Verification and Controller Synthesis for Resource-Constrained Real-Time Systems: Case Study of an Autonomous Truck

Shuhao Li
Center for Embedded Software Systems
Aalborg University
Selma Lagerlöfs Vej 300, 9220 Aalborg, Denmark
li@cs.aau.dk

Paul Pettersson
Mälardalen Real-Time Research Center
Mälardalen University
P.O. Box 883, 72123 Västerås, Sweden
paul.pettersson@mdh.se

Abstract

An embedded system is often subject to timing constraints, resource constraints, and it should operate properly no matter how its environment behaves. This paper proposes to use timed game automata to characterize the timed behaviors and the environment uncertainties, and use piecewise constant integer functions to approximate the continuous resources in real-time embedded systems. Based on these formal models and techniques, we employ the real-time model checker UPPAAL *to verify a system against a given functional and/or timing requirement. In addition, we employ the timed game solver* UPPAAL-TIGA *to check whether a given control objective can be enforced, and if so, we synthesize a controller for the system. We carry out a case study of this approach on a battery-powered autonomous truck. Experimental results indicate that the method is effective and computationally feasible.*

1 Introduction

When designing and developing a real-time embedded application, in addition to implementing the required functionalities, designers also need to ensure that the system:

- satisfies the specified timing constraints;
- satisfies the given resource constraints in terms of e.g., available CPUs, memory, buses, communication ports, network bandwidths, or energy; and
- operates properly no matter how its uncontrollable (or "hostile") environment behaves.

For such kind of resource-constrained real-time embedded systems, we may use timed automata (TA) [2] as the analysis framework. To assure the system correctness, we need to verify the systems against the user-specified requirements. It would be even better if we can achieve software synthesis [4], or synthesize a controller [19] that enforces the given requiremetns.

There are techniques [3, 7, 10, 13, 11] and tools [8] for resource-constrained analysis of real-time systems based on the formalisms of e.g., *weighted* (*priced*) *timed automata* [3, 7] and *multi-priced timed automata* [17]. However, they do not address the controller synthesis problem. On the other hand, there are techniques [18, 16] and tool UPPAAL-TIGA [5] for timed game solving and controller synthesis based on the formalism of *timed game automata* (TGA) [18]. However, in those work there is no notion of resource usage (e.g., by means of a `cost` variable). Although resource-constrained controller synthesis for real-time systems based on *weighted* (*priced*) *timed game automata* [1, 12] have been attempted in the past few years [22, 1, 12, 15, 9, 14], the problems have been limited to reachability games, or systems that have only one clock [14], or resources that change (piecewise) linearly.

In this paper, we aim to conduct *tool-supported, automatic* verification and controller synthesis for a class of *resource-constrained* real-time systems, where the continuous resources such as memory, network bandwidth and energy can be consumed at varying rates. Rather than trying to achieve strict cost-optimality in controller synthesis, we would like to know whether it is possible to synthesize practical controllers for the given reachability and safety control objectives under the given resource constraints?

In order to model continuous resources with timed game automata which have no continuous `cost` variables, we propose to use *step functions* (a.k.a. piecewise constant integer functions) to over- or under-approximate the evolving of these resources. We model how these resources are consumed and refilled over time, and how they constrain the behaviors of the systems. We use CTL logic to specify the properties and/or control objectives. Then we employ UPPAAL [6] to verify the system against the properties, and employ UPPAAL-TIGA to synthesize controllers for the sys-

tem. We apply this approach on a case study of a battery-powered autonomous truck. Experimental results indicate that this approach is effective and computationally feasible.

1.1 Related work

For resource-constrained real-time systems, techniques such as cost-optimal reachability analysis [3, 7], WCTL (weighted CTL) model checking [13] and optimal (infinite) scheduling [10, 11] have been proposed. At a higher description level, the REMES approach [20] can model different types of resources in component-based real-time embedded systems, and enable rigorous feasibility, trade-off, and optimal resource utilization analysis by encoding these problems into the priced timed automata and weighted CTL frameworks. Some of these research results have been implemented in tools such as UPPAAL-CORA [8].

For real-time applications that are modeled as controllable systems and their uncontrollable environments, techniques and methods for game solving and controller synthesis based on timed game automata [18] have been proposed along the years. More recently, an efficient on-the-fly algorithm for controller synthesis has been proposed [16] and implemented in tool UPPAAL-TIGA [5]. If the game is checked to be solvable, then UPPAAL-TIGA can generate a state-based winning strategy as the synthesized controller, which ensures that the specified control objective is enforced on the system, no matter how the (hostile) environment behaves.

To conduct *resource-constrained* game solving and controller synthesis for real-time systems, the formalism of *weighted* (*priced*) *timed game automata* [1, 12] has been defined. For reachability games, the problem of synthesizing cost-optimal winning strategies for weighted TGA is decidable under the *acyclicity* [22] or the *strong cost non-zenoness* [1, 12] assumptions. Further results show that in the general case, cost-optimal strategy synthesis for weighted TGA with three or more clocks are undecidable [15, 9], whereas for weighted TGA with only one clock it is decidable [14]. However for safety games, in general it is conjectured to be undecidable.

Among the above-mentioned approaches to cost-optimal controller synthesis, [22, 1, 15, 9] address the problems mainly from a theoretical viewpoint. While [12] can synthesize strategies by reducing the problem to games on linear hybrid automata, the user is supposed to manually implement the solution (i.e., to encode the game solving and strategy synthesis problems) in the tool HyTech. While the suggested algorithm in [14] has a prototype implementation, it applies to priced TGA with only one clock. Furthermore, for solving practical resource-constrained control problems, there are two common limitations with these approaches: (1) they consider only reachability games; (2) for those methods that are based on linear weighted (priced) timed automata, the derivative of a cost variable in locations is a constant, which sometimes cannot faithfully characterize the dynamics of resource usages.

1.2 Contributions

The main contributions of this paper include:

- We conduct modeling and verification of a class of resource-constrained real-time systems, where the resource may evolve (i.e., consume and refill) at non-constant rates, and its status may affect the system behaviors and timings. We also show how to synthesize practical controllers that enforce the user-specified control objectives;
- We use piecewise constant functions and integer arithmetics to approximate a class of continuous resources (e.g., linear, affine and piecewise affine) within the timed automata framework, thus enabling the automatic verification and controller synthesis for systems with such resources; and
- We apply the suggested methods and techniques on a case study of an autonomous truck system, carry out quantitative evaluations of model checking and controller synthesis, and report the experimental results.

2 Problem description

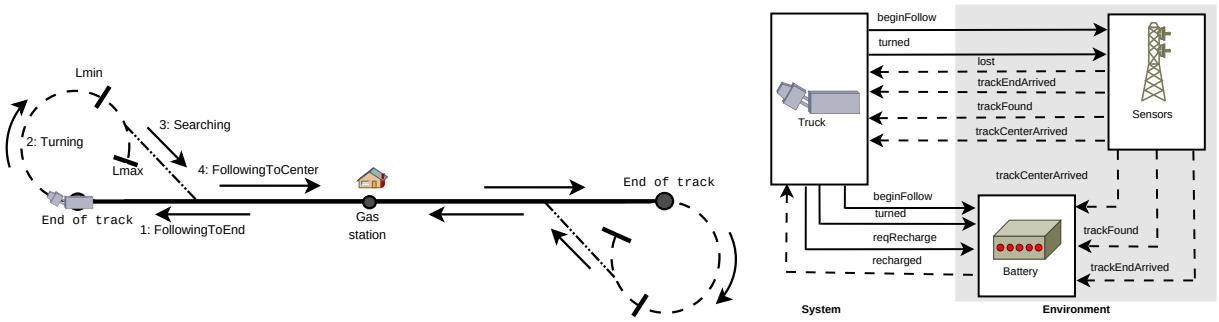
We consider an autonomous transportation system (Fig. 1(a)), which consists of a battery-powered electric truck, a straight line track (Fig. 1(a), thick solid line) and some sensors. The system operates as follows:

1. The truck moves along the track (`FollowingToEnd` phase), until it arrives at an end of the track;
2. When the truck reaches the track end (the truck will be signalled of this by a sensor), it will begin turning right (the `Turning` phase);
3. After being turning for a certain period of time, the truck itself determines to stop turning, and then to begin searching for the track (the `Searching` phase);
4. When the truck finds the track (the truck will again be signalled of this by a sensor), it aligns itself with the track, and moves along the track towards the other end (the `FollowingToCenter` phase).

Timing constraints

Assume that the truck starts from the middle point of the track. The truck needs a period of time to follow the track until it reaches the end. This duration depends on the speed of the truck.

We assume that the turning angle of the truck is a fixed value. There are two locations L_{max} and L_{min} on the turning trajectory. They correspond to the latest and earliest



(a) Movement of the truck.

(b) System structure overview.

Figure 1. An autonomous truck system.

moments that the truck can stop turning and begin searching. Depending on the turning speed, the truck needs some time to start from the track end and arrive in this "safe zone" (i.e., between L_{min} and L_{max}). If it stops turning slightly too early or slightly too late (there is a tolerance value), it *may* risk getting lost. If it stops turning much too early or much too late (another tolerance value), it will *definitely* get lost. As long as the truck is progressing within this "safe zone", it has the freedom to choose when to stop turning and begin searching for the track.

The `Searching` and `FollowingToCenter` phases have similar timing constraints as `FollowingToEnd`.

Resource constraints

We assume that the electric truck is battery-powered. The sole resource in this case study is the battery power.

The truck can progress as long as it is not running out of battery. When it is progressing, the battery power will be consumed at certain rates, which may differ from phase to phase, and from round to round of runs.

We assume that there is a "gas station" in the middle point of the track (Fig. 1(a)), where the truck can get its battery recharged if it is currently not at full capacity.

Once the truck runs out of battery, it immediately "dies" (i.e., gets stuck) there. The only exception is that if the battery power drops to zero when the truck arrives at the gas station, then it can be "rescued" there (i.e., getting recharged before dying).

Resource-constrained system behaviors and timings

We assume that the power status of the truck determines its speed. When the truck battery is high, it runs faster; when it gets low, it runs slower.

Given a certain distance, if the truck runs faster, then it needs less time to cover this distance. In the timed automata framework, this kind of timing can be modeled in terms of the upper bounds *UppBnd* of the location invariants "$clock \leq UppBnd$" of the relevant timed automata. The upper bounds will be updated regularly.

Furthermore, the battery needs some time to be recharged to its full capacity. The (expected) duration depends on how much power are left there right before the recharging.

Environment uncertainties

We assume that the truck autonomously decides when to stop turning and begin searching for the track, and also decides whether or not to get recharged when it arrives at the gas station. Other actions are mastered by the environment (e.g., the sensors) and are thus not controllable by the truck. More precisely, the environment uncertainties include:

- *non-deterministic choice of transitions*. The environment can autonomously choose one from several enabled uncontrollable transitions to be triggered. For example, if the truck has been turning for slightly too short or slightly too long a period of time (i.e., slightly outside the "safe zone") when it decides to stop turning, then it may get lost, or may not; and
- *timing uncertainty* of uncontrollable transitions. For example in the `FollowingToEnd` phase (Fig. 1(a)), the truck could possibly be signalled of reaching the end of the track a little earlier than the scheduled arrival time (the *FollowingToEnd_Delay*) due to e.g. the mild wind resistance, which constitutes an implicit part of the environment. Similar timing uncertainties apply to the `Searching` and `FollowingToCenter` phases, and the battery recharging process.

Considering that we will use the timed automata framework where the synchrony assumption is adopted by the channel synchronizations, it makes sense to allow timing uncertainty of the uncontrollable actions of the environment. For example, the amount of "tolerance" can be used to model the message transmission delays under the *asynchronous* communication mode.

Analysis and synthesis problems

For this case study, we would like to know whether some particular properties are satisfied by the system models, e.g.,

- "Will the system ever get deadlocked?"
- "Is it at all possible for the truck to "die" in the journey or get lost?"

We would also like to know whether the truck can be guided by a controller to make the turnings and rechargings at appropriate moments in time such that a given control objective is enforced on the system, e.g.,

- "Can we synthesize a controller for the truck to ensure that the truck will never get lost, never run out of battery, and in the first 327 time units it makes 8 rounds with no more than 3 rechargings and no more than 78 units power consumption, no matter how the uncontrollable (or "hostile") environment behaves?"

3 System modeling

Fig. 1(b) outlines the system architecture, i.e., how many components are there in the system, and how they communicate. The UPPAAL timed automata framework allows handshake/broadcast channel synchronizations as well as shared variable communications. For simplicity, in Fig. 1(b) we do not show how these processes read and write the shared data variables.

In Fig. 1(b) the Truck is the system in question, and the other processes (i.e., Sensors and Battery) constitute the environment (in shaded area). Environment-emitted events are uncontrollable, and are thus drawn in dashed lines.

3.1 Behavioral models

Fig. 2 and 3 present the timed game automata (TGA) models for the components of the truck system, where dashed lines represent uncontrollable (message or silent) transitions that are mastered only by the environment.

In the TGA of the Truck (Fig. 2), when the Truck is in location `Idle`, it can choose either to follow the track, or to get the battery recharged (provided that the battery is not currently at full capacity). When the Truck is in location `Turning`, it can choose to stop turning and begin to search for the track by issuing turned! at an appropriate moment in time. In locations `FollowingToEnd`, `Searching`, `FollowingToCenter` and `Recharging`, the Truck can just passively wait to be signalled of the completions of the relevant phases. Note that in urgent location (i.e., a special location that should be exited in zero time delay after it is entered) `L2`, the Truck can choose to progress on to location `Searching`. But if at this moment the environment issues a lost!, then the Truck-controllable transition will be preempted (this is because that UPPAAL-TIGA operates on competing timed games, and it assumes higher priority for uncontrollable edges).

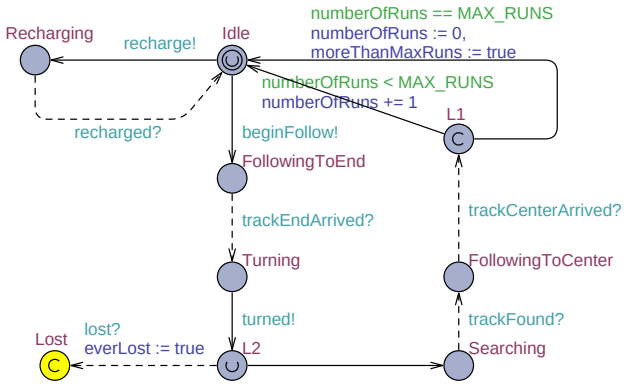


Figure 2. Timed game automaton of the Truck.

In the TGA of the Sensors (Fig. 3(a)), there could be non-deterministic choices of transitions in locations `L2` and `L3`. It means that when the Truck has been turning for a slightly longer or slightly shorter period of time, then the Truck may or may not get lost. In locations `FollowingToEnd`, `Searching` and `FollowingToCenter`, there are timing uncertainty of uncontrollable transitions. It means that the environment may be less hostile such that the Truck can complete these phases a little earlier than it normally does.

In the TGA of the Battery (Fig. 3(b)), there is timing uncertainty in location `Recharging`, meaning that the Battery may finish with the recharging a little earlier than it normally does.

The detailed models can be found in `http://www.cs.aau.dk/~li/papers/TruckCaseStudy.pdf`.

3.2 Modeling continuous resources using step functions

Popular real-time model checkers (such as KRONOS and UPPAAL) and existing timed game solver (like UPPAAL-TIGA) support no more than boolean and bounded integer data types. This section shows how to model a class of continuous resources using step functions (i.e., piecewise constant functions) and integer arithmetics.

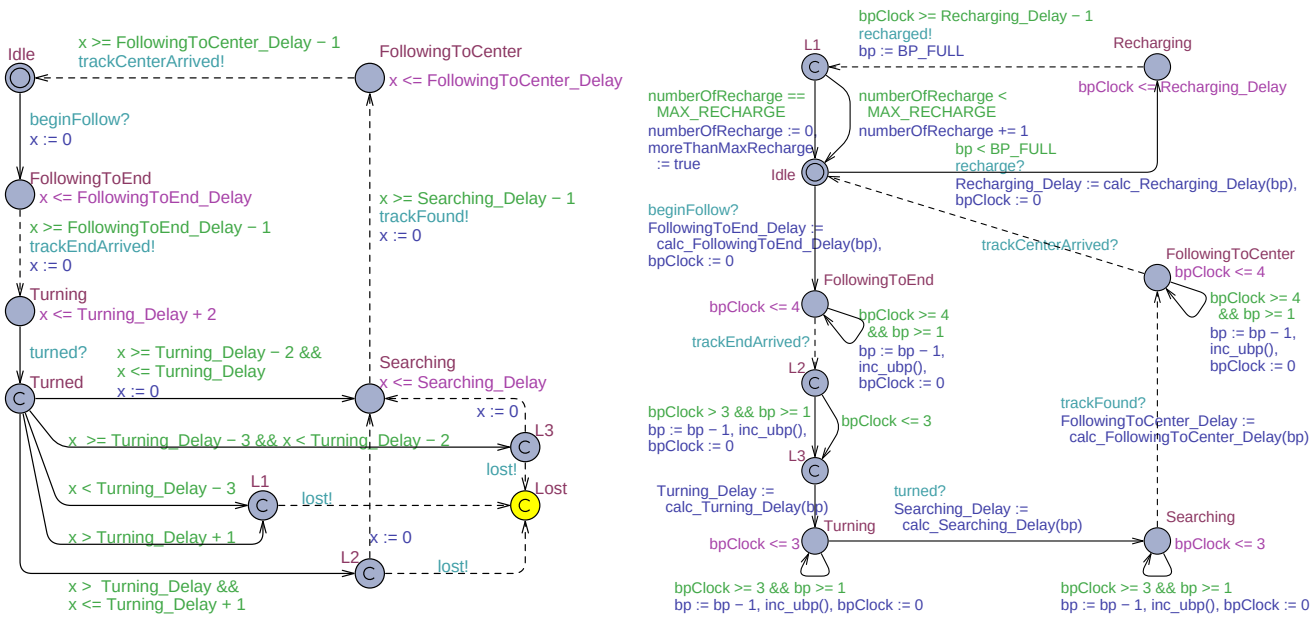
This case study will consider only a single continuous resource, i.e., the battery power (i.e., energy) of the truck.

Let the full **cap**acity of the truck battery be a constant $\mathsf{CAP} \in \mathbb{R}_{>0}$. Let p and v be the current battery power and velocity of the truck, respectively. We assume that v is determined by p as follows:

$$v = k_1/(k_2 + (\mathsf{CAP} - p)), \tag{1}$$

where $k_1, k_2 \in \mathbb{R}_{>0}$ are constants, and $0 < p \leq \mathsf{CAP}$.

Let s be a distance, and t the time (duration) that is needed to cover this distance. According to the formula



(a) TGA of the Sensors.

(b) TGA of the Battery.

Figure 3. Timed game automata of the environment components.

$s = v \cdot t$ and by Eq. (1),

$$t = \frac{s}{v} = \frac{s(k_2 + (\mathsf{CAP} - p))}{k_1} = \frac{s \cdot k_2}{k_1} + \frac{\mathsf{CAP} - p}{\frac{k_1}{s}}, \quad (2)$$

where $s \in \mathbb{R}_{>0}$, $k_1, k_2 \in \mathbb{R}_{>0}$, and $0 < p \leq \mathsf{CAP}$.

By Eq. (2), in the best case (i.e., when the truck battery is at full capacity), the truck needs $(s \cdot k_2/k_1)$ time units to cover the distance. When p decreases by k_1/s, then t increases by 1. We call $\mathsf{BCD} = (s \cdot k_2/k_1)$ the ***best-case duration***, and $\mathsf{STEP} = k_1/s$ the *step length*. Therefore, $t = f(p) = \mathsf{BCD} + (\mathsf{CAP} - p)/\mathsf{STEP}$, where between $(\mathsf{CAP} - p)$ and STEP it is a normal (i.e., a mathematical or a floating point) division. For example, let CAP be 100, BCD be 15, and STEP be 10. Then function $f(p)$ is plotted in Fig. 4(a), the thick solid slanted line.

If we define a **s**tep integer **f**unction $t = sf(p) = \mathsf{BCD} + (\mathsf{CAP} - p)/\mathsf{STEP}$, where between $(\mathsf{CAP} - p)$ and STEP it is an integer division, then the function $sf(p)$ is plotted in Fig. 4(a), the thick solid horizontal line segments which each has a hollow and a solid end points.

To model environment uncertainties, we assume that the truck may take a slightly shorter or slightly longer period of time to cover the distance. Let this tolerance be Δ, which could be e.g. $1, 2, 3, \ldots$ time units. In the ideal case, the possible durations for completing the distance are shown in Fig. 4(a), the shaded area. Accordingly, the possible durations with the step function $sf(p)$ are shown in Fig. 4(a), the areas inside the three dash-dot line rectangles which each span across a thick solid horizontal line segment.

We can calculate how well the step function approximates the continuous function with respect to the given tolerance Δ. For over-approximation, we calculate the **d**egree **o**f **a**pproximation doa_O as the ratio of the area of the continuous function to that of the step function. For example, in Fig. 4(a),

$$doa_O = \frac{(2 \cdot \Delta) \cdot \mathsf{STEP}}{(2 \cdot (\Delta + 1)) \cdot \mathsf{STEP}} = 1 - \frac{1}{\Delta + 1}, \quad (3)$$

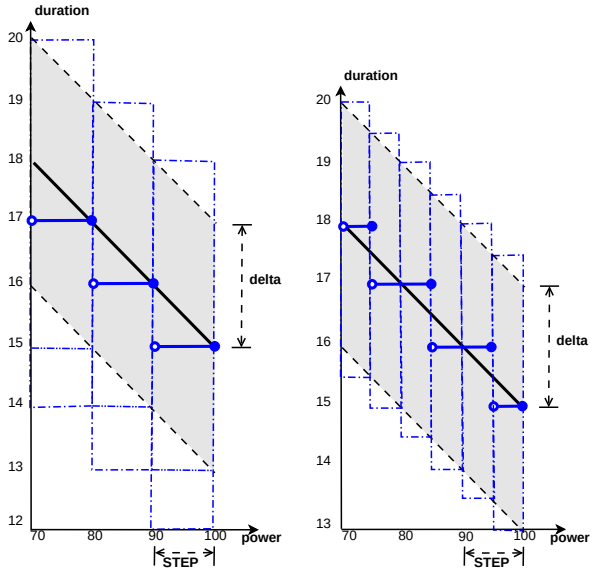
where $\Delta > 0$.

In the calculation of Eq. (3), we are assuming symmetric (equal-length) tolerances above and below the horizontal line segments. By using asymmetric tolerance intervals, we can achieve better degrees of approximation. For example, in each rectangle of Fig. 4(a), we may remove the square below the dash-dot-dot lines. By doing so, the degree of approximation in Eq. (3) is improved to

$$doa'_O = \frac{(2 \cdot \Delta) \cdot \mathsf{STEP}}{(2 \cdot (\Delta + 1) - 1) \cdot \mathsf{STEP}} = 1 - \frac{1}{2 \cdot \Delta + 1}, \quad (4)$$

where $\Delta > 0$.

We can achieve **t**ighter approximations by considering half-step length (note that the step length is uniquely determined by constant k_1 and distance s, see Eq. (2)). In this case, the integer arithmetic in Listing 1 improves the step function $sf(p)$ to $sf_T(p)$, which is plotted in Fig. 4(b), the thick solid horizontal line segments which each has a hollow and a solid end points.



(a) over-approximation (b) tight over-approximation

Figure 4. Normal and tight over-approximations using step functions.

Listing 1. The step function $sf_T()$ for tight approximations in UPPAAL.

```
const int CAP := 100; // Full capacity of truck battery
const int BCD :=  15; // Best case duration
const int STEP := 10; // batteryPower drops by STEP

int sf_T (int batteryPower) {
  if ( ( CAP − batteryPower ) % STEP >= STEP / 2 )
    return BCD + ( CAP − batteryPower + STEP / 2 ) / STEP;
  else
    return BCD + ( CAP − batteryPower ) / STEP;
}
```

When tightly over-approximating $f()$ with asymmetric tolerances, we let the upper and lower tolerances be *DeltaUpper* and *DeltaLower*, respectively. When the battery power is p, then the lower and upper bounds of the approximated delaying period will be $sf_T(p) - DeltaLower$ and $sf_T(p) + DeltaUpper$, respectively. Listing 2 shows how to calculate *DeltaUpper* and *DeltaLower* externally (i.e., outside UPPAAL). The degree of approximation in Fig. 4(b) is:

$$doa_O^{T\prime} = \frac{(2\cdot\Delta)\cdot(\frac{1}{2}\cdot \mathsf{STEP})}{(2\cdot\Delta+\frac{1}{2}\cdot 1)\cdot(\frac{1}{2}\cdot \mathsf{STEP})} = 1 - \frac{1}{4\cdot\Delta+1}, \quad (5)$$

where $\Delta > 0$.

Listing 2. Calculating the asymmetric tolerances of Fig. 4(b).

```
void AsymTolerance4TightAppr (int batteryPower, Delta) {
  if ( ( CAP − batteryPower ) % STEP >= STEP / 2 ) {
    DeltaUpper := Delta;
    DeltaLower := Delta + 1/2;}
  else {
    DeltaUpper := Delta + 1/2;
    DeltaLower := Delta;}
}
```

As can be seen in Eq. (3), (4) and (5), the larger the value of Δ, the better the degrees of the approximation. Furthermore, the degree of approximation of $sf_T()$ is much better than that of $sf()$. This agrees with our intuitive comparisons of Fig. 4(b) with Fig. 4(a).

Similarly, we can achieve under-approximation of continuous resources using step functions.

The variables *FollowingToEnd_Delay*, *Turning_Delay*, *Searching_Delay* and *FollowingToCenter_Delay* that appear as the upper bounds of the location invariants of the Sensors automaton (Fig. 3(a)) are updated in the Battery automaton (Fig. 3(b)) by the relevant step functions (we may use the approximation of either $sf()$ or $sf_T()$), see Appendix A of the full version of this paper. These updates occur at the very beginning of each phase. Since the battery will get lower and lower, the truck will progress more and more slowly phase by phase. In this way, we manage to approximate how the continuous resources affect the system behaviors and timings by using step functions.

We should stress that in this case study, for simplicity, we assume that $sf(p)$ is an affine function of p, which has a negative coefficient. By programming $sf(p)$ with appropriate integer arithmetics, it is possible to approximate other types of resource usage modes such as piecewise affine functions.

4 Verification and controller synthesis

Given a network of timed game automata and a CTL property, UPPAAL can carry out the automatic verification. Furthermore, if a control objective is given, then UPPAAL-TIGA can check whether it can be enforced, and if so, UPPAAL-TIGA can generate a state-based (memoryless) strategy, which is a partial function that maps a semantic state to a controllable action or "delay", i.e.,

$$f : S \to (Act_c \cup \{\mathsf{delay}\}).$$

4.1 Effectiveness and (computational) feasibility

We can verify the autonomous truck system against a number of **p**roperties (P), and synthesize controllers for a number of **c**ontrol **o**bjectives (CO), such as:

// There will be no deadlocked situation.
P1: `A[] not deadlock`

Table 1. Verification and synthesis results.

properties/ control objectives	satisfied?	performance results: states explored	time (s)	memory (KB)	strategy size(KB)
P1	N				
P6	N				
CO6	Y	987307	24.45	271992	52480
CO15	Y	1235764	15.14	192492	12443

System parameters: BP_FULL: 30; MAX_RECHARGE: 10; MAX_RUNS: 10; MAX_USED_POWER: 70. (cf. full version paper)
Experiment platform: Dell PowerEdge 2950, 2×2.5GHz CPU, 32GB RAM; Redhat Enterprise Linux 5 - 64bit; UPPAAL-TIGA 0.13.

// The Truck will never "die" in the journey, and never get lost.
P6: `A[] (bp == 0 imply Battery.Idle) && not Truck.Lost`

// There exists a winning strategy for the system (i.e., Truck) such that the Truck will never "die" in the journey or get lost, no matter how the environment (i.e., Sensors and Battery) behaves.
CO6: `control: A[] (bp == 0 imply Battery.Idle) && not Truck.Lost`

// There exists a winning strategy for the system (i.e., Truck) such that it can be guaranteed to have worked for no less than 8 rounds with no more than 3 rechargings and no more than 78 units power consumptions after the passage of 327 time units.
CO15: `control: A<> (bp == 0 imply Battery.Idle) && (not Truck.Lost) && globalClock == 327 && numberOfRecharge <= 3 && (moreThanMaxRecharge == false) && numberOfRuns >= 8 && (moreThanMaxRuns == false) && (ubp <= 8) && (moreThanMaxUsed == true)`

Table 1 shows the results of verification and controller synthesis for the properties and control objectives (more of them can be found in a full version of this paper). The "Y" ("Yes") answers to the "satisfied?" column indicate that it is possible to synthesize resource-constrained controllers for safety (CO6) and reachability (CO15) winning objectives. As the experimental results in Table 1 suggest, controller synthesis for non-trivial control objectives can be carried out with reasonable time overheads and memory consumptions. The synthesized controllers are of manageable sizes.

4.2 Scalability

Given a system model and a control objective (e.g., CO6), Table 2 shows how the performances of game solving and controller synthesis scale with the range of the variable that is used to model the quantity of the resource (in this case study, the variable *bp* that models **b**attery **p**ower).

As can be seen from Table 2, with the increase of BP_FULL (the constant value of **full b**attery **p**ower), the time overheads and memory consumption increase rapidly (however, the function seems to have a negative derivative). The game solving problem becomes intractable after BP_FULL gets larger than 100. A possible justification of the sensitivity of the range of a resource variable (e.g., *bp*) is that this resource variable determines the values of the durations (e.g., *FollowingToEnd_Delay*, *Turning_Delay*) that will serve as the upper bounds of the location invariants of the relevant timed automata.

Table 2. Method scalability w.r.t. control objective CO6 and model constant BP_FULL.

BP_FULL	performance results: states explored	time(s)	mem(KB)	strategy size(KB)
20	211719	3.31	67388	23291
30	987307	24.45	271992	52480
50	4454542	301.49	1280004	491284
70	13112137	2250.27	3644156	886748

System parameters: MAX_RECHARGE: 10; MAX_RUNS: 10;
Experiment platform: (same as in Table 1)

The restriction of the resource variables to limited ranges indicate that our approach in it current form enables only limited-precision resource modeling.

4.3 Comparing different approximation methods

The results in Table 2 are based on the "normal" approximation using step functions. If we use the tight approximation, then we get the results in Table 3, where all system parameters remain the same as in Table 2.

Table 3. The "tight" version of Table 2.

BP_FULL	performance results: states explored	time(s)	mem(KB)	strategy size(KB)
20	174668	2.74	55660	21690
30	888747	22.19	243284	52464
50	4354723	292.31	1245720	245492
70	13231301	2312.92	3698704	835936

System parameters: MAX_RECHARGE: 10; MAX_RUNS: 10;
Experiment platform: (same as in Table 1)

By comparing Table 3 with Table 2, we notice that tight approximation does not necessarily require more CPU time or more memory. In this case study, it is actually slightly cheaper than the normal approximation version when BP_FULL is relatively small.

4.4 Method applications

Although UPPAAL-TIGA does not support (strict) cost-optimal controller synthesis, we can still try to make worst-case execution time or resource constraints (or both combined) guarantees by synthesizing controllers for the relevant control objectives. Before doing this, we can use wild guess, binary search, and model checking to conduct predetermination of the (near-optimal) relevant values.

5 Conclusions

We show how to model, verify, and synthesize practical controllers for a class of resource-constrained real-time

systems. By employing existing model checker UPPAAL and game solver UPPAAL-TIGA, and modelling the approximations of continuous resource constraints into these tools using step functions and integer arithmetics, the verification and controller synthesis for these systems are fully automated. Experimental results on a case study of an autonomous truck indicate that this approach is feasible and computationally solvable. Another advantage of our approach is that the synthesized strategy is easily understandable. It can be generated as federations of DBMs (Difference Bound Matrices), or combined BDD/CDDs (Clock Difference Diagrams), or the even-closer-to-implementation form of "label-test-jump" style C-code [5]. From a methodological point of view, our approach facilitates the design of timing- and resource constraints-predictable building blocks for component-based development of real-time embedded systems [21].

One limitation of our approach is that the step functions must be designed with care. If a step function-updated variable is used in the upper bounds of the location invariants, then the range of this variable should be of reasonable size. Furthermore, the number of step function-updated variables that are used in the location invariants is limited. When the number grows large, the memory consumption may increase rapidly.

Compared with the `cost` variable approaches in weighted (priced) timed automata, our step function approach has limited precision (granularity) for modeling continuous resources in timed systems. Furthermore, approximations of complex functions such as quadratic functions and logarithmic functions may require variable step lengths or large ranges of the relevant variables, and thus might be hard to manage and difficult to be faithful.

Future work with the autonomous truck case study include: model reductions for improved performances; and enhancing the case to consider more features and more interesting scenarios, e.g., two or more trucks follow a double line track which has a switch, and truck collisions will occur if a certain separation distance is not respected.

At the methodology level, we hope to improve the method scalability (e.g., by defining partial observability to achieve a coarser partitioning of the timed game state space), and to consider the problems of resource-efficient (infinite) (near-)optimal scheduling which may possibly involve verification-guided parameter (pre-)estimations for controller synthesis.

References

[1] R. Alur, M. Bernadsky, and P. Madhusudan. Optimal reachability for weighted timed games. In *Proc. ICALP'04*, 2004.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[3] R. Alur, S. L. Torre, and G. J. Pappas. Optimal paths in weighted timed automata. In *Proc. HSCC'01*, 2001.

[4] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. L. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki. Synthesis of software programs for embedded control applications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(6):834–849, 1999.

[5] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. UPPAAL-TIGA: Time for playing games! In *Proc. CAV'07*, pages 121–125, 2007.

[6] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *Proc. SFM'04*, pages 200–236, 2004.

[7] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. W. Vaandrager. Minimum-cost reachability for priced timed automata. In *Proc. HSCC'01*.

[8] G. Behrmann, K. G. Larsen, and J. I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Performance Evaluation Review*, 32(4):34–40, 2005.

[9] P. Bouyer, T. Brihaye, and N. Markey. Improved undecidability results on weighted timed automata. *Inf. Process. Lett.*, 98(5):188–194, 2006.

[10] P. Bouyer, E. Brinksma, and K. G. Larsen. Staying alive as cheaply as possible. In *Proc. HSCC'04*, 2004.

[11] P. Bouyer, E. Brinksma, and K. G. Larsen. Optimal infinite scheduling for multi-priced timed automata. *Formal Methods in System Design*, 32(1):3–23, 2008.

[12] P. Bouyer, F. Cassez, E. Fleury, and K. G. Larsen. Optimal strategies in priced timed game automata. In *Proc. FSTTCS'04*, pages 148–160, 2004.

[13] P. Bouyer, K. G. Larsen, and N. Markey. Model-checking one-clock priced timed automata. In *Proc. FoSSaCS'07*.

[14] P. Bouyer, K. G. Larsen, N. Markey, and J. I. Rasmussen. Almost optimal strategies in one clock priced timed games. In *Proc. FSTTCS'06*, pages 345–356, 2006.

[15] T. Brihaye, V. Bruyère, and J.-F. Raskin. On optimal timed strategies. In *Proc. FORMATS'05*, pages 49–64, 2005.

[16] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Proc. CONCUR'05*, pages 66–80, 2005.

[17] K. G. Larsen and J. I. Rasmussen. Optimal conditional reachability for multi-priced timed automata. In *Proc. FoSSaCS'05*, pages 234–249, 2005.

[18] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *Proc. STACS'95*, pages 229–242, 1995.

[19] L. Palopoli, C. Pinello, A. L. Sangiovanni-Vincentelli, L. Elghaoui, and A. Bicchi. Synthesis of robust control systems under resource constraints. In *Proc. HSCC'02*, 2002.

[20] C. C. Seceleanu, A. Vulgarakis, and P. Pettersson. Remes: A resource model for embedded systems. In *Proc. ICECCS'09*, pages 84–94, 2009.

[21] S. Sentilles, A. Pettersson, D. Nyström, T. Nolte, P. Pettersson, and I. Crnkovic. Save-ide - a tool for design, analysis and implementation of component-based embedded systems. In *Proc. ICSE'09*, pages 607–610, 2009.

[22] S. L. Torre, S. Mukhopadhyay, and A. Murano. Optimal-reachability and control for acyclic weighted timed automata. In *Proc. IFIP TCS'02*, pages 485–497, 2002.

Appendix A: The Uppaal model declarations of the Autonomous Truck case study

The UPPAAL-TIGA tool can be downloaded from `http://www.cs.aau.dk/~adavid/tiga/`, and the model files of the Autonomous Truck system are here `http://www.cs.aau.dk/~li/papers/AutonomousTruck.zip`.

The UPPAAL global declarations of the model are shown in Listing 3.

Listing 3. UPPAAL global declarations for the autonomous truck system.

```
// the full capacity of the battery of the Truck. (BP_FULL: FULL "Battery Power")
const int BP_FULL := 30;
typedef int [0, BP_FULL] bp_t;
// At the beginning, the Truck is fully recharged. (bp: "battery power")
bp_t bp := BP_FULL;

const int MAX_RECHARGE := 10;
// how many times has the Truck been recharged until now?
int [0, MAX_RECHARGE] numberOfRecharge := 0;
bool moreThanMaxRecharge := false;

const int MAX_RUNS := 10;
int [0, MAX_RUNS] numberOfRuns := 0;
bool moreThanMaxRuns := false;

// uncontrollable actions
broadcast chan trackEndArrived, trackFound, trackCenterArrived;
chan lost, recharged;

// controllable actions
broadcast chan beginFollow, turned;
chan recharge;

// best case durations (BCDs)
const int minFollowingToEnd_Delay := 15;
const int minTurning_Delay := 4;
const int minSearching_Delay := 5;
const int minFollowingToCenter_Delay := 12;
const int minRecharging_Delay := 6;

// step lengths (STEPs)
const int stepSize_FollowingToEnd := 5;
const int stepSize_Turning := 10;
const int stepSize_Searching := 10;
const int stepSize_FollowingToCenter := 5;
const int stepSize_Recharging := 5;

typedef int [minFollowingToEnd_Delay , minFollowingToEnd_Delay +
        (BP_FULL - 0) / stepSize_FollowingToEnd] FollowToEnd_t;
typedef int [minTurning_Delay , minTurning_Delay +
        (BP_FULL - 0) / stepSize_Turning] Turning_t;
typedef int [minSearching_Delay , minSearching_Delay +
        (BP_FULL - 0) / stepSize_Searching] Searching_t;
typedef int [minFollowingToCenter_Delay, minFollowingToCenter_Delay +
        (BP_FULL - 0) / stepSize_FollowingToCenter] FollowingToCenter_t;
typedef int [minRecharging_Delay , minRecharging_Delay +
        (BP_FULL - 0) / stepSize_Recharging] Recharging_t;
```

```
FollowToEnd_t        FollowingToEnd_Delay;
Turning_t            Turning_Delay;
Searching_t          Searching_Delay;
FollowingToCenter_t FollowingToCenter_Delay;
Recharging_t         Recharging_Delay;

clock globalClock;  // for verification only.


// How much time the Truck needs to go from the middle point to the track end?
FollowToEnd_t calc_FollowingToEnd_Delay(bp_t batteryPower) {
  return minFollowingToEnd_Delay +
         (BP_FULL − batteryPower) / stepSize_FollowingToEnd;
}

// How much time the Truck needs to make the turning?
Turning_t calc_Turning_Delay(bp_t batteryPower) {
  return minTurning_Delay +
         (BP_FULL − batteryPower) / stepSize_Turning;
}

// How much time the Truck needs to make the searching?
Searching_t calc_Searching_Delay(bp_t batteryPower) {
  return minSearching_Delay +
         (BP_FULL − batteryPower) / stepSize_Searching;
}

// How much time the Truck needs to go from the landing point to the middle point?
FollowingToCenter_t calc_FollowingToCenter_Delay(bp_t batteryPower) {
  return minFollowingToCenter_Delay +
         (BP_FULL − batteryPower) / stepSize_FollowingToCenter;
}

// How much time the Truck needs to get recharged to full capacity?
Recharging_t calc_Recharging_Delay(bp_t batteryPower) {
  return minRecharging_Delay +
         (BP_FULL − batteryPower) / stepSize_Recharging;
}


const int MAX_USED_POWER := 70;
int [0, MAX_USED_POWER] ubp := 0; // "ubp": the "used battery power" thus far.
bool moreThanMaxUsed := false;


// Whenever the variable is decreased, then "ubp" will be increased accordingly.
void inc_ubp() {
  if ( ubp < MAX_USED_POWER )
    ubp += 1;
  else {
    ubp := 0;
    moreThanMaxUsed := true;
  }
}
```

The UPPAAL local declarations of the Sensor and Battery models are shown in Listings 4 and 5, respectively.

Listing 4. UPPAAL local declarations for the Sensor model.

```
// To manage the timings of the uncontrollable environment events.
clock x;
```

Listing 5. UPPAAL local declarations for the Battery model.

```
// To manage the timing of the (uncontrollabe) battery consumption and battery recharging behaviors.
clock bpClock; // (bpClock: "battery−power−Clock")
```

The system component declarations of the truck system in UPPAAL is shown in Listings 6.

Listing 6. System component declarations for the truck system in UPPAAL.

```
system Sensor, // The (uncontrollable) environment event generator.
       Battery, // The (uncontrollable) recharging and consumption of battery power along time.
       Truck; // How the Truck behaves during its life cycle.
```

Appendix B: Properties, control objective, and verification and controller synthesis results

In this case study, we verify the system against a number of **p**roperties (P), and solve the timed games towards a number of **c**ontrol **o**bjectives (CO), such as:

P1: `A[] not deadlock`
P2: `A[] deadlock imply ( (bp == 0 && not Battery.Idle) || Truck.Lost )`

P3: `A[] not Truck.Lost`
CO3: `control: A[] not Truck.Lost`

P4: `E<> bp == 0 && not Battery.Idle`

P5: `A[] bp == 0 imply Battery.Idle`
CO5: `control: A[] bp == 0 imply Battery.Idle`

P6: `A[] (bp == 0 imply Battery.Idle) && not Truck.Lost`
CO6: `control: A[] (bp == 0 imply Battery.Idle) && not Truck.Lost`

CO7: `control: A<> (bp == 0 imply Battery.Idle) && (not Truck.Lost) && globalClock == 111 && (numberOfRecharge == 0 && moreThanMaxRecharge == false)`

CO9: `control: A[] (bp == 0 imply Battery.Idle) && (not Truck.Lost) && ( (globalClock <= 80) imply (numberOfRecharge == 0 && moreThanMaxRecharge == false) )`

CO12: `control: A<> (bp == 0 imply Battery.Idle) && (not Truck.Lost) && globalClock <= 327 && numberOfRecharge <= 3 && (moreThanMaxRecharge == false) && numberOfRuns >= 8 && (moreThanMaxRuns == false)`

CO13: `control: A[] (bp == 0 imply Battery.Idle) && (not Truck.Lost) && (globalClock <= 323 imply (numberOfRecharge <= 3 && moreThanMaxRecharge == false) )`

CO15: `control: A<> (bp == 0 imply Battery.Idle) && (not Truck.Lost) && globalClock == 327 && numberOfRecharge <= 3 && (moreThanMaxRecharge == false) && numberOfRuns >= 8 && (moreThanMaxRuns == false) && (ubp <= 8) && (moreThanMaxUsed == true)`

For example, property P3 asks whether the Truck can always (invariantly) avoid getting lost. Accordingly, the control objective CO3 asks whether there exists a winning strategy for the system (i.e., Truck) such that the control objective can be enforced, no matter how the environment (i.e., Sensors and Battery) behaves.

Note that not all constants and variables are defined for all properties or control objectives. For example, the constant MAX_USED_POWER and the variable ubp (i.e., the "**u**sed **b**attery **p**ower") are defined only for CO15. See Appendix A for the inline explanations of these variables.

Table 4 shows the results of verification and controller synthesis for those properties and control objectives.

Table 4. Experimental results on method effectiveness and feasibility.

properties/ control objectives	satisfied?	performance results			
		states explored	time (s)	memory (KB)	strategy size(KB)
P1	N				
P2	Y				
P3	N				
CO3	Y	947738	26.02	306640	52464
P4	Y				
P5	N				
CO5	Y	805123	21.04	259972	141568
P6	N				
CO6	Y	987307	24.45	271992	52480
CO7	Y	3482	0.05	132	127
CO9	Y	977464	25.45	271796	87616
CO12	Y	594242	10.00	92104	4019
CO13	Y	1435923	52.74	360724	84640
CO15	Y	1235764	15.14	192492	12443

System parameters: BP_FULL: 30; MAX_RECHARGE: 10; MAX_RUNS: 10; MAX_USED_POWER: 70. (see Appendix A)
Experiment platform: Dell PowerEdge 2950, 2×2.5GHz CPU (Quad Core Intel Xeon), 32GB RAM; Redhat Enterprise Linux 5 - 64bit; UPPAAL-TIGA 0.13.

The "Y" ("Yes") answers to "satisfied?" in Table 4 show that it is possible to synthesize resource-constrained controllers to ensure that:

- (safety) The Truck will never "die" in the journey, and never get lost (CO6);
- (reachability) The Truck can always work for 111 time units without a recharging (CO7);
- (safety) The Truck needs no recharging in the first 80 time units (CO9);
- (reachability) The Truck can be guaranteed to work for no less than 8 rounds with no more than 3 rechargings in the first 327 time units (CO12);
- (safety) The Truck needs no more than 3 rechargings in the first 323 time units (CO13); and
- (reachability) The Truck can be guaranteed to have worked for no less than 8 rounds with no more than 3 rechargings and no more than 78 units power consumptions after the passage of 327 time units (CO15).

As can be seen from Table 4, all these game solving can be carried out with reasonable time overheads ($< 53s$) and memory consumptions (< 390MB). The synthesized controllers are of manageable sizes (< 139MB).