabled for the current mode. This edge is defined as

$\langle m,\ (g_c \ \wedge\ nop = 0 \ \wedge\ notk = 0),\ none,\ r,\ m\rangle$. where the set r includes the clock resets r_c, flag sets $flag_r$ and assignment $notk := n[m]$.

(3) The third kind of edge, corresponding to the fail condition, is defined as

$\langle m,\ (g_c \ \wedge\ nop \neq 0 \ \vee\ notk \neq 0),\ none,\ \emptyset,\ fail\rangle$.

- There is one clock invariant on each location of the form *clock* $\leq$ *mode period*. This, coupled with the clock guard g_c on edges, ensures that the mode switch occurs, if at all, exactly at the end point of *mode period*.

4.3 Task invocation function Φ_{TI}

We have two timed automata (shown in Fig. 12 and Fig. 13) for the two parts of a task invocation – task driver and task function.

The timed automaton $\mathbb{A}_{td}$ for the task driver has

- four locations *start*, *A*, *B* and *waiting*,
- the initial location is *start*,
- the edges listed below

(1) $\langle start,\ (\ g_m \ \wedge\ flag = 0),\ request!,\ r_c,\ A\rangle$,
(2) $\langle A,\ \top,\ ack?,\ \emptyset,\ B\rangle$,
(3) $\langle A,\ \top,\ wait?,\ \emptyset,\ waiting\rangle$,
(4) $\langle waiting,\ \top,\ ack?,\ \emptyset,\ B\rangle$,
(5) $\langle B,\ \top,\ run!,\ \emptyset,\ start\rangle$.

A task requests the resource through edge 1. If the resource is available, it is acknowledged by the *resource* automata on edge 2 and then the task driver calls the task function by *run* on edge 5. If the resource is currently unavailable, edge 3 takes the control to the *waiting* state. Edge 4 is enabled when the resource becomes available to this task.

- this automaton has no invariant.

The timed automaton $\mathbb{A}_{tf}$ for the task function has

- three locations *start*, *pass* and *exec*,
- the initial location *start*,
- three edges:

(1) $\langle start,\ \top,\ run?,\ (r_c,\ flag := flag + 1),\ exec\rangle$: this edge is synchronised with the edge of the task driver automaton.
(2) $\langle exec,\ g_c,\ release!,\ \emptyset,\ pass\rangle$,
(3) $\langle pass,\ g_c,\ none,\ notk := notk - 1,\ start\rangle$.

The clock guard on edge 2 and invariant on location *exec* ensure the task executes for its worst case execution time, after which the resource is released by edge 2. Then, the task passes the remaining time on location *pass* because

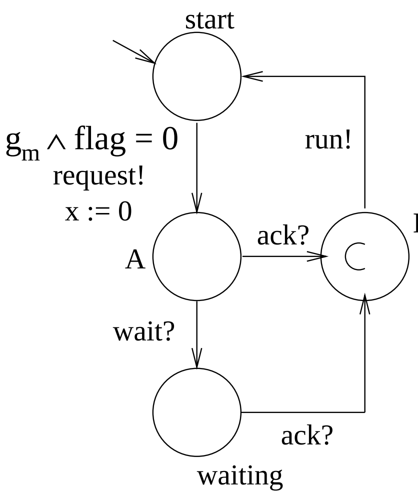


Fig. 12: Task driver.

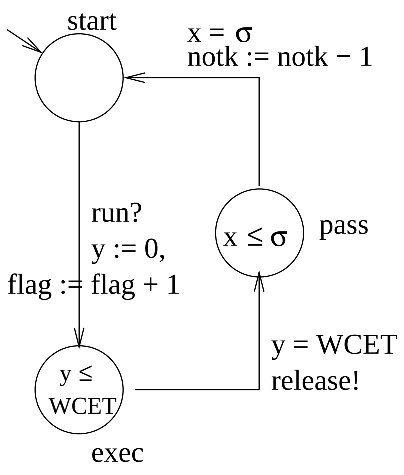


Fig. 13: Task function.

the task output should be available only after its logical period expires. If the task frequency is *one*, *flag* is incremented on edge 1, else it is incremented on edge 3.

- two invariants on locations *exec* and *pass* to ensure the task execution for the corresponding task period.

4.4 Actuator update function Φ_{AU}

This function is exactly the same as that of the scheme in the previous section.

5. Case study: an elevator system

We illustrate the first translation scheme in Section 3 for verification of functionality, by applying it to an existing Giotto model for an elevator system (provided along with Giotto tool distribution, see `http://embedded.eecs.berkeley.edu/giotto/demo.html`). The system is verified against some properties using UPPAAL.

The system modelled in Giotto consists of one elevator which serves five floors. Each floor has a request button that a user presses to get the elevator to come to that floor and open its doors. This system is subject to verification of the following properties

Safety The elevator never moves with its doors open.

Liveness The requests to be delivered from a particular floor are eventually serviced.

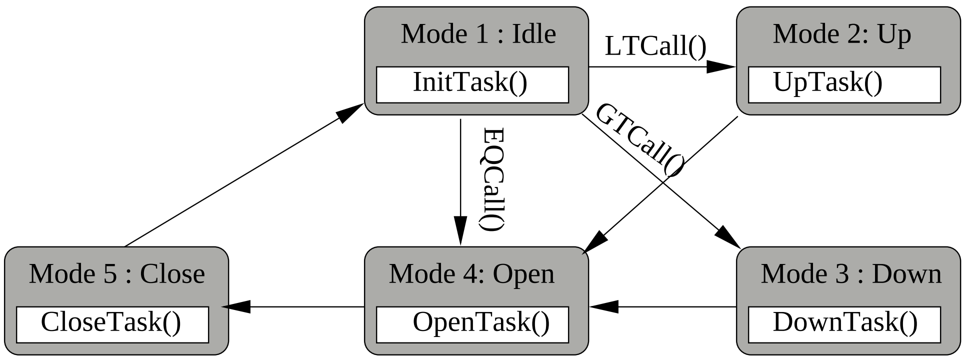


Fig. 14: Components of a elevator system.

5.1 The structure of the elevator system

The elevator system has five modes of operation (see Fig. 14). In each mode *Idle*, *Up*, *Down*, *Open* and *Close*, there is one task which has frequency 1. The system has one sensor to read the requests from each floor using *GetButtons*. There are two actuators, *Motion* and *Door* which use the update functions *PutMoveMotor* and *PutDoorMotor*. Each actuator is updated once in one round of every mode. The elevator is in *Idle* mode initially. There are three possible mode switches from this mode depending upon three mode switch drivers *PGTC*, *PLTC*, *PEQC* each of which uses one condition *CondPosGTCall*, *CondPosLTCall* and *CondPosEQCall*.

If the elevator is at the requested floor, the system control moves to *Open* mode, else it moves either to *Up* or *Down* mode. *Up* mode has one task *TaskUp*, which sets the required parameter *tmotion* to be read by actuator drivers *Move* and *Door*. Similarly, *Down* has one task *TaskDown* which sets the required parameters to take the elevator down. After reaching the required floor, the control moves to the *Open* mode, in which the task *TaskOpen* sets the *tdoor* parameter. This mode has a switch to the *Close* mode, which in turn switches to the *Idle* mode. The period of each mode is 500ms and every mode has one task with frequency 1, so at a time only one task is executed. Each task has a period of 500ms.

5.2 Translating Giotto model into network of timed automata

We show the translation to UPPAAL for the elevator system described in Section 5.1 using the translation scheme described in Section 3. The system has five tasks, so we have ten automata for the task invocations. Fig. 15 and Fig. 16 stand for the task *Idle*. These and the following figures are UPPAAL timed automata which are slightly different from the timed automata used in the earlier sections. In an UPPAAL timed automaton, the initial location is double circled and a C-like notation is used for guards and assignments instead of standard mathematical notation. The *currmode* variable is used to keep track of the current mode. An integer variable *flag* ensures the execution of a task only after the mode switch drivers are executed and is incremented by the task function automaton. This variable is reset by the

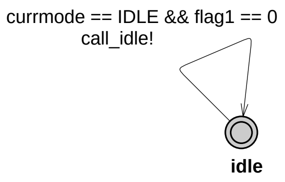


Fig. 15: Task driver for task *Idle*.

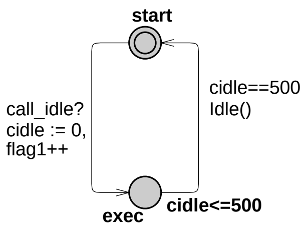


Fig. 16: Task function for task *Idle*.

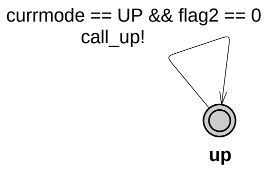


Fig. 17: Task driver for task *Up*.

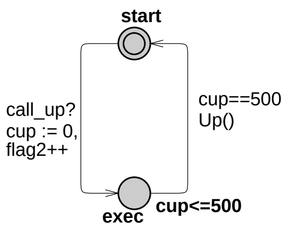


Fig. 18: Task function for task *Up*.

mode switch automaton.

Fig. 17 and Fig. 18 refer to the task *Up*. Other task invocations have similar translations.

Fig. 19 and Fig. 20 represent the timed automata for the actuator updates. According to Giotto semantics, an actuator update is an instantaneous communication and takes logical zero time. This is exhibited by the committed locations in the two timed automata. The *updatemotion* and *updatedoor* actions represent the actuator functions *PutMoveMotor* and *PutDoorMotor*.

Fig. 21 is a timed automaton corresponding to all the mode switches. Since the elevator system has five modes, the automaton has five locations. Every location has one self edge to represent the fact that control is resumed in the same mode. Edges between different locations represent mode switches. Each such edge first checks the mode switch driver guard. If a mode switch edge is taken, the *currmode*

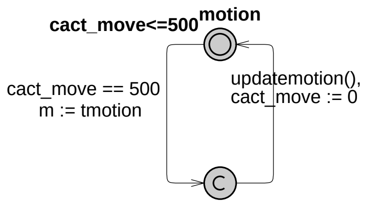


Fig. 19: Actuator *Motion*.

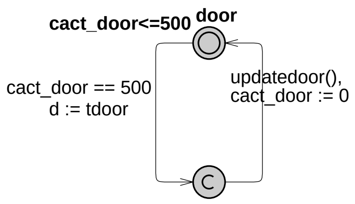


Fig. 20: Actuator *Door*.

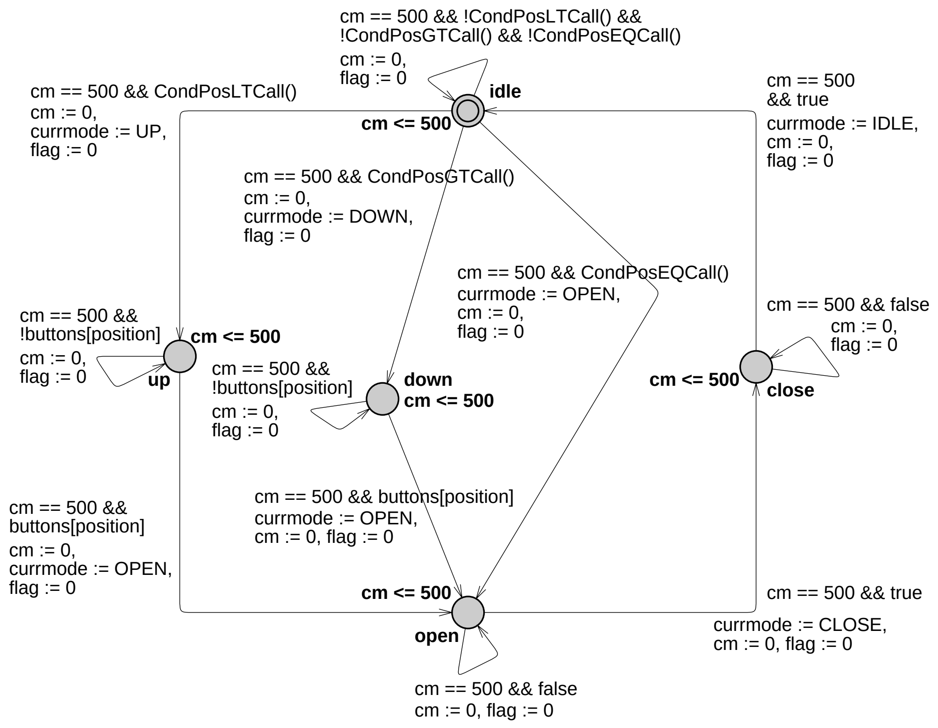


Fig. 21: Timed automaton for mode switches of elevator system in Fig. 14.

variable is updated and the clocks used in the target mode are reset. This automaton represents the controller of the whole system.

Sensors are updated by the environment. The environment can provide the values periodically or non periodically, but Giotto samples the sensors periodically. We used several different sensor reading timed automata to simulate the environment's periodic and non periodic behaviour.

5.3 *Results*

The system expressed in Giotto is vulnerable to a starvation problem *i.e.* the liveness property described above doesn't hold for the system. This is because of the Giotto policy of determining a mode switch. According to the Giotto semantics, all mode switches are deterministic *i.e.* at a given point of time, only one mode switch is enabled. Now consider the following scenario. The elevator is in *Idle* mode at the third floor (floor 2) and there are requests from floor 0 and floor 4. The Giotto policy is to first service the requests from the floor which is above the current floor, so the elevator reaches the fifth floor. Right after the elevator reaches the fifth floor, there is a request from the fourth floor. The elevator comes down to the fourth floor. Meanwhile, a request comes from the fifth floor (recall, the request from the ground floor has not been serviced yet). After coming to the fourth floor, the system checks for any request from the above floors and finds one from the fifth, so it goes to the fifth floor. Now, if this sequence of requests from the fifth and the fourth floors is continued with a very short time delay, the elevator will never reach the floor 0, hence violating the liveness property.

The other flaw we detected in the elevator system is an unwarranted time delay of one period of *Idle* mode. After reaching the mode *Close*, the system goes to the *Idle* mode irrespective of any pending request. According to our analysis, the system may directly jump to the *Up* or *Down* mode according to the request, thus saving one *Idle* mode period time.

6. Case study: Hovercraft

Here we apply the second translation scheme for schedulability analysis in Section 4 to a hovercraft system in Giotto [Marco *et al.* 2004]. This system shows more complex behaviour than the elevator system, in terms of number of tasks a mode has and the control laws that govern the system behaviour. As described in Section 4, the computational steps in the task functions are omitted.

In this system, the user provides a location that serves as a target for the hovercraft, which will move to that selected location under its own power. A Giotto model for controlling a hovercraft simulator with two degrees of freedom is described in [Marco *et al.* 2004]. The program computes the position of the hovercraft, computes the power for the left and right jet engines and then moves the hovercraft toward the target destination and orientation.

6.1 The structure of Hovercraft system

Fig. 22 illustrates the structure of hovercraft system. It has four modes: initially the system resides in *Idle* mode. When the target is positioned somewhere, the hovercraft switches to any one of *Rotate*, *Forward* or *Point* modes. Every mode has one task *errorTask* which continuously determines the difference of the target position and angle with respect to the hovercraft's position and angle. This information,

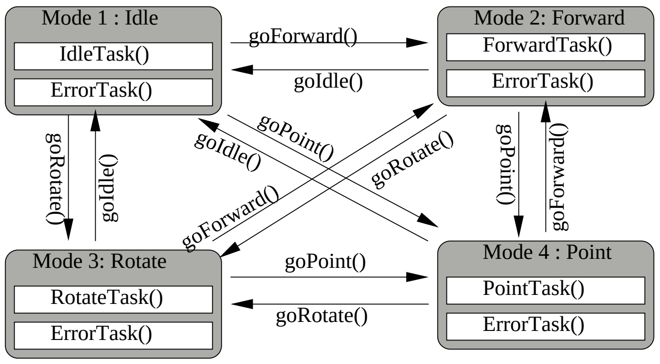


Fig. 22: Components of a hovercraft system.

stored in the *errorX*, *errorY*, *errorAngle* and *targetDirection* variables, is used by other functions.

The mode *Forward* has one more function *forwardTask*(), which uses the above information to calculate the *turn* and *thrust* values. The tasks *turnTowardsTargetTask*() and *turnToTargetTask*() in modes *Rotate* and *Point*, respectively, do the same function. These *turn* and *thrust* values are used by the actuator driver functions *ComputeLeftJetPower*() and *ComputeRightJetPower*() to compute the left and right engine's power. These power values are actually applied to the left and right engines by the *PutRightJet* and *PutLeftjet* functions, respectively.

The hovercraft has six sensors - *positionX*, *positionY*, and *angle* are for the hovercraft's position; *targetX*, *targetY* and *targetAngle* are for the target's position. Every mode has a mode period of 200 ms. All the task invocations, actuator updates and mode switches have frequency 1 in every mode.

6.2 Translating the Hovercraft system into a network of timed automata

We now describe the translation of the hovercraft control system according to the second scheme described in Section 4. This system has five tasks, so we have a total of ten timed automata for task invocations. The tasks automata for the task *Idle* are shown in Fig. 23 (task driver) and Fig. 24 (task function). The task id 1 for the task *Idle* is used by the resource management automaton. This task has a worst case execution time of 100ms and a logical execution time of 200ms, as can be seen in the timed automata for the corresponding task function.

This system has two actuators - *Left jet* and *Right jet*. The functions *ComputeLeftJetPower*() and *ComputeRightJetPower*() on the first edges of the automata are driver functions which compute the left and right jet's power. The second edges apply this power to the actuators by the actuator update functions. The actuators are updated every 200ms in every mode as can be seen in Fig. 25 and Fig. 26.

Fig. 27 depicts the timed automaton for mode switches. It has five locations – one for each mode and one *fail* location. The mode period is 200ms for each mode.

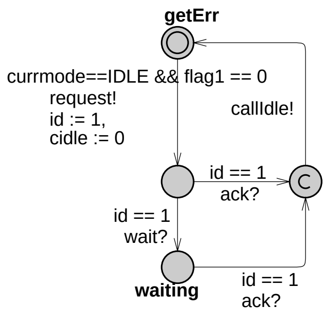


Fig. 23: Task driver for task *Idle*.

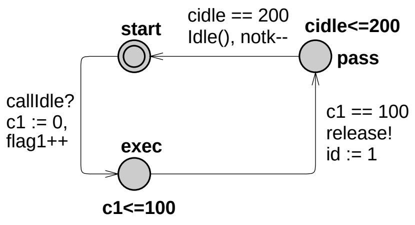


Fig. 24: Task function for task *Idle*.

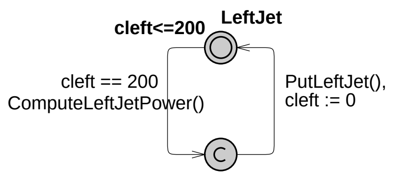


Fig. 25: Actuator *Left jet*.

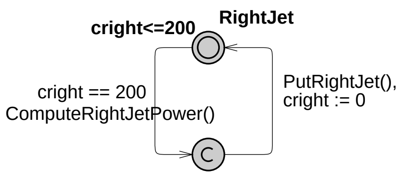


Fig. 26: Actuator *Right jet*.

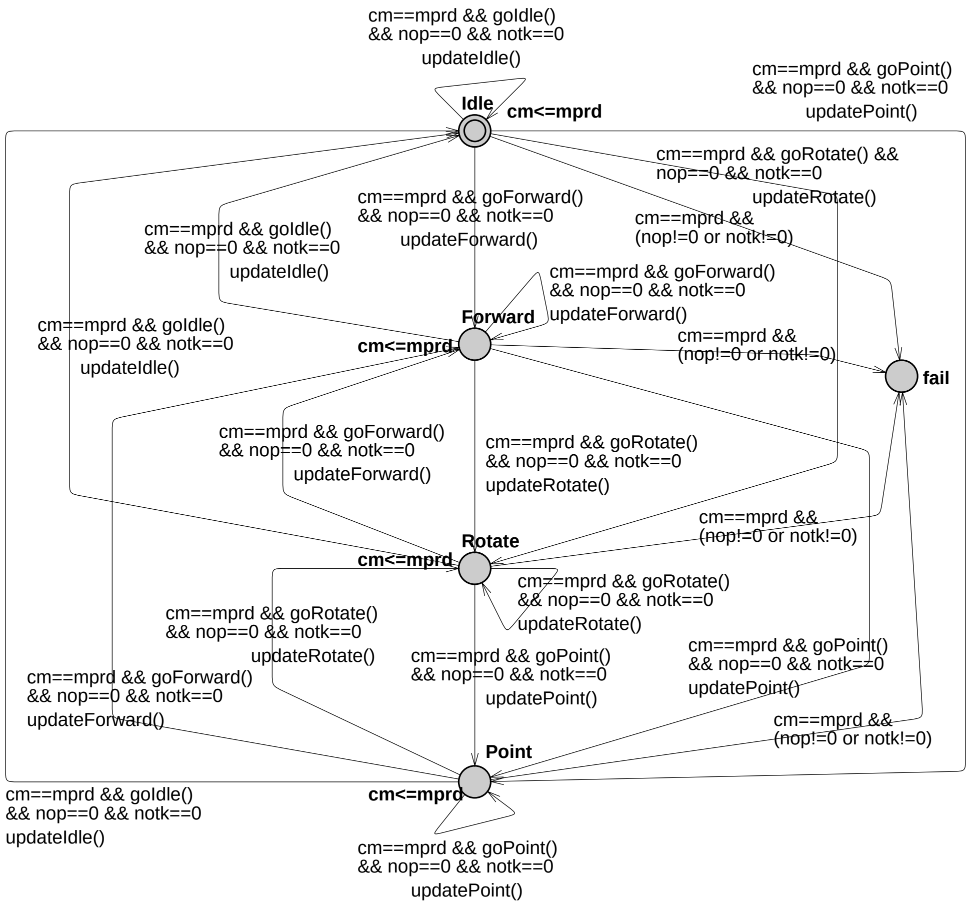

Idle
cm<=mprd
Forward
Rotate
Point
fail
cm==mprd && goIdle() && nop==0 && notk==0 updateIdle()
cm==mprd && goForward() && nop==0 && notk==0 updateForward()
cm==mprd && goRotate() && nop==0 && notk==0 updateRotate()
cm==mprd && goPoint() && nop==0 && notk==0 updatePoint()
cm==mprd && (nop!=0 or notk!=0)

Fig. 27: Timed automaton for mode switches of hovercraft system in Fig. 22.

There is one edge from every mode to the *fail* location. This edge is enabled at the end of a mode period if either of *nop* and *notk* (these variable are described in Section 4) is non-zero. The update functions (*updateIdle, updateForward etc.*) on the edges of the mode switch automaton set the corresponding flags, reset the clock and update the value of *currmode* variable.

For this system, we consider an implementation platform with two processors.

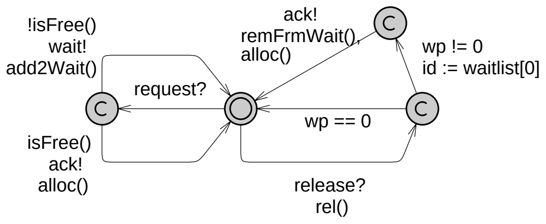


Fig. 28: Resource: processor.

The timed automaton shown in Fig. 28 manages the allocation and release of these processors. The resources are allocated on the basis of first-come-first-served and fixed-priority non-preemptive scheduling policies.

6.3 Results

To begin with, we take a platform which has one processor. We take a WCET of 100ms for each task, while the logical execution time for every task is 200ms. All tasks have equal priorities. Now, we try to verify the following counter schedulability property for the *Idle* mode:

$$A\Box(ModeSwitch.Idle \Rightarrow A\Diamond ModeSwitch.fail)$$

where, *ModeSwitch* is the timed automaton representing mode switch and *fail* is the location in that automaton as described in Section 4.

The property states that, whenever the *Idle* mode is reached, all the paths from this mode will lead to *fail* location. So, if the system is schedulable in this mode, the above property must not be satisfied. The UPPAAL verification engine reports that the property is not satisfied for the given system, *i.e.*, the given elevator system is schedulable for the given platform.

We check the counter schedulability property for each mode:

$$A\Box(ModeSwitch.Forward \Rightarrow A\Diamond ModeSwitch.fail)$$
$$A\Box(ModeSwitch.Rotate \Rightarrow A\Diamond ModeSwitch.fail)$$
$$A\Box(ModeSwitch.Point \Rightarrow A\Diamond ModeSwitch.fail)$$

None of the properties is satisfied, leading to the conclusion that the system is schedulable in all modes on the given platform.

To test the system further, we take two processors. Since this system has only two tasks in each mode, we include a dummy task in each mode to make the scheduling problem non-trivial (two tasks can be trivially scheduled on two processors). Every task can be executed on any of the processor, every task has the same WCET on both the processors. For example, we take a WCET of 100ms for every task, while the logical execution time for every task is 200ms. The counter schedulability property for every mode is not satisfied, *i.e.*, the system is schedulable since two of the tasks can be executed on one processor and the third task on the other.

Now, we change the WCET of one task (for *e.g.*, of mode *Rotate*) to 200ms and of the other to 150ms, retaining the WCET of 100ms for the third task. This time the property for the mode *Rotate* is satisfied and all others are not satisfied. Upon exploring the trace, we find the following reason for the failure - in the *Rotate* mode, the task with WCET 200ms is allocated to one processor and the task with WCET 150ms is allocated to the other. After 150ms, the second task releases the processor which is allocated to the third task with WCET 100ms. Since this task has already waited for 150ms, it can not finish its execution before its logical execution time deadline of 200ms. There are other traces witnessing the failure of the same property.

We carried out the same experiments for fixed-priority scheduling and obtained similar results. In UPPAAL, priorities can be assigned in the system definition line.

7. Conclusion

The software implementation of a control system design on a platform is always susceptible to errors due to the various layers of abstraction involved in the design process. It is possible for a system designed by a control engineer to be flawless in terms of its high level behaviour specified by the control laws, and yet exhibit incorrect or unexpected behaviour when implemented using software. Thus a verification of the implementation against the high-level control specification is necessary for the end reliability and robustness of the system.

We propose a general scheme to translate a Giotto model into a network of timed automata so that its high level functional requirements can be verified using UPPAAL. The translation scheme takes into account the syntactical structure of Giotto constructs and various timing parameters. We also extended the scheme to verify the schedulability of the system under given platform constraints using non-preemptive scheduling policies. These schemes are quite general – they can be applied to any control system that can be modelled in Giotto. Of course, the functionality of the tasks has to modelled explicitly in UPPAAL in each case.

Our work suffers from several limitations. UPPAAL's inability to support adequate libraries for implementing complex mathematical functions limits us to model relatively simple systems. We have made an assumption that the mode period of a mode must be equal to the LCM of the task periods of all tasks in that mode. Relaxation of this assumption is part of future work. Further, in illustrating the schemes we have considered the task frequency to be *one*. This is not a serious limitation, however, as the schemes can be easily extended for frequency more than one by chaining the task driver automata. We have already mentioned that we consider only non-preemptive scheduling in this paper. Verification of the schedulability of a set of tasks using preemptive scheduling schemes in the framework of timed automata has been investigated by Fersman *et al.* [2002, 2004] by using the notion of *task automata*. The same method can be adapted to our verification of schedulability of Giotto tasks.

Our future work includes the automation of the scheme presented here and extending the scheduling scheme for more resource types and complex platforms.

A platform specification consists of a number of processing elements, the worst case execution time for tasks and worst case communication times (WCCT). For simplicity, we have assumed the WCCT to be zero. The worst case execution times vary from one processor to another. In this paper we have considered all the processors of the same type, *i.e.*, a computation has the same WCET on all processors.

In summary, this paper is an attempt to bridge the gap between control law design and real-time implementation, by offering verification options at an intermediate layer (the Giotto model) that can check both platform independent and platform dependent properties.

References

Alur, R. and Dill, D. L. 1990. Automata for modeling real-time systems. In *Proc. of the 17th International Colloquium on Automata, Languages and Programming Automata, ICALP90*, Volume 443 of *Lecture Notes in Computer Science*. Springer, 322–335.

Alur, R. and Dill, D. L. 1994. A theory of timed automata. *Theoretical Computer Science 126*, 2, 183–235.

Behrmann, G., David, A., and Larsen, K. G. 2004. A Tutorial on UPPAAL. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004*, Volume 3185 of *Lecture Notes in Computer Science*. Springer, 200–236.

Berry, G. and Gonthier, G. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming 19*, 2, 87–152.

Bertin, V., Closse, E., Poize, M., Pulou, J., Sifakis, J., Venier, P., Weil, D., and Yovine, S. 2001. TAXYS=Esterel+Kronos. A tool for verifying real-time properties of embedded systems. In *Proc. of the 40th IEEE Conference on Decision and Control*, Volume 3. IEEE, 2875–2880.

Clarke, E. M., Grumberg, O., and Peled, D. 1999. *Model Checking*. MIT Press.

David, A. and Yi., W. 2000. Modelling and analysis of a commercial field bus protocol. In *Proc. of the 12th Euromicro Conference on Real Time Systems*. IEEE Computer Society Press, 165–172.

Fersman, E., Pettersson, P., and Yi, W. 2002. Timed Automata with Asynchronous Processes: Schedulability and Decidability. In *Proc. of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Volume 2280 of *Lecture Notes in Computer Science.* Springer–Verlag, 67–82.

Fersman, E. and Yi, W. 2004. A Generic Approach to Schedulability Analysis of Real-Time Tasks. *Nordic Journal of Computing 11*, 2, 129–147.

Henzinger, T. A., Horowitz, B., and Kirsch, C. M. 2001. Embedded Control Systems Development with Giotto. In *Proc. of the International Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM Press, 64–72.

Henzinger, T. A., Horowitz, B., and Kirsch, C. M. 2003. Giotto: A Time-triggered Language for Embedded Programming. *Proceedings of the IEEE 91*, 1, 84–99.

Hune, T., Larsen, K. G., and Pettersson., P. 2000. Guided synthesis of control programs using UPPAAL. In *Proc. of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation*. IEEE Computer Society Press, 15–22.

Iversen, T. K., Kristoffersen, K. J., Larsen, K. G., Laursen, M., Madsen, R. G., Mortensen, S. K., Pettersson, P., and Thomasen, C. B. 2000. Model-checking real-time control programs – Verifying LEGO mindstorms systems using UPPAAL. In *In Proc. of 12th Euromicro Conference on Real-Time Systems*. IEEE Computer Society Press, 147–155.

Lindahl, M., Pettersson, P., and Yi, W. 2001. Formal design and analysis of a Gear Controller. *International Journal on Software Tools for Technology Transfer 3*, 3, 353–368.

Marco, A. A., Sanvido, and Walburg, A. 2004. Giotto Tutorial. Tech. report, UCB/ERL M04/30, University of California at Berkeley.

Milner, R. 1989. *Communication and Concurrency*. Prentice Hall.

Sifakis, J., Tripakis, S., and Yovine, S. 2003. Building models of real-time systems from application software. *Proceedings of the IEEE 91*, 1, (Jan.), 100–111.