Verification of Safety Properties of a Program for Line Tracing Robot using a Timed Automaton Model

Kozo Okano†, Toshifusa Sekizawa‡, Hiroaki Shimba†, Hideki Kawai‡, Kentaro Hanada†, Yukihiro Sasaki†, and Shinji Kusumoto†

†Graduate School of Information Science and Techlology, Osaka University, Japan
‡Faculty of Informatics, Osaka Gakuin University, Japan
{okano, h-shimba, k-hanada, y-sasaki, kusumoto}@ist.osaka-u.ac.jp
{sekizawa, 09s0100}@ogu.ac.jp

Abstract - Ensuring the reliability of embedded systems has become very important. Reliability may be ensured by a number of formal verification techniques including model checking. We study one such verification technique through a physical example of an embedded system, a line tracing robot. This paper describes how to model the behavior of a line tracing robot program as a network of timed automata and we also present experimental results for this verification by the UPPAAL model checker. The line tracer was built using LEGO Mindstorms and this paper also describes the implementation using LeJOS, a Java development environment for LEGO Mindstorms.

Keywords: Embedded System, Real-time System, Formal Verification, Timed Automaton

1 INTRODUCTION

Embedded systems have become ubiquitous in our society touching many aspects of our daily life. Ensuring the safety properties of these embedded systems has become crucially important. Model checking techniques are often used in order to ensure these properties. Most model checking techniques are based on the finite state machine model. The behavior of the target system is modeled as a tuple of automata. Usually, program variables, such as integer variables, are translated into corresponding state variables. For example, a general 32-bit integer variable might be translated into an 8-bit integer as long as the target program does not use constants with values lying outside the -128 – 127 range available. Even with these constraints such modeling can detect important faults during the design phase. Some embedded systems, however, require time properties in their specification. Several models have been proposed to deal with such real-time systems. One such model is the timed automaton, proposed by Alur and Dill [1]. The most interesting point of the timed automaton model is that it uses clock variables which range over real numbers. Locations and transitions within the timed automaton model have a limited form of syntax for applying constraints on clocks. Timed automaton models can, therefore, naturally represent the behavior of real-time systems. The most popular verifier for the timed automaton model is UPPAAL [2], developed by Wang-Yi's research group. The timed automaton used in UPPAAL is a strong extension of the original timed automaton. It can deal with bounded integer variables and guard expressions on transitions can express constraints on such variables. Several successful applications of UPPAAL have been reported, these include verification of audio-video protocols [3], a gear controller [4], and timeliness properties of multimedia systems [5].

Embedded systems sometimes control continuous systems. For example, a water level controller may observe the level in a certain tank and control the inflow and outflow valves associated with that tank. Note that the water level and the valve flow are usually continuous values. Hybrid automaton models have been proposed in order to deal with such systems consisting of discrete and continuous sub-systems. Several studies have proposed simulators for hybrid systems.

The question at the core of our research is how can we formally verify the behavior of such hybrid systems [6]? Our first step is to find what properties can be verified using conventional verifiers such as UPPAAL by working on a real application. We use a line tracer as our application for the following reasons.

- It contains time properties as design specification;
- We can implement a real system with reasonable costs using LEGO Mindstorms kit [7];
- We can freely describe the control program in Java using LeJOS [8].

A line tracer is an autonomous robot which traces a line painted in black on white background sheet according to values read by color sensors.

Through experiments, we have succeeded in the verification of safety properties of a line tracer, using timed automaton model and UPPAAL. The main contributions of this study are; i) the translation of a control program and its operating environment into a formal model, and ii) to show applicability of model checking for verifying embedded systems. Our study is still in its preliminary stage because we set limitations such as no disturbance and handling only straight lines. However, we believe that this study leads to verification of real embedded systems using model checking.

The rest of the paper is organized as follows. Sec. 2 outlines the foundations of our work. Sec. 3 and Sec. 4 describe the model and implementation of our line tracer. Sec. 5 presents preliminary but promising experimental results and Sec. 6 offers some discussion of these results. Finally, Sec. 7 provides a concluding summary and outlines future plans for our work.

2 PRELIMINARIES

Here, we will briefly outline the background to our work and introduce definitions and notions used in this paper.

2.1 Timed Automata

A timed automaton is an extension of the conventional automaton with clock variables and constraints for expressing real-time dynamics. These are widely used in the modeling and analysis of real-time systems.

Definition 1 (constraints) *We use the following constraints on clocks.*

1. *C represents a finite set of clocks.*
2. *Constraints $c(C)$ over clocks C are expressed as inequalities in the following BNF (Bacchus Naur Form).*

$$E ::= x \sim a \mid x - y \sim b \mid E_1 \wedge E_2,$$

where $x, y \in C, \sim \in \{\leq, \geq, <, >, =\}$, and $a, b \in \mathbb{R}_{\geq 0}$, in which $\mathbb{R}_{\geq 0}$, is a set of all non-negative real numbers.

Time constraints are used to mark edges and nodes of the timed automata and for describing the guards and invariants.

Definition 2 (timed automaton) *A timed automaton $\mathscr{A}$ is a 6-tuple (A, L, l_0, C, T, I), where*

- *A: a finite set of actions;*
- *L: a finite set of locations;*
- *$l_0 \in L$: an initial location;*
- *C: a finite set of clocks;*
- *$T \subseteq L \times c(C) \times A \times 2^C \times L$ is a set of transitions. The second and fourth items are called a guard and clock resets, respectively; and*
- *$I : L \to c(C)$ is a mapping from location to clock constraints, called a location invariant.*

A transition $t = (l_1, g, a, r, l_2) \in T$ is denoted by $l_1 \overset{a,g,r}{\to} l_2$.

A map $v : C \to \mathbb{R}_{\geq 0}$, is called a clock assignment (or clock valuation). We define $(v + d)(x) = v(x) + d$ for $d \in \mathbb{R}_{\geq 0}$ and some $x \in C$.

For guards, resets and location invariants, we introduce some notation for clock valuations. For each guard $g \in c(C)$, a function $g(v)$ stands for the valuation of the guard expression g with the clock valuation v. For each reset r, where $r \in 2^C$, we introduce a function denoted by $r(v)$, and let $r(v) = v[x \mapsto 0], x \in r$. For each location invariant I, we shall introduce a function denoted by $I(l)(v)$, which stands for the valuation of the location invariant $I(l)$ of location l with the clock valuation v.

The dynamics of a timed automaton may be expressed via a set of states and their evaluations. Changes from one state to a new state may be as a result of either the firing of an action or an elapsed time.

Definition 3 (state of timed automaton) *For a given timed automaton $\mathscr{A} = (A, L, l_0, C, T, I)$, let $S = L \times \mathbb{R}^C_{\geq 0}$ be the complete set of states of $\mathscr{A}$, where $\mathbb{R}^C_{\geq 0}$ is a complete set of clock evaluations on C.*

The initial state of $\mathscr{A}$ can be given as $(l_0, 0^C) \in S$. For a transition $l_1 \overset{a,g,r}{\to} l_2$, the following two transitions are semantically defined. The first one is called an action transition, while the latter one is called a delay transition.

$$\frac{l_1 \overset{a,g,r}{\to} l_2, g(v), I(l_2)(r(v))}{(l_1, v) \overset{a}{\Rightarrow} (l_2, r(v))}, \qquad \frac{\forall d' \leq d \quad I(l_1)(v + d')}{(l_1, v) \overset{d}{\Rightarrow} (l_1, v + d)}$$

The semantics of a timed automaton can be interpreted as a labeled transition system.

Definition 4 (semantics of a timed automaton) *For a timed automaton $\mathscr{A} = (A, L, l_0, C, T, I)$, an infinite transition system is defined according to the semantics of $\mathscr{A}$, where the model begins with the initial state. By $\mathscr{T}(\mathscr{A}) = (S, s_0, \overset{\alpha}{\Rightarrow})$, the semantic model of $\mathscr{A}$ is denoted, where $\alpha \in A \cup \mathbb{R}_{\geq 0}$.*

Definition 5 (run of a timed automaton) *For a timed automaton $\mathscr{A}$, a run σ is finite or infinite sequence of transitions of $\mathscr{T}(\mathscr{A})$.*

$\sigma = (l_0, \nu_0) \overset{\alpha_1}{\Rightarrow} (l_1, \nu_1) \overset{\alpha_2}{\Rightarrow} (l_2, \nu_2) \overset{\alpha_3}{\Rightarrow} \cdots$

2.2 Computation Tree Logic

In model checking, specifications are written as logical formulas. Computation Tree Logic (CTL) [9] is a temporal logic suited to dealing with such formulas. Using CTL we are able to describe specifications relating to behaviors of a program for a line tracer robot.

Let AP be a set of atomic propositions. The syntax of CTL is defined as follows:

$$\begin{aligned}\varphi ::= \quad & \bot \mid \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \to \varphi \\ & \mid \mathsf{AX}\varphi \mid \mathsf{EX}\varphi \mid \mathsf{A}\Diamond\varphi \mid \mathsf{E}\Diamond\varphi \mid \mathsf{A}\Box\varphi \mid \mathsf{E}\Box\varphi \\ & \mid \mathsf{A}[\varphi\ \mathsf{U}\ \varphi] \mid \mathsf{E}[\varphi\ \mathsf{U}\ \varphi],\end{aligned}$$

where p is an atomic proposition in AP. The symbols $\bot$, $\top$, $\neg$, $\vee$, $\wedge$ and $\to$ have their usual meanings. The symbols X ("next"), $\Diamond$ ("eventually"), $\Box$ ("globally"), and U ("until") are temporal operators. The symbols A ("always") and E ("exists") are path quantifiers. Intuitively, temporal operators represent statements of a path, and path quantifiers represent statements on one or more paths which are branching forwards from a state. In a CTL formula, temporal operators are preceded by a path quantifier. Due to space limitation, we omit semantics. Please refer to Emerson [9] for details of the semantics of CTL.

For example, a safety property that "variable x is less than 10 for all paths" is written as a CTL formula $\mathsf{A}\Box(x < 10)$.

2.3 UPPAAL

UPPAAL, Wang-Yi et al. [2], is a popular model checker for extended timed automata. It supports model checking for both conventional and timed automata. UPPAAL allows verification of expressions described in an extended version of CTL. In addition, it supports local and global integers and primitive operations on integers, such as addition, subtract

and multiplication with constants. Such expressions are also allowed on the guards of transitions. System models can be created from multiple timed automata which are synchronized via a CCS (Common-Channel Signaling)-like synchronization mechanisms.

An important point is that, with the exception of clocks, the extended timed automaton used in UPPAAL cannot deal with real valued variables. We, therefore, have to round real values to integer values when we model the target systems.

3 MODEL

A "line tracer" is a vehicle which traces a course, assumed to be painted in black on white background using a line of constant width. The line tracer's starting point may or may not be on the course. For example, an oval course (the same as the track used in an athletic field) is frequently used.

A model for a line tracer consists of the following three models:

- Controller Behavior;
- State Transition of Environment;
- Disturbance.

Controller behavior can be modeled using a state machine. Usually, controller programs change the values of some state variables depending on the values of other state variables.

For example, the state variables of a line tracer may be the location of the tracer, the locations of the right and left sensors, the output values of the right and left sensors, direction of the line tracer and the rotation speed of left and right wheels.

The output values of the right and left sensors are used as inputs to the controller. The rotation speed of left and right wheels are set by outputs from the controller.

State transitions in the environment are usually represented by differential equations on state variables. In a hybrid system, such equations are used, while in a finite state model, differential-difference equations are used as an approximation.

For a line tracer, the principle state variables are summarized in Table 1.

We need some additional constants to complete the model, more specifically, these describe the physical dimensions of the line tracer. These are listed in table 2, los, ros are a tuple of (l, a), where l and a are distance and angle relative to the center of the vehicle. Figure 1 illustrates the relationships between state variables and constants.

Let us assume that a line tracer turns at speed directly related to that of the left and right wheels, h_s and l_s. Then its equation of motion can be written as follows.

$$\frac{d\theta}{dt} = \frac{h_s - l_s}{w} \quad (1)$$

$$\frac{dx}{dt} = -r_c \cdot \sin\theta \cdot \frac{d\theta}{dt} \quad (2)$$

$$\frac{dy}{dt} = r_c \cdot \cos\theta \cdot \frac{d\theta}{dt} \quad (3)$$

Table 1: State Variables of a Line Tracer

variable	description
x:	x-coordinate of the center of a line tracer
y:	y-coordinate of the center of a line tracer
θ:	direction of a line tracer
slx:	x-coordinate of the left sensor of a line tracer
sly:	y-coordinate of the left sensor of a line tracer
srx:	x-coordinate of the right sensor of a line tracer
sry:	y-coordinate of the right sensor of a line tracer
wl:	revolution speed of the left wheel of a line tracer
wr:	revolution speed of the right wheel of a line tracer
sl:	the sensed value of left sensor
sr:	the sensed value of right sensor

Table 2: Constants

constant	description
w :	width between left and right wheels of the line tracer
los:	offset to the left sensor from the vehicle center
ros:	offset to the right sensor from the vehicle center

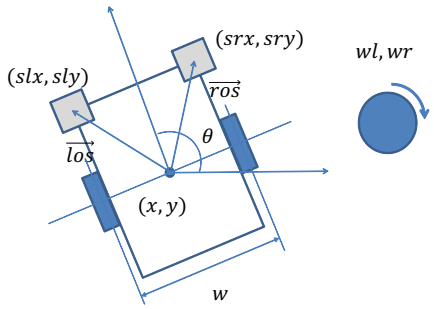


Figure 1: Constants and State Variables

Table 3: Conversion Table for Sine Function

domain of x (degree)	round of $100\times$ sin(x)
$[0, 10)$	8
$[10, 20)$	26
$[20, 30)$	42
$[30, 40)$	57
$[40, 50)$	71
$[50, 60)$	82
$[60, 70)$	91
$[70, 80)$	96
$\vdots$	$\vdots$
$[350, 360)$	-9

$$r_c = \frac{w}{2} \cdot \frac{h_s + l_s}{h_s - l_s} \quad (4)$$

Disturbances can be modeled as an uncertainty error for each of the observed variables. For example, the line tracer's sensor output value, s, may change with an uncertainty value given by the following equation: $s_o = s_r + \varepsilon(s)$, where s_o, s_r, and $\varepsilon(s)$ represent the observed value, ideal value and error in observation, respectively.

3.1 Quantization

The timed automaton used in UPPAAL can model the behavior of our controller. As we have noted above, however, UPPAAL uses integer variables only. Most of the state variables in our model have real values, therefore, we must approximate these as integer variables.

Most of our state variables use trigonometric functions (for example, equations (2) and (3)). Thus, we have to approximate these functions by rounding to integer values as long as we use finite models. The values of trigonometric functions range in [-1,1], this is clearly not suited to integer representation which would give only three values -1, 0, and 1. Therefore, we assume that trigonometric functions range in [-100, 100]. Also we adopt degree as unit for angle. Table 3 shows an approximation conversion table for the sine function.

3.2 Sampling

Another problem is that we cannot deal with functions on time. Usually state variables can be represented as a function on time, however, UPPAAL does not provide suitable functions. Therefore, we have to regard state variables as discrete signals.

Sampling is a useful method for reducing a continuous signal into a discrete signal. For a discrete signal, we can then model its change in time as a timed automaton with update functions.

Let us consider the state variables x, y, θ, slx, sly, srx, sry, wl, wr, sl, and sr. Values for slx and sly are calculated using x and y with the constants listed in Table 2. Values for sl and sr are determined from the values of x, y, los and ros, and a course model, which consists of some parameters and the equations of the course. The values of wl and wr are determined by the controller.

Table 4: Logic for Color Sensors

		RightSensor	
		black	white
LeftSensor	black	go straight	turn left
	white	turn right	go straight

Therefore, we need calculate the current value of x, y and θ using equations (1) $\sim$ (4).

Using sampling and update functions, we can construct a model where the values of variables are updated at a fixed time interval using small deltas. We will describe update expressions in detail in Sec. 5.

4 IMPLEMENTATION

LEGO Mindstorms NXT [10] is a kit for assembling robots and machines with various actuators and sensors. These robots and machines can be programmed with user defined behaviors. The actuators include stepping motors which allow accurate control of rotation angles. There is a range of sensors which include color sensors, touch sensors, and sound sensors. Various programming languages are available for programming the NXT kit. The most popular languages are NXC (Not eXactly C) [11] and LeJOS which is a Java development environment. NXC and LeJOS have program classes for NXT sensors and actuators.

This research uses LeJOS for developing the control program for a line tracer. Our line tracer has two color sensors located left front and right front of the tracing car.

Table 4 shows the controller logic associated with these sensors. If, for example, LeftSensor and RightSensor sense white and black respectively, then the controller issues a "turn right" command to the motors.

The output of these sensors is a bounded integer value. If the value is greater than some threshold level then the controller treats it as white. Wheel motors react independently to turn commands. For example, "turn left" makes left and right wheel motors speed up and slow down, respectively.

Figure 2 shows the controller in LeJOS. Figure 3 shows the physical implementation as a line tracer robot.

5 EXPERIMENTS

Here, we consider an ideal model and we ignore disturbances. In this experiment we use a simple controller program, where the rotation speed of the wheels has only two values, h_s and l_s. Also, we assume that sensors discriminate only between white and black in the robot's operating environment. In other words, the values of sl and sr are dependent on the position of the line tracer. We explicitly model the delays in sensors and actuators. We do this by setting parameters d_s, d_a, and d_t for delay between the time when the program senses color and the time when the sensors obtain the values of colors, delay between the time when the program issues a command and the time when the motor reacts, and sleeping time before next sense-act loop, respectively.

```
import lejos.nxt.Button;
import lejos.nxt.ColorSensor;
import lejos.nxt.SensorPort;
import lejos.nxt.ColorSensor.Color;
import lejos.nxt.LCD;
import lejos.nxt.Motor;
public class Controller {
  public static void main(String[] args)
    throws Exception {
   int rid,lid;
   final int HS = 420, LS = 120, BLACK = 7,
   MS = 360, HSEC = 500;
   Color colorR ,colorL;
   ColorSensor sensorR =
     new ColorSensor(SensorPort.S3);
     // 1(S3):right
   ColorSensor sensorL =
     new ColorSensor(SensorPort.S4);
     // 2(S4):left
   Motor motor = new Motor();
   motor.B.setSpeed(MS);
   motor.C.setSpeed(MS);
   Thread.sleep(HSEC);
     // wait for devices to be stable
   motor.B.forward();
   motor.C.forward();
   while(true) {
     rid = sensorR.getColorID();
     lid = sensorL.getColorID();
     if (rid == BLACK)
       motor.B.setSpeed(LS);
     else       motor.B.setSpeed(HS);
     if (lid == BLACK)
       motor.C.setSpeed(LS);
     else
       motor.C.setSpeed(HS);
     if (Button.readButtons()
       == Button.ENTER.getId())
       break;
   }
  }
}
```

Figure 2: Controller in LeJOS

Figure 3: The Implemented Line Tracer

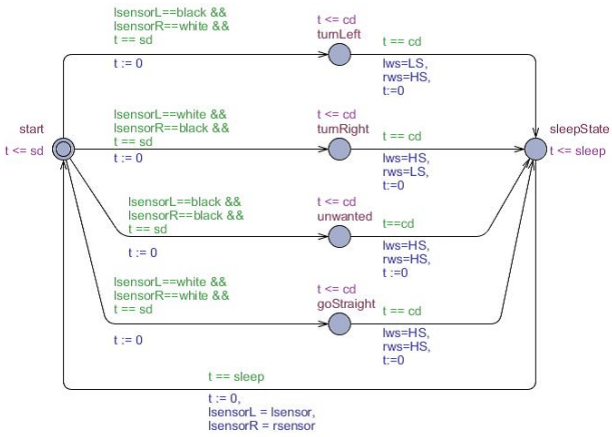


Figure 4: Timed Automaton Representing the Controller

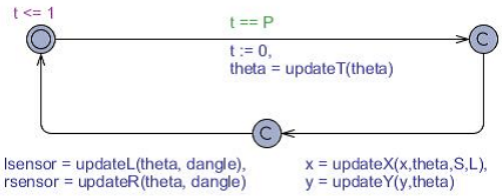


Figure 5: Timed Automaton Representing Update

For modeling and verifying the behavior of a line tracer, it is necessary to model not only the controller program but also position updates depending on the course that the robot is tracing. We therefore use the two models shown in Figs. 4 and 5, which correspond to the controller and updating processes, respectively.

The control behavior model shown in Fig. 4 corresponds to the LeJOS program `Controller` described in Fig. 2. As described in Section 4, the `Controller` decides motor speeds according to the four possible combinations of read values from the color sensors. This timed automaton represents the control program. From the initial location, represented by double circle, one can see that there are four transition each of which corresponds to a pair of sensor values.

Figure 5 shows the timed automaton which updates state variables at regular, discrete time intervals. The automaton periodically calls functions updateX, updateY, updateT, updateL, and updateR which update state variables x, y, θ, sl, and sr, respectively. The automaton deals with these in sequence first updating the value of θ, and then the values of x and y. It then updates the values of sl and sr based on the new values of x, y, and θ. The following equations for θ, x, and y are used by the update functions.

$$\theta' = \theta + \alpha \quad (5)$$

$$x' = x + \frac{wl + wr}{2} \cos\theta \quad (6)$$

$$y' = y + \frac{wl + wr}{2} \sin\theta \quad (7)$$

$$\alpha = 90 \cdot \frac{wr - wl}{w \cdot \pi} \quad (8)$$

If we assume that the time increment between samples is small then the distance moved by the vehicle can be approximated to $(h_s + l_s)/2$. The above equations use this fact. Note that here we are using the pseudo sin $/100$ defined in Table 3 and not the standard $\sin$ function. Also we let the values of the parameter p range in $[0, 359]$ by using an expression $(p + 360)\%360$.

To allow the updating of parameters including positions it is necessary to include the course in model. We assume that the course is a straight line along with x-axis. We also assume that if the value of x becomes greater than 1000 then it is reset to 50. This device let a line tracer run infinitely in the finite state model. The automaton shown in Fig. 5 incorporates information on the straight course.

We obtain a model representing the line tracer by combining the above-mentioned automata. We then checked the correctness of the line tracer program by verifying the following queries

1. $\mathsf{E}\Diamond(900 < x)$.
2. $\mathsf{E}\Diamond(\texttt{C.turnRight})$.
3. $\mathsf{E}\Diamond(\texttt{C.turnLeft})$.
4. $\mathsf{E}\Diamond(\texttt{C.unwanted})$.
5. $\mathsf{A}\Box\neg(\texttt{C.unwanted})$.
6. $\mathsf{E}\Diamond(\texttt{C.goStraight})$.
7. $\mathsf{A}\Box((x > 280) \rightarrow (-100 < y < 100))$.
8. $\mathsf{A}\Box((x > 280) \rightarrow (\theta < 10 \vee 350 < \theta))$.
9. $\mathsf{E}\Diamond((x > 280) \rightarrow \texttt{C.turnRight})$.
10. $\mathsf{E}\Diamond((x > 280) \rightarrow \texttt{C.turnLeft})$.

The first of these, query (1), means that the line tracer will reach the area $x > 900$. Queries (2) and (3) mean that the controller eventually reaches state C.turnRight and C.turnLeft. Queries (4) and (5) mean that the controller eventually reaches state C.unwanted and that the controller never reaches state C.unwanted, respectively, where both of sensors detect black color. Note that queries (4) and (5) contradict each other, i.e., query (5) is negation of query (4). Query (6) means that the controller eventually reaches state C.goStraight.

Queries (7), (8), (9) and (10) use the assumption that the line tracer is in stable state. Note that we consider that the tracer is in a stable state after the point $x = 280$. This can be observed in traces of the simulation. The traces are obtained from the UPPAAL using the simulation mode view. Queries (7) and (8), respectively, mean that the line tracer roughly keeps on track and moves in the appropriate direction when it is in its stable state. The last two queries mean that the line tracer eventually turns left or right even if the tracer is in stable state.

Each of these queries (except for query (4)) was verified successfully using the parameters in Table 5. Verifications were performed within one second using UPPAAL ver. 4.0.13 on Windows 7 64 bit OS, Intel Core i7 960 3.20GHz, with 12 GB memory. Figure 6 shows verification process using UPPAAL.

Table 5: Parameters used for Verification

params	value	description
wc:	100	width of the track line
w:	120	width between left and right wheels of the line tracer
los:	(180, 30°)	offset to the left sensor from the vehicle center
ros:	(180, −30°)	offset to the right sensor from the vehicle center
h_s:	12	high speed
l_s:	6	low speed
x_0:	-200	initial value of x-coordinate of the center of the vehicle
y_0:	200	initial value of y-coordinate of the center of the vehicle
θ_0:	340°	initial value of direction of the vehicle
d_s:	1	time delay of sensors
d_a:	1	time delay of actuators
d_s:	2	periodical sleeping time

Every query, except for query (4), represents a specification that the line tracer has to satisfy. Ideally, conjunction of all queries should be verified. Unfortunately, UPPAAL does not allow nesting of path quantifiers in a formula so we verified the queries one by one. However, when we consider all the queries together, they describe necessary conditions to check the behavior of the line tracer.

Let us consider the effect of changing parameters. Query (4) is verified if we change the parameters los and ros to (170, 30°) and (170, -30°). As an obvious consequence of this Query (5) is no longer verified. This is because changing the positions of the two sensors, los and ros, reduces the distance between them. If the width of line is unchanged then the possibility of both sensors detecting black becomes higher. Verification of query (4) means that state C.unwanted is eventually reached.

6 DISCUSSION

Here we discuss our experiments and consider related work in the field of control engineering.

6.1 Discussions on the Experiments

These experimental results are not enough to convince us that the line tracer runs safely. They do, however, show that from the theoretical point of view, our approach of using a verifier for timed automata, will work.

The parameters used in verification are not the same as the those used in the implementation. For example, the width ws between the left sensor and the right sensor is 180. This value of ws is greater than 120, the width of the line tracer. This might lessen the validity of the model. The parameters are, however, acceptable because the value ws is greater than 100, the width of the track, which allows the control logic to work.

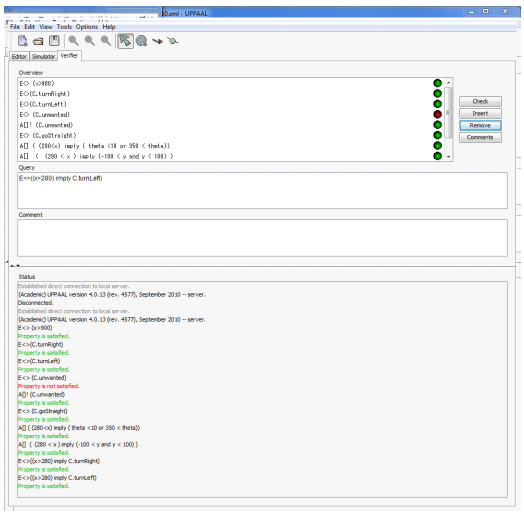


Figure 6: Verification using UPPAAL

Also the wheel speed 6 or 12 is acceptable given the size of the line tracer.

The workload involved in modeling the system was considerable (it took over 2 man-months) due to our limited knowledge of modeling, especially that of dealing with with continuous models. Some of parameters in Table 5 are very sensitive to variation, if these values are different by only a small amount then the behavior of the whole system changes and, consequently, verification will fail. For example, if we change the value of d_s to 4, then the verification fails.

One might think that low wheel speed increases the possibility of successful verification. In other words, the slower a line tracer moves, the more successively it keeps on track. However, due to quantization, a low wheel speed causes the delta values per unit of time to be 0 in our model. Therefore, we cannot use a value smaller than 6 as the low speed for the model's wheels. This problem can be resolved by increasing the physical sizes in the model. However, such a revision causes a so-called state explosion which means that model checker either cannot respond in a reasonable time or it exhausts its available memory.

Nevertheless, this still shows the importance of design analysis and verification in the early stages of development.

During the modeling, we would have liked to have had an automated generation tool which translated from an abstract parameter model to a concrete UPPAAL model, as well as a simple tool to analyze counter-examples and simulation results obtained from UPPAAL. Such tools would have been very useful in refining the model.

6.2 Related Work

It is important to assure reliability in the field of control engineering and other fields. In this section, we briefly describe related work on formal verification of applications in control engineering.

One of major approaches to verification of control systems is conversion from continuous into discrete systems using various techniques such as abstraction and reduction. This approach allows verification using ordinary model checking. For example, T. K. Iversen et al. reported on the verification of a real-time control programs using UPPAAL [12]. In their paper, the authors constructed a brick sorter system using LEGO RCX and wrote control programs in Not Quite C (NQC). The paper also describes the verification of safety and liveness properties by automatic translation from the control programs. The brick sorter system has similar characteristics to our research at the point that writing programs to simulate real-time systems. However, the brick sorter system is essentially a discrete system even though it contains time dependencies.

In verification of robotics, Sharygina et al. carried out a survey of model checking of the control system of NASA robotics systems [13]. In this survey, the authors summarize various techniques for verification and show verification of a robot control system. Safety and liveness properties are verified but these properties were not related to continuous dynamics. Even though survey does not cover the handling of continuous dynamics, it is a good resource. As a similar area, the verification of a real vehicle is described by Proetzsch et al.[14]. Even though our aim is the verification of continuous systems, our approach in reflects those above, i.e., conversion to timed automaton using quantization and sampling techniques.

There are alternative approaches to handling control systems. For the analysis of real models, such as cyber-physical systems, hybrid systems [15] seem to promise models which reflect the target systems more precisely. This is because a hybrid system is one in which continuous and discrete dynamics are mixed with time progression. Several approaches have been proposed to deal with hybrid systems. One of these approaches is hybrid automaton [16] which is a formal model for describing mixed discrete-continuous systems. HyTech [17] is a model checker for linear hybrid automata. Another approach is hybrid constraint languages such as Hybrid CC [18] and HydLa [19]. These languages are declarative and provide the power to write programs with logical formulas. Execution environments for these languages are available, Hybrid CC interpreter and Hyrose, respectively.

As with many control systems, a line tracers can be considered as a hybrid system by describing their movements using differential equations and their control programs in discrete time. It is generally accepted that real embedded systems are too big to fully verify. Therefore, it is usual to focus on important behaviors. As an example of hybrid approaches, Fehnker et al. [20] described the verification of the behaviors of a line tracer by constructing a model using hybrid I/O automata and correctness proofs. In that paper, the authors presented verification of safety property, that is, a line tracer should move along a straight line and never run off. However, the authors noted that some time details, such as time delay between two motors, were not considered .

7 CONCLUSION

We have modeled the controller of a line tracer as a timed automata. We have also verified the model so as to ensure that the line tracer keeps on track by using the UPPAAL model

checker. In this paper, we dealt with simplified conditions such as there being no disturbance and the course being straight. However, we believe that our study shows the applicability of model checking for verifying real embedded systems.

Future plans for this research involve exploring other control algorithms, hybrid modeling and detailed analysis of delays in the model. We would like to introduce a PID controller (proportional-integral-derivative controller), which is a widely used feedback control system. PID control is widely used in control systems in control engineering. When PID control is applied to a line tracer, it enables smooth motion. However, PID control requires the maintenance of some historical records of state variables, and also requires complicated calculation. This approach seems to be better suited to hybrid modeling. We intend to use hybrid model, as well as verifiers and simulators to determine suitable parameters for PID control. Finding suitable parameters for tuning PID controllers to deal with particular problems is difficult and we believe that our approach may prove viable.

Another research direction is timing analysis of motor delays. From preliminary experiments, we have found that motor delay cannot be ignored in the design of a controller program if we wish to obtain a high quality controller.

ACKNOWLEDGMENTS

This research is partially supported by Grant-in-Aid for Scientific Research (C) (21500036). We also thank Professor Nobukazu Yoshioka, National Institute of Informatics (NII), for providing several education materials related to modeling and verification of a line tracer.

We express our sincere gratitude to Associate Professor R. D. Logie, Osaka Gakuin University, for his proofreading.

REFERENCES

[1] R. Alur, and D. L. Dill, “A theory of timed automata,” Journal of Theoretical Computer Science, 126(2), pp.183-235 (1994).

[2] J. Bengtsson, and W. Yi, “Timed Automata: Semantics, Algorithms and Tools,” Lecture Notes in Computer Science, Vol.3098, pp.87-124 (2004).

[3] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, “Verification of an Audio Protocol with bus collision using UPPAAL,” Lecture Notes in Computer Science, Vol.1102, pp.244-256 (1996).

[4] M. Lindahl, P. Pettersson, and W. Yi, “Formal Design and Analysis of a Gear Controller: An Industrial Case Study using UPPAAL,” Lecture Notes in Computer Science, Vol.1384, pp.289-297 (1998).

[5] B. Bordbar, and K. Okano, “Verification of Timeliness QoS Properties in Multimedia Systems,” Lecture Notes in Computer Science, Vol.2885, pp.523-540 (2003).

[6] J. Fitzgerald, P. G. Larsen, K. Pierce, M. Verhoel, and S. Wolff, “Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems,” Proc. of IFM 2010, pp.12-26 (2010).

[7] B. Bagnall, “Intelligence Unleashed: Creating LEGO NXT Robots with Java,” Variant Press (2011).

[8] LeJOS Java forLEGO Mindstorm, `http://lejos.sourceforge.net`

[9] E. A. Emerson, “Temporal and Modal Logic,” Handnook of Theoretical Computer Science Vol.B: Formal Methods and Semantics, pp.997-1072, The MIT Press (1990).

[10] LEGO Mindstorms NXT Official website, `http://www.legoeducation.jp/mindstorms/`

[11] NXC Tutorial, `http://bricxcc.sourceforge.net/nbc/nxcdoc/NXC_tutorial.pdf`

[12] T. K. Iversen, K. J. Kristoffersen, K. G. Larsen, M. Laursen, R. G. Madsen, S. K. Mortensen, P. Pettersson, and C. B. Thomasen, “Model-Checking Real-Time Control Programs: Verifying LEGO Mindstorms Systems Using UPPAAL,” Proc. of ECRTS. pp.147-155, IEEE Computer Society (2000).

[13] N. Sharygina, J. Browne, F. Xie, R. Kurshan, and V. Levin, “Lessons Learned From Model Checking a NASA Robot Controller,”, Formal Methods in System Design, Vol.25, No.2-3, pp.241-270 (2004).

[14] M. Proetzsch, K. Berns, T. Schuele, and K. Schneider, “Formal Verification of Safety Behaviours of the Outdoor Robot RAVON.”, Proc. of 4th International Conference on Informatics in Control, Automation and Robotics, Robotics and Automation, ICINCO-RA, pp.157-164, INSTICC Press (2007).

[15] J. Lunze, and F. Lamnabhi-Lagarrigue, “Handbook of Hybrid Systems Control: Theory, Tools, Applications,” Cambridge University Press (2009).

[16] T. A. Henzinger, “The theory of hybrid automata,” Proc. of Eleventh Annual IEEE Symposium on Logic in Computer Science, LICS ’96, pp.278-292 (1996).

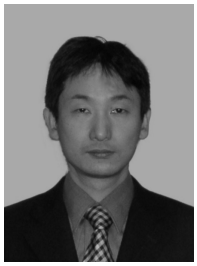
[17] T. A. Henzinger, Pei-Hsin Ho, and H. Wong-Toi, “HYTECH: A model checker for hybrid systems,” Lecture Notes in Computer Science, Vol.1254, pp.460-463 (1997).

[18] V. Gupta, R. Jagadeesan, V. Saraswat, and D. G. Bobrow, “Programming in hybrid constraint languages,” Lecture Notes in Computer Science, Vol.999, pp.226-251 (1995).

[19] K. Ueda, H. Hosobe, and D. Ishii, “Declarative semantics of the hybrid constraint language HydLa,” Computer Software, JSSST, 28(1) pp.306-311 (2011). (in Japanese).

[20] A. Fehnker, F. W. Vaandrager, and M. Zhang, “Modeling and verifying a lego car using hybrid I/O automata,” Proc. of 3rd Int. Conf. on Quality Software, pp.280-289. IEEE Computer Society (2003).

(Received October 19, 2012)
(Revised December 1, 2012)



Kozo Okano received the BE, ME, and Ph.D degrees in Information and Computer Sciences from Osaka University, in 1990, 1992, and 1995, respectively. Since 2002, he has been an associate professor in the Graduate School of Information Science and Technology, Osaka University. In 2002, he was a visiting researcher of the Department of Computer Science, University of Kent at Canterbury. In 2003, he was a visiting lecturer at the School of Computer Science, University of Birmingham. His current research interests include formal methods for software and information system design. He is a member of IEEE_CS, IEICE of Japan and IPS of Japan.

Toshifusa Sekizawa received his M.S. degree in physics from Gakushuin University in 1998, and Ph.D. in information science and technology from Osaka University in 2009. He previously worked at Nihon Unisys Ltd., Japan Science and Technology Agency, and National Institute of Advanced Industrial Science and Technology. He is currently working at Osaka Gakuin University. His research interests include model checking and its applications.

Hiroaki Shimba received the BI degree from Osaka University in 2012. He is a master course student in Osaka University. His research interests include model translation, especially translation between OCL and JML.

Hideki Kawai is an undergraduate student at Faculty of Informatics, Osaka Gakuin University. His research is interested in dependability of control engineering. He is currently researching behaviors of vehicle using model checking.

Kentaro Hanada received the BI degree from Osaka University in 2011. He is a master course student in Osaka University. His research interests include model translation, especially translation between OCL and JML.

Yukihiro Sasaki received the BI degree from Osaka University in 2012. He is a master course student in Osaka University. His research interests include automatic testcase generation, especially for dynamic generation of assertion.

Shinji Kusumoto received the BE, ME, and DE degrees in information and computer sciences from Osaka University in 1988, 1990, and 1993, respectively. He is currently a professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include software metrics and software quality assurance technique. He is a member of the IEEE, the IEEE Computer Society, IPSJ, IEICE, and JFPUG.