SCHEDULE VERIFICATION AND SYNTHESIS FOR EMBEDDED REAL-TIME COMPONENTS*

Purandar Bhaduri
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Guwahati 781039, India
pbhaduri@iitg.ernet.in

Abstract In this paper we address the problems of schedule synthesis and timing verification for component based architectures in embedded systems. We consider a component to be a set of tasks with response times that lie within specified intervals. When a set of components is deployed to implement a desired functionality, we want to guarantee that the components can achieve the timing constraints of the application. We solve the associated synthesis and verification problems using the framework of timed interface automata and timed games.

Keywords: Component-based embedded real-time systems, real-time scheduling, timed interfaces, timed games, schedule synthesis.

1. Introduction

Component based development has been proposed as a framework for dealing with the complexity of embedded control systems. It is based on the premise that generic components can be developed so as to be *reused* in different contexts. While the encapsulation of behaviour in component interfaces does lead to modularity and enhanced reuse, the verification of non-functional aspects (such as timing and resource constraints) of an assembly of components remains a major challenge.

In this paper we analyse whether a given set of components satisfies the timing constraints of an embedded control application. We consider a component to be a collection of *tasks*, which are *functionally* and *logically* related. In turn, each task has a response time (*i.e.*, the time between task release and completion) that is guaranteed to lie within a specified interval by the component implementation. When a set of components is deployed to implement a de-

*This work was supported by a grant from GM R&D.

sired functionality, we want to guarantee that the components can achieve the timing constraints of the application. The application-level timing properties we consider here are the *end-to-end* timing constraints of *transactions*. Each transaction is typically a loop consisting of reading sensors, computing control inputs and writing to actuators. The constituent tasks of a transaction may be part of different components. The specific problems we are interested in are (a) *timing verification*: to ascertain whether the given components can satisfy the end-to-end timing constraints of the application, and (b) *schedule synthesis*: if the answer to (a) is yes, to determine a sequence of task release actions that will lead to satisfaction of the constraints.

We refer to the above problem as *component scheduling*, to distinguish it from *task scheduling*, the staple of real-time scheduling theory. In task scheduling, we already know the deadlines, periods and execution times of tasks, and want to know whether the tasks can be scheduled to meet their deadlines. In component scheduling we know that the tasks can be scheduled to meet certain deadlines (which may *not* be related or derived from the application at hand), but want to know whether these tasks can be released in such a way that the end-to-end constraint of a transaction can be met. Task scheduling is a top-down analysis – from the real-time requirements we identify tasks and their characteristics, identify the platform and check whether the tasks can be scheduled. Component scheduling is bottom-up – given the components and the constituent tasks, along with their pattern of release and completion times, we want to verify whether they can satisfy the end-to-end constraints. The component scheduling problem becomes relevant when the tasks are not identified based on the real-time requirements of the particular application, but the application itself is built by composing pre-existing components.

Our approach to solving the timing verification and schedule synthesis problems for components is based on the formalism of timed interface automata (TIA) [de Alfaro et al., 2002]. We view the problems as a *timed game* between two players – one representing the environment (the scheduler or *Input*) and an adversary representing the system (the component or *Output*). The environment can decide on when to release tasks for execution, but not their completion times, which can be decided only by the component. Both players make certain assumptions about the other player, and deliver certain guarantees. The overall goal is to check that there is a sequence of allowed moves by the environment (release of tasks) which leads to satisfaction of the high-level timing requirements; in other words, there is a winning strategy for the environment in the corresponding timed game. The existence of such a winning strategy guarantees that the components can be used together to satisfy the end-to-end timing constraints.

Timed games have been used to solve several scheduling problems – see [Altisen et al., 1999, Altisen et al., 2002] for example. Unlike these works, we solve a new scheduling problem that is unrelated to traditional task or job-shop scheduling. A key feature of our work is that all the timing requirements (both task characteristics and external timing constraints) are captured using the TIA formalism, a formalism for compositional reasoning about timed systems. Our techniques are therefore *modular*, and can be applied in a *compositional* and *incremental* manner.

The main novelty of this work is that we define a notion of component scheduling and propose methods for solving the associated verification and synthesis problems. Contrary to the classical notion of task scheduling, component scheduling deals with transactions involving a set of tasks rather than separate task instances. In our setting, checking for deadline violation corresponds to checking that the end-to-end constraints of a transaction are satisfied. Component scheduling is motivated by the fact that modern embedded control systems are typically built out of existing components. Components consist of tasks representing component services; transactions are application specific jobs that span across a set of components. The main technical contribution of this work is twofold: *encoding* the specification of component scheduling problem as timed interface automata and *reduction* of the verification and synthesis problem for component scheduling to finding a winning strategy in the game structure for the associated timed interface automata. Our use of TIA for modelling both tasks and transactions is novel. So is our use of the formalism for solving scheduling problems, since we go beyond checking compatibility of timed components

As an application, we apply our component scheduling framework to the problem of deriving a static time-triggered schedule for a set of periodic tasks. We are given a set of processors and a number of tasks with known frequencies, and execution times lying in fixed intervals. Each task is statically allocated to a processor, called a *TTA node*, and must communicate with other tasks through a shared bus. The problem is to find a static schedule on each processor along with a bus schedule, such that all task and communication deadlines are met without any task being preempted when executing. The solution using our approach is worked out on an automotive Adaptive Cruise Control (ACC) application.

2. The Component Scheduling Problem

The typical design flow in component-based development of embedded systems is as follows. To implement a given *feature* of the system to be built, such as the adaptive cruise control feature in an automobile, a number of transactions, each consisting of a related set of tasks, is identified. A transaction is actually a partial order on the tasks reflecting their interdependence. The tasks comprising a transaction usually span multiple components. The end-to-end

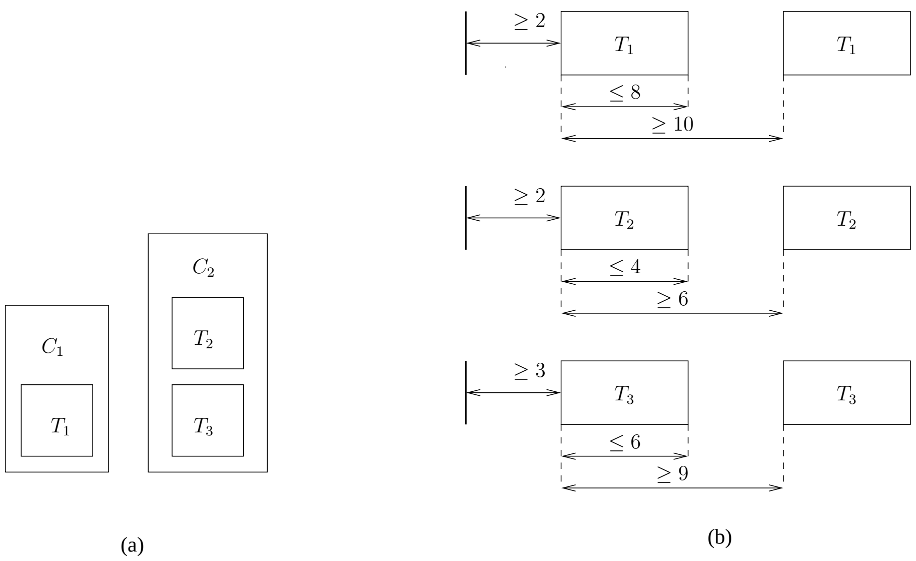


Figure 1. Components, Tasks and Timing Constraints

timing constraints for each transaction are derived from the feature requirements, and must be met by the tasks from different components that constitute the transaction. This is the essence of the component scheduling problem.

Tasks and task graphs

According to our view, a component is a set of tasks, with each task satisfying certain timing constraints. A component is a black-box which hides the internal details of how tasks are actually scheduled. The interface only exposes the timing constraints in the form of *assumptions* about task release times and *guarantees* about task completion times. In our setting the release and execution times of a task may not be strictly periodic, but can lie within a specified interval. This facilitates modelling of jitter and communication delays and leads to more flexibility in scheduling, as tasks with fixed periods are too simplistic and lead to pessimistic analysis.

EXAMPLE 1 (COMPONENTS AND TASKS) *Figure 1(a) shows two components C_1 and C_2. Tasks T_1 belongs to component C_1, while tasks T_2 and T_3 belong to C_2.*

Figure 1(b) shows various timing constraints for the tasks T_1, T_2 and T_3. For instance, task T_1 cannot be released within the first 2 *time units, which is an assumption on the environment; we call such a constraint an* offset constraint. *Once the task T_1 is released, it must complete within 8 time units, a* *guarantee provided by the component; we call such a constraint an* execution time constraint. *Further, the delay between two successive task-releases has to be at least 10 time units, again an assumption on the environment; we call such a constraint a* period constraint.

Task Graphs. We model transactions as *task graphs*, i.e., partial orders (or DAG's) on tasks. The partial ordering reflects the data dependencies between tasks in a particular transaction: an edge from task T_i to T_j indicates that task T_i must complete before task T_j begins. We associate an end-to-end deadline with each transaction, as well as constraints on inter-task separation to guarantee freshness of data. The constraints on deadline and the inter-task separation are collectively referred to as *end-to-end constraints*. Note that a transaction represented by a task graph is periodic, the period being determined by the sampling frequency of the associated control loop. We assume that the period of the transaction is given by the end-to-end deadline of the task graph.

EXAMPLE 2 (TASK GRAPH) *Figure 2 shows a task graph for a transaction involving components C_1 and C_2 in Example 1. It says task T_2 must be released after tasks T_1 and T_3 have completed. The transaction has an end-to-end deadline of* 14 *time units, and a constraint that says T_2 must be released within* 6 *time units from the completion of T_3 (to ensure* freshness *of data, for instance).*

Another constraint that is implicit in the task-graph is that the transaction it represents is required to execute an infinite number of time, a liveness *constraint.*

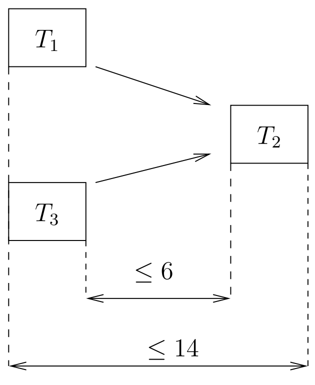


Figure 2. Task graph and transaction constraints

DEFINITION 3 *A* task graph with end-to-end constraints *is a triple $G = (\mathcal{T}, <_{\mathcal{T}}, d)$ where $\mathcal{T} = \{T_1, \ldots, T_m\}$ is a set of m tasks, $<_{\mathcal{T}}$ is a strict partial* *order (i.e., an irreflexive transitive relation) on $\mathcal{T}$ and d is a set of constraints of the form $d(\mathcal{T}) \leq C$ or $d(a, b) \leq C$ for $a, b \in \bigcup_{1 \leq i \leq m}\{r_i, c_i\}$, where C is an integer. The constraint $d(\mathcal{T}) \leq C$ represents an end-to-end deadline of C time units for the task graph, while the constraint $d(a, b) \leq C$ represents a maximum separation of C time units between the two actions a and b, which are either the release r_i of a task T_i or the completion c_j of a task T_j. We denote by $\Pi(T)$ the set of immediate predecessors of task T in the partial order $(\mathcal{T}, <_{\mathcal{T}})$.*

The Problem

The scheduling problem we are trying to solve is: given a set of tasks with timing constraints on their release and completion, and a task graph with end-to-end constraints, to find a schedule, i.e., a timed sequence of release actions (which may depend on the timed sequence of preceding completion actions), which satisfies the constraints imposed by the task graph. The latter constraints are: (1) a task can be released only if all its predecessors have completed; (2) the time duration between the earliest release and the latest completion action is bounded by the end-to-end deadline of the task graph; and (3) the time duration between each pair of actions in a specified list is bounded by the corresponding separation limit.

DEFINITION 4 *A* timed trace *on an alphabet A of actions is a sequence $\sigma = (a_0, t_0), (a_1, t_1), \ldots$, where each $a_j \in A$ and each $t_j \in \mathbb{R}^{\geq 0}$, with $t_0 <= t_1 <= t_2 \ldots$. We call t_j the* time-stamp *of the j^{th} action occurrence in the sequence.*

DEFINITION 5 *Given a set of tasks $\mathcal{T}$ with associated timing constraints on their release and completion actions, a* release-schedule *σ is a function, that given a time instant for the completion of the task instances released earlier, assigns a time instant $\sigma(r_{ij}) \in \mathbb{R}^{\geq 0}$ to the release of the j^{th} instance of task T_i for each $i \in \{1, \ldots, m\}$ and each $j \geq 0$. Such an assignment must satisfy the offset and period constraints of each task. Likewise, a* completion-schedule *τ is a a function, that given the release times of the j^{th} instance of task T_i and other tasks started earlier, assigns a time instant $\tau(c_{ij}) \in \mathbb{R}^{\geq 0}$ to the completion of the j^{th} instance of task T_i for each $i \in \{1, \ldots, m\}$ and each $j \geq 0$. Such an assignment must satisfy the execution time constraint of each task.*

Given a release-schedule σ and a completion-schedule τ, we can define the outcome $Outcome(\sigma, \tau)$ of the two schedules in the usual inductive way. This is a set of timed traces over $\bigcup_{1 \leq i \leq m}\{r_i, c_i\}$.

DEFINITION 6 *Given a set of tasks $\mathcal{T}$ with associated timing constraints on their release and completion actions, and a task graph G expressing end-to-end constraints of a transaction, a* schedule *σ is a release-schedule, such that* *for all completion-schedules τ, every timed trace $\pi \in Outcome(\sigma, \tau)$ satisfies the following conditions:*

1 Precedence*: For every pair $T_i <_{\mathcal{T}} T_j$ in G, the n^{th} occurrence of r_j is preceded by the n^{th} occurrence of c_i in π, for every n.*

2 End-to-end deadline*: For an end-to-end deadline constraint of the form $d(\mathcal{T}) \leq C$, $max(\{ts(\alpha') - ts(\alpha)\}) \leq C$, where α, α' range over all the n^{th} occurrences of actions c_j,r_k respectively in π, for all $j, k \in \{1 \ldots m\}$ and for all n. Here $ts(\alpha)$ denotes the time-stamp of action α.*

3 Separation constraints*: For every constraint of the form $d(a, b) \leq C$, $ts(\alpha') - ts(\alpha) \leq C$, where α, α' are the n^{th} occurrences of a, b respectively in π, for all n.*

4 Liveness*: There is an n^{th} occurrence of $r_i?$ for every $i \in \{1 \ldots m\}$, for every n.*

Intuitively, the above definition captures the fact that a schedule must specify a correct timed sequence of releasing tasks, no matter how much time the tasks take for completion, as long as they are within specified bounds. We now formally define the verification and synthesis problem we are interested in.

DEFINITION 7 *The* timing verification *and* schedule synthesis *problems for end-to-end constraints are defined as follows. Given a set of tasks $\mathcal{T}$ and a task graph G, verify that there exists a schedule (i.e., a way of generating release actions for tasks) that satisfies the end-to-end constraints in G,* no matter when the tasks complete, as long as they satisfy the given constraints, *and synthesise such a schedule if it exists.*

EXAMPLE 8 *Consider the set of tasks specified in Figure 1 and the task graph in Figure 2. In this example, the components C_1 and C_2 do meet the end-to-end constraints of the transaction. A possible schedule for meeting the requirements would be to release each task according to the timed trace $(r_1?, 2), (r_3?, 4), (r_2?, t)$ where t is the maximum of the completion times of T_1 and T_3, which is guaranteed to be within 10 time units. Note that releasing the task T_3 earlier than 4 time units (say at 3 time units) can lead to a violation of the freshness constraint (depending on when T_1 completes its execution, which the environment cannot control), although the interface for T_3 does not itself rule out the possibility.*

From the above example it is clear that the two timing analyses mentioned above can be carried out at the level of tasks rather than components, since they involve the timing assumptions and guarantees of only individual tasks. However, the component view would be essential when we consider the following situations:

- Tasks in a component have resource conflicts due to shared resources such as buffers.
- Components may not be “reentrant”, in which case, the execution of two tasks of the component cannot be overlapped.
- Two different transactions can share the computations of certain tasks; for example, a sensor component will typically not perform the sensing task for different transactions separately – the sensor data will be broadcast to all the components with tasks that depend on the data.

All these situations can be modelled using the TIA framework, though the resulting TIA models will be more complex in general. For instance, resource conflicts can be modelled by using an additional TIA for for modelling the resource access, and guaranteeing mutual exclusion by allowing synchronisation with the resource TIA. An example of this kind is treated in Section 5, where we apply our component scheduling framework to derive a static time-triggered schedule for a set of distributed tasks.

3. Modelling Component Scheduling with Timed Interfaces

In this section, we model the tasks, and task-graphs of the previous section using timed interface automata. Interface automata were presented in [de Alfaro and Henzinger, 2001] as a formalism for studying compatibility of components in an *open* system. Timed interface automata (TIA) [de Alfaro et al., 2002] were proposed as an extension to model real-time constraints on interacting components. Due to lack of space we cannot present all the relevant details of the TIA model here. The reader is referred to [de Alfaro et al., 2002] for the formal definitions and the important properties of the TIA model. Our use of the TIA framework is novel, and is different from the one in [de Alfaro et al., 2002]: our goal is to synthesise schedules rather than to check compatibility of components.

Timed Interface Automata for Tasks

Timed interface automata are syntactically similar to traditional timed automata as in [Alur and Dill, 1994], with the exception that location invariants are classified as either *input* or *output* invariants. The crucial difference lies in the semantics – timed interface automata correspond to *games* between players *Input* and *Output*, rather than just labelled transition systems. It is the responsibility of player *Input* to ensure that all the input invariants are met; similarly for the output invariants with respect to player *Output*.

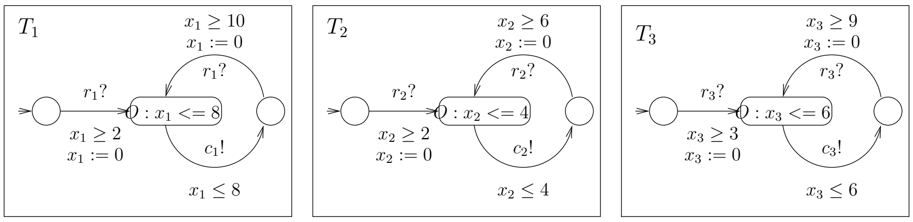


Figure 3. TIA for tasks T_1, T_2 and T_3

EXAMPLE 9 (TIA) *Figure 3 shows timed interface automata corresponding to the tasks in Example 1.*

- *The release and completion events of tasks are described using actions $r_i?$ and $c_i!$ of the task T_i.*
- *The clock variable x_i in the timed interface automaton for task T_i keeps track of the time elapsed since the last release of the task.*
- *The guards on the transitions describe when the actions $r_i?$ and $c_i!$* may *take place.*
- *The location invariants describe when certain actions* must *take place; for example the location invariant $O : x_1 < 8$ is an* output-invariant *(indicated by the label O), indicating that the output $c_1!$ must be produced while $x_1 < 8$ holds, otherwise player Output loses the game.*
- *The guards on the transitions with input action $r_i?$ specify that a minimum inter-arrival time should be maintained, otherwise player Input loses the game.*

DEFINITION 10 *Let $\mathcal{T} = \{T_1, \ldots, T_m\}$ be a set of m tasks. The* TIA for a task $T_i \in \mathcal{T}$ *(also denoted by T_i) is given by a TIA with a single clock x_i, input action $r_i?$ and output action $c_i!$. The clock constraints appearing as invariants and guards express the pattern of release and completion times of the task. We assume that each TIA T_i is well-formed, i.e., both players have a strategy to let time diverge, unless the other player is to be blamed for monopolising the game from some point on (see [de Alfaro et al., 2002]).*

From Task Graph to Specification Automaton

To solve the component scheduling problem, we use TIA in two distinct ways – first, to model the timing properties of tasks as presented above, and second to model a task graph for a transaction. We call the TIA for a task graph

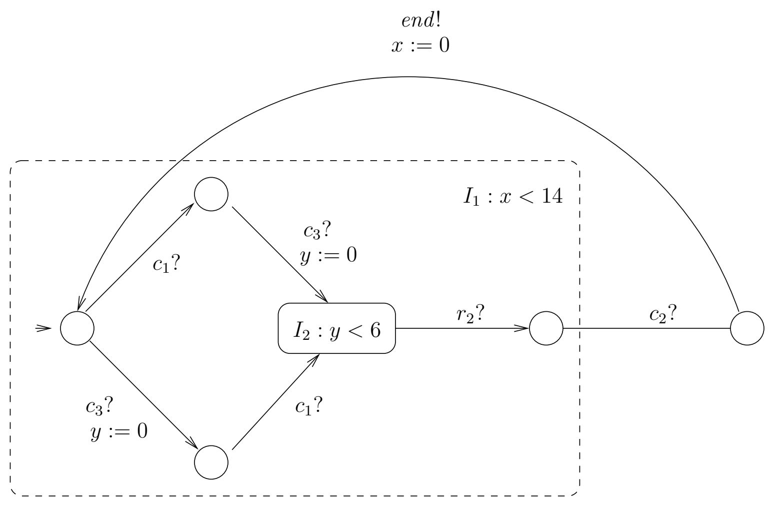


Figure 4. Specification automaton for task graph in Figure 2

a *specification automaton.* Before describing the procedure for obtaining a specification automaton from a task graph, we give an example.

EXAMPLE 11 (SPECIFICATION AUTOMATON) *The specification automaton corresponding to the task graph in Figure 2 is shown in Figure 4. It uses a clock x to record the time since the transaction was started, and a clock y to record the time since T_3 completed. The specification automaton has each r_i and c_i as* input *actions – it is an observer which detects violations of timing constraints by flagging an error state, and does not generate any output action (except the special action $end!$). It specifies all the legal runs of the environment (the scheduler) and the components that do not violate the end-to-end timing constraints.*

There is an input invariant $I_1 : x < 14$ associated with every location in the specification automaton, except the one on the extreme right (which is the final location*). This represents the fact that meeting the end-to-end timing deadline is the responsibility of player Input. For brevity, we use a statechart-like notation: an invariant associated with a super-location (the dotted oval in Figure 4) represents an invariant on all the locations contained in the super-location. Violation of the input invariant I_1 leads to a timed error state, where the progress of time is blocked. Similarly, the violation of the input invariant $I_2 : y \leq 6$ in the oval shaped location signifies violation of the freshness constraint and leads to a timed error state. The output action $end!$ is a new action not shared by any other automaton which signifies the end of the transaction.*

We following the ideas in [Abdeddaïm et al., 2003] to obtain the specification automaton from a task graph. First, we build a specification TIA P_i

for each task T_i, consisting of three locations, corresponding to the task states *waiting*, *executing* and *completed*. The transition from the *waiting* to the *executing* state is taken when the specification TIA for the tasks in $\Pi(T_i)$ are all in their final locations.

DEFINITION 12 *Let* $G = (\mathcal{T}, <_{\mathcal{T}}, d)$ *be a task graph. For every task* $T_i \in \mathcal{T}$ *its associated* specification TIA *is* $P_i = (Q_i, q_i^{init}, q_i^{final}, \mathcal{C}_i, \mathcal{A}_i^I, \mathcal{A}_i^O, Inv_i^I, Inv_i^O, \tau_i)$ *with the set of locations* $Q_i = \{p_i^0, p_i^1, p_i^2\}$*, the initial location* $q_i^{init} = p_i^0$*, the final location* $q_i^{final} = p_i^2$*, the set of input actions* $\mathcal{A}_P^I = \{r_i?, c_i?\}$*, the set of output actions* $\mathcal{A}_i^O = \emptyset$*, and the set of transitions* τ_i *include the tuples*

$$(p_i^0, \bigwedge_{T_j \in \Pi(T_i)} p_j^2, r_i?, \emptyset, p_i^1)$$

and

$$(p_i^1, true, c_i?, \emptyset, p_i^2).$$

The global specification automaton is obtained as a composition of the individual specification automata. The composition can be treated as composition of ordinary timed automata since the components have no shared actions. The composition ensures that the release actions of tasks do not violate the precedence constraints in the task graph. Next, we add some clocks and clock constraints, both as guards on transitions as well as location invariants, to take care of the end-to-end constraints in the task graph $G = (\mathcal{T}, <_{\mathcal{T}}, d)$. For the end-to-end deadline constraint $d_{\mathcal{T}} \leq C$, there is a clock t_e and an input invariant $I_e : t_e < C$ on all the locations of the composed automaton except its final location. For a separation constraint of the form $d(a, b) \leq C$, there is a clock t_{ab} which is reset on every transition with the action label a, and an input invariant $I_{ab} : t_{ab} < C$ on all locations that are sources of transitions labelled with action b. Finally, there is a transition labelled with the output action $end!$ from the final location of the composed automaton to the initial location which resets the clock t_e.

The specification automaton in Figure 4 is actually obtained by applying some optimisations on the result of the above transformation on the task graph in Figure 2: the release actions $r_1?$ and $r_2?$ do not appear in Figure 4. A general optimisation scheme based on chain coverings of a partial order is presented in [Abdeddaïm et al., 2003].

4. Timing Verification and Schedule Synthesis

In this section, we explain how the timing verification and schedule synthesis problems can be viewed as an instance of a timed game (see [Maler et al., 1995, de Alfaro et al., 2002]) between players *Input* (the environment) and

Output (the system). Further, synthesising a schedule, *i.e.*, a timed sequence of task release actions that obeys the precedence constraints in the task graph and leads to all the end-to-end constraints being satisfied, corresponds to finding a winning strategy for *Input* in such a game.

As in all timed games, there are two kinds of moves available to each player: a player can either let time progress, as long as this does not violate an invariant for the player, or make a discrete transition to a new state when the associated guard becomes enabled. Thus a move of player *Input* (a *controllable* action) either triggers a task T_i via action $r_i?$ or allows time to elapse in a location. Similarly a move of *Output* (an *uncontrollable* action) either completes execution of a task T_i via action $c_i!$ or allows time to elapse in a location.

The game structure for the schedule synthesis and verification problem *i.e.*, the graph on which the game is played (called a *timed interface* in [de Alfaro et al., 2002]), is obtained from the product of the timed interface automaton for each task and the specification automaton obtained from the task graph. The specification automaton has an *input invariant* on several locations capturing the end-to-end constraints. Violation of this invariant leads to a *timed error state*. The winning plays are those sequences of states in the game graph that avoid the error state, and in addition complete the transaction infinitely often, i.e., the goal involves both a *safety* and *liveness* condition. Finding a schedule for a given set of components that meets the end-to-end constraints of a task graph then amounts to finding a winning strategy for *Input* in the corresponding timed game.

EXAMPLE 13 (TIMED GAME STRUCTURE) *For our running example, the product of the timed interface automata for the tasks T_1, T_2 and T_3 in Figure 3 with the specification automaton in Figure 4 represents the game structure on which the timed game is played. The fact that there exists a schedule satisfying the end-to-end constraints means that player Input has a winning strategy in the game.*

In the rest of this section we elaborate on the solution to the timing verification problem in terms of winning strategies for a timed game. In the following discussion, we assume we are given a set of tasks $\mathcal{T} = \{T_1, \ldots, T_m\}$ and a task graph $G = (\mathcal{T}, <_{\mathcal{T}}, d)$ on the set $\mathcal{T}$. The global specification automaton for the task graph G, defined in Section 3, is denoted T_G.

Consider the product TIA $T = T_1 \otimes T_2 \ldots \otimes T_m \otimes T_G$, *i.e.*, the joint behaviour of all the TIA's corresponding to the tasks together with the specification automaton. Intuitively, the game structure $[\![T]\!]$ corresponding to the TIA T has the set of states $(s_1, s_2, \ldots, s_m, s)$ where each component s_i is a pair (q_i, v_i) of a location in T_i and a clock valuation over the single clock x_i, and likewise s is a pair (q, v) of a location in T_G and clock valuation over the clocks t_e and t_{ab}, where a, b range over the the actions r_i, c_j (see the paragraph in Section 3 following Definition 12). The input and output transition relations of $[\![T]\!]$ encode the possible moves of the corresponding player at a given state, and the new state that results, in the combined system of the m tasks and the specification automaton. Each transition is caused either by an immediate action (release or completion of a task T_i) or a timed action, where the player chooses to let time elapse. The available moves of a player in a state must conform to the location invariants for the player in the source and target location and the enabled transition in the source location for each component TIA T_i. An input strategy is a partial function from sequences of states to the set of the enabled moves for *Input* in the final state of the sequence. So an input strategy is a way to specify the times at which the release actions for tasks occur, given the completion times for instances of tasks released earlier, while conforming to all the constraints imposed by the TIA for each task T_i. Likewise, one can define an output strategy.

Given an input and an output strategy, one can define the resulting set of outcomes starting from the initial state s_0 of $[\![T]\!]$ (see [de Alfaro et al., 2002]). These are finite and infinite sequences of the form $\sigma = s_0, \alpha_1, \gamma_1, s_1, \alpha_2, \gamma_2, \ldots$ where α_i is the move made by player $\gamma_i \in \{I, O\}$ in state s_i. A *winning* input strategy in $[\![T]\!]$ is one for which all possible output strategies lead to outcomes which avoid reaching all timed error states. Clearly, a winning input strategy corresponds to what we call a schedule (see Definition 8), except the liveness property may not be satisfied. In particular, an outcome can be empty – if no tasks are released there are no constraints to violate (assuming there are no input invariants in the TIA for the tasks, as is the case in Figure 3).

The following procedure takes care of the liveness problem. We take the composition (see [de Alfaro et al., 2002]) of the TIA corresponding to each task and the TIA for the task graph, and then find a winning input strategy for the goal $\Box\Diamond\, t^{final}$ (which says that the final location of the task graph component in the product is reachable) in the result. Intuitively, the composition $T_1 \parallel T_2 \ldots \parallel T_m \parallel T_G$ represents all schedules that satisfy the end-to-end constraints of the task graph G, without necessarily satisfying the liveness constraint; the latter is taken care of by the goal $\Box\Diamond\, t^{final}$. Details of how such games can be solved using symbolic fixed point computations can be found in [Maler et al., 1995, de Alfaro et al., 2002]. The correctness of the procedure is captured by the following theorem.

THEOREM 14 *A schedule satisfying the end-to-end constraints in G is a winning strategy for Input in the game structure $[\![T]\!]$ for the TIA given by the product $T = T_1 \otimes T_2 \ldots \otimes T_m \otimes T_G$, with the goal*

$$[\Box\, Good([\![T_1]\!], \ldots, [\![T_m]\!], [\![T_G]\!])] \cap (t\_div \cup blame^O) \cap \Box\Diamond\, t^{final}$$

where $Good([\![T_1]\!], \ldots, [\![T_m]\!], [\![T_G]\!])$ *is the set of all states in the game structure for the product* T *that are not immediate error states,* t_div *is the set of outcomes along which time diverges,* $blame^O$ *is the set of all outcomes where player Output monopolises the game, and* t^{final} *is the set of all states whose* T_G*-component has the final location of the specification automaton* T_G*.*

Implementation. Currently, there is no implementation of timed interface automata. In order to experiment with our component scheduling framework, we hand-coded our TIA using the timed game automata (TGA) in the UPPAAL TIGA tool [UPPAAL TIGA, 2006, Cassez et al., 2005]. Unfortunately, the synchronisation behaviour of TGA in UPPAAL TIGA is quite different from that of TIA. As a result, the task graphs cannot be represented as specification automata any more. Instead the precedence constraints have to be encoded using shared boolean variables, and the end-to-end deadline has to be specified as part of the winning condition (*i.e.*, goal) for the controller. Note that this encoding in UPPAAL TIGA breaks the nice compositionality properties of the TIA framework. Also, the specification language used in UPPAAL TIGA for expressing goals is not very expressive, especially with respect to liveness constraints. The results of our experiments using the UPPAAL TIGA tool are described in the next section.

5. Application: Time-triggered Schedule Synthesis

The time-triggered architecture (or TTA, see [Kopetz and Bauer, 2003]) is a platform for distributed implementations of hard real-time systems used in automotive and avionics applications. It consists of a number of processors, called TTA nodes, that communicate by passing messages over a shared bus. The computation tasks running on the TTA nodes use the shared bus using a time-division multiple-access (TDMA) discipline based on a static schedule which recurs periodically. The problem of deriving a time-triggered schedule for a set of tasks is as follows (see [Caspi et al., 2003]). We are given a set of m periodic tasks $\{T_1, \ldots, T_m\}$ and n processors. Every task is statically allocated to a processor. The task T_i has period P_i and is allocated to processor $host_i$. Its execution time lies in the interval $[l_i, u_i]$. Tasks can model computations as well as messages. There is a special processor modelling the bus – all tasks corresponding to messages are allocated to that processor. There is a precedence relation among tasks defined by data-flow constraints. This relation includes a computation task and a message task when the former is the sender of the message. Likewise, a message task precedes the computation task that is the receiver of the message. Tasks cannot be preempted once they start running. Tasks also have relative deadlines among them to model end-to-end constraints. These are of the form $\theta_i - \theta_j \leq C$, where $\theta_i \in \{s_i, e_i\}$, where s_i

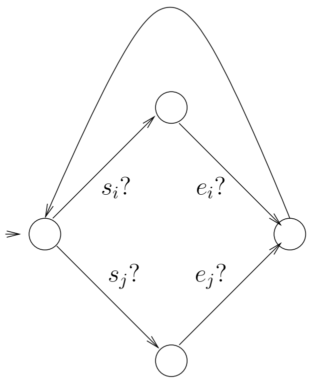


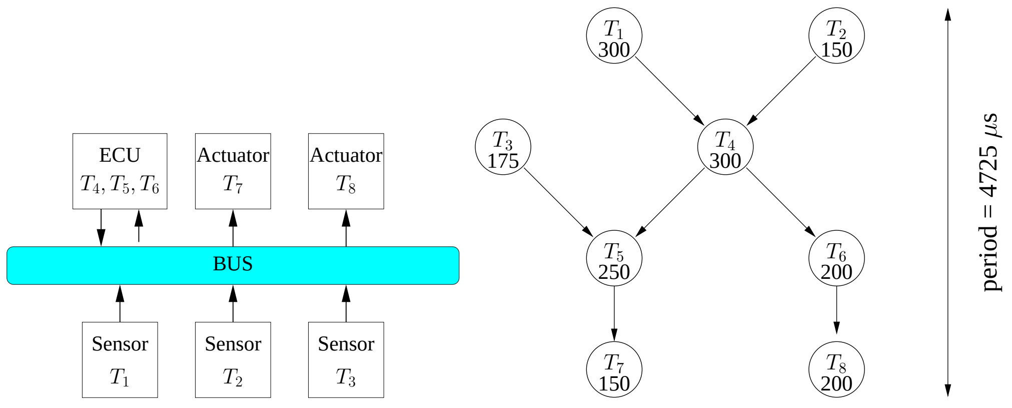
Figure 5. Synchronisation automaton for enforcing non-preemptive serial execution of tasks T_i and T_j

is the start time and e_i is the completion time of task P_i. The problem is to find a static schedule for the bus for transmission of messages, and a schedule for each TTA node for the tasks that are allocated to that node, so that all timing constraints are satisfied.

In order to apply our framework to this problem, we start with a TIA for each processor (TTA node or bus). Since each task is periodic, and has a best-case computation time l_i and a worst-case execution time u_i, we can model it using TIA, as in Section 3. However, now we have the complication that several tasks can be allocated to a single processor, and tasks cannot be preempted. This constraint can be captured using a simple device: just take the composition of the tasks allocated to a single process with a synchronising automaton which enforces the execution of only one task at a time. For every two tasks T_i and T_j allocated to the same processor, such a synchronising automaton is shown in Figure 5. Intuitively, the automaton serialises the execution of T_i and T_j. This example illustrates the case where a component (see the description in Section 2) corresponds to a set of tasks with resource constraints among them. The resource constraint here is the non-preemptive nature of task execution, and a component describes the set of tasks allocated to a processor.

Note that the constraint that each task can run only in its allocated slot is taken care of by the strict periodicity constraint. If the tasks do not have the same period, we can take the lcm of the periods to be the working period, and create multiple instances of each task to fit the period. New precedence constraints must be added between these new instances to indicate their order.

The end-to-end constraints can be modelled as TIA as in Section 3. The composition of all the TIA involved, if defined, gives us a feasible schedule for the execution of the tasks. However, the schedule is not static, since it is an *Input* strategy in which input moves can depend on previous output moves.



(a) Architecture (b) Task graph

Figure 6. Adaptive Cruise Control

To extract a static schedule, we can take the specification of execution times as worst case execution times of tasks (worst case communication times for messages) instead of intervals. This restricts the choices for player *Output* – tasks can complete only after a fixed known duration after they start.

Discussion. Various approaches to the problem of synthesising a static time-triggered schedule based on constraint solving, branch-and-bound techniques and mixed integer linear programming (MILP) have been proposed in the literature – see [Schild and Würtz, 2000, Caspi et al., 2003, Zheng et al., 2005] for example. Because of disjunctions in mutual exclusion constraints, when posed as an optimisation problem, the feasible region is not convex (see [Caspi et al., 2003]). The typical workaround is either to use backtracking techniques based on branch-and-bound search (as in [Caspi et al., 2003]), or code the problem using binary decision variables and use a MILP solver (as in [Zheng et al., 2005]). The latter technique involves guessing a large constant M, which should be as small as possible for feasibility reasons.

It is not clear whether our approach is more scalable than the above approaches. For a definitive answer, we need an implementation of TIA that we can use to carry out experiments on real-life time-triggered systems. Our expectation is that using on-the-fly techniques of [Cassez et al., 2005] we can effectively conquer the inherent EXPTIME-complexity of the timed control synthesis problem for reachability and safety objectives.

EXAMPLE 15 (ADAPTIVE CRUISE CONTROL) *This example is adapted from [Kandasamy et al., 2003] and [Zheng et al., 2005], except we require*

Source	**Target**	**Delay** (μs)
T_1	T_4	350
T_2	T_4	650
T_3	T_5	1425
T_4	T_5	500
T_4	T_6	500
T_5	T_7	500
T_6	T_8	500

Figure 7. WCCT for messages in ACC

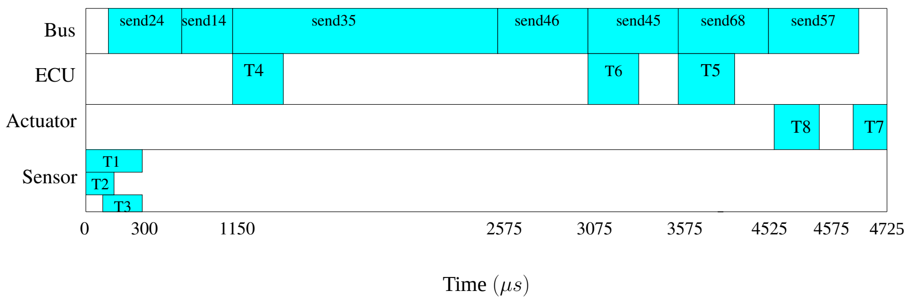


Figure 8. Time-triggered schedule for ACC example

task scheduling on an ECU to be non-preemptive. The adaptive cruise-control *(ACC) feature in an automobile automatically adapts the speed of the vehicle to the speed and distance of the vehicle in front. The ACC application involves the timely interaction among a number of tasks that are distributed, and must interact by sending messages. These tasks can be grouped as follows:*

- Sensors*:*

  T_1: *Object distance and speed*

  T_2: *Vehicle speed*

  T_3: *Throttle position*

- Controllers*:*

  T_4: *Desired speed*

  T_5: *Desired throttle position*

  T_6: *Desired brake position*

- Actuators*:*

  T_7*: Throttle actuator*

  T_8*: Brake actuator*

Figure 6(a) shows the physical architecture of the system – all sensors and actuators are directly connected to the bus, and one ECU (electronic control unit) hosts all the controller tasks. Figure 6(b) shows the task graph, with the WCET (worst case execution time) of each task appearing below the task name. The end-to-end deadline of the entire transaction is the same as the period, i.e., 4725 μ*s.*

Figure 7 shows the WCCT (worst case communication time) of the messages. The time-triggered schedule synthesised by our method is shown in Figure 8.

6. Conclusion

Component based development poses new problems for embedded control systems software. Traditional real-time scheduling theory has been successful in investigating whether a set of tasks can be scheduled on a given platform using the characteristics of the tasks and the platform. The underlying assumption is that the task characteristics have been derived from the application requirements. Since today's embedded systems are not monolithic, but are built using pre-designed components which are composed to realise a given functionality, what is needed is a new approach that combines task scheduling within a component with what we call component scheduling. This paper is an attempt to define and solve the component scheduling problem.

As future work, we would like to have an implementation of timed interface automata in order to carry out experiments to demonstrate the scalability of our approach. Experiments on small examples based on hand-coding of TIA using UPPAAL TIGA have been encouraging.

Acknowledgments

The author wishes to thank Prahlad Sampath and S. Ramesh for their detailed discussions and feedback on this work. Thanks are due to Sri Satya Aravind Akella for carrying out some of the experiments using UPPAAL TIGA described in Section 5 .

References

[Abdeddaïm et al., 2003] Abdeddaïm, Yasmina, Kerbaa, Abdelkarim, and Maler, Oded (2003). Task graph scheduling using timed automata. In *Parallel and Distributed Processing Symposium 2003*. IEEE Computer Society.

[Altisen et al., 1999] Altisen, Karine, Gößler, Gregor, Pnueli, Amir, Sifakis, Joseph, Tripakis, Stavros, and Yovine, Sergio (1999). A framework for scheduler synthesis. In *IEEE Real-Time Systems Symposium*,, pages 154–163.

[Altisen et al., 2002] Altisen, Karine, Gößler, Gregor, and Sifakis, Joseph (2002). Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems*, 23(1-2):55–84.

[Alur and Dill, 1994] Alur, Rajeev and Dill, David L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.

[Caspi et al., 2003] Caspi, Paul, Curic, Adrian, Maignan, Aude, Sofronis, Christos, Tripakis, Stavros, and Niebert, Peter (2003). From simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. *ACM SIGPLAN Notices*, 38(7):153–162.

[Cassez et al., 2005] Cassez, Franck, David, Alexandre, Fleury, Emmanuel, Larsen, Kim Guldstrand, and Lime, Didier (2005). Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR 2005 - Concurrency Theory, 16th International Conference*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer.

[de Alfaro and Henzinger, 2001] de Alfaro, L. and Henzinger, T.A. (2001). Interface automata. In *Foundations of Software Engineering*, pages 109–120. ACM Press.

[de Alfaro et al., 2002] de Alfaro, Luca, Henzinger, Thomas A., and Stoelinga, Mariëlle (2002). Timed interfaces. In *Embedded Software, Second International Conference, EMSOFT 2002*, volume 2491 of *Lecture Notes in Computer Science*, pages 108–122. Springer.

[Kandasamy et al., 2003] Kandasamy, Nagarajan, Hayes, John P., and Murray, Brian T. (2003). Dependable communication synthesis for distributed embedded systems. In *SAFECOMP 2003 Proceedings*, volume 2788 of *Lecture Notes in Computer Science*, pages 275–288. Springer.

[Kopetz and Bauer, 2003] Kopetz, Hermann and Bauer, Günther (2003). The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126.

[Maler et al., 1995] Maler, Oded, Pnueli, Amir, and Sifakis, Joseph (1995). On the synthesis of discrete controllers for timed systems. In *Theoretical Aspects of Computer Science*, volume 900 of *LNCS*, pages 229–242. Springer-Verlag.

[Schild and Würtz, 2000] Schild, Klaus and Würtz, Jörg (2000). Scheduling of time-triggered real-time systems. *Constraints*, 5(4):335–357.

[UPPAAL TIGA, 2006] UPPAAL TIGA (2006). UPPAAL TIGA home page. `http://www.cs.auc.dk/˜adavid/tiga/`.

[Zheng et al., 2005] Zheng, Wei, Chong, Jike, Pinello, Claudio, Kanajan, Sri, and Sangiovanni-Vincentelli, Alberto L. (2005). Extensible and scalable time triggered scheduling. In *Application of Concurrency to System Design (ACSD 2005)*, pages 132–141. IEEE Computer Society.