Model-Checking Real-Time Control Programs

Verifying LEGO® MINDSTORMS™ Systems Using UPPAAL

Torsten K. Iversen Kåre J. Kristoffersen Kim G. Larsen Morten Laursen
Rune G. Madsen Steffen K. Mortensen Paul Pettersson Chris B. Thomasen

BRICS,[*] Department of Computer Science, Aalborg University,
Fredrik Bajersvej 7E, DK-9220 Aalborg East, Denmark.
E-mail: {`torsten,jelling,kgl,morten,rune,kondrup,paupet,chris`}`@cs.auc.dk`

Abstract

In this paper, we present a method for automatic verification of real-time control programs running on LEGO® RCX™ bricks using the verification tool UPPAAL. *The control programs, consisting of a number of tasks running concurrently, are automatically translated into the timed automata model of* UPPAAL. *The fixed scheduling algorithm used by the LEGO® RCX™ processor is modeled in* UPPAAL, *and supply of similar (sufficient) timed automata models for the environment allows analysis of the overall real-time system using the tools of* UPPAAL. *To illustrate our techniques we have constructed, modeled and verified a machine for sorting LEGO® bricks by color.*

1 Introduction

Real-time systems consist of a control program operating in a time-sensitive environment (a piece of hardware or a physical plant). As such it is imperative that the services of the control program are offered in a timely manner in order that the behavior of the environment is supervised and controlled appropriately. The interaction between the (discrete) control program and the (potentially analog) environment takes place via various sensors and actuators (see Figure 1).

Designing and verifying real-time systems not only requires a model of the tasks constituting the control program, but also calls for a model of the environment which is sufficiently detailed for the properties of concern. As the interaction between the tasks of the control program and the environment is assumed to be time-sensitive it is important that our model also takes into account the overhead introduced by the particular algorithm applied for scheduling tasks.

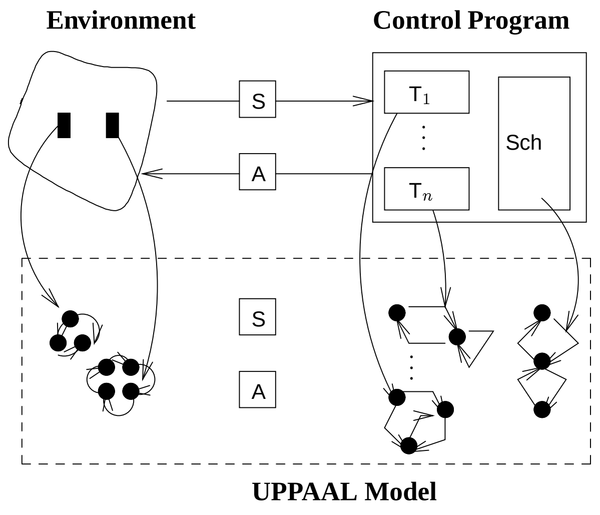

Figure 1. In our framework the tasks and scheduler are automatically translated to timed automata. Sensors and actuators are modeled by integer variables containing their readings and settings. The user provides a timed automaton model of the environment.

In this paper, we present a method for automatic verification of real-time control programs running on LEGO® RCX™ bricks using the verification tool UPPAAL [9]. The LEGO® RCX™ brick (see Figure 9) is part of LEGO® MINDSTORMS™ and LEGO® ROBOLAB®. The RCX™ brick is essentially a big LEGO® brick with a small processor inside. The brick has six communication ports to the environment: three sensor inputs and three actuator outputs.

[*] **B**asic **R**esearch **I**n **C**omputer **S**cience, Centre of the Danish National Research Foundation.

```
*** Task 0 = main
000 PlaySound  1          51 01
002 Delay      100        43 02 64 00
006 Jump       0          72 87 00
```

Figure 2. A simple RCX™ program that periodically plays a sound.

An infrared port enables communication between RCX™ bricks and allows for easy download of programs

The control programs — consisting of a number of tasks running concurrently — are automatically translated into the timed automata model of UPPAAL. The fixed scheduling algorithm used by the LEGO® RCX™ processor is modeled in UPPAAL, and supply of similar (sufficient) timed automata models for the environment allows analysis of the overall real-time system using the tools of UPPAAL.

Related work includes the work of Corbett [4] and Björnfot [3], which both address the problem of analyzing Ada programs with preemptive scheduling using verification tool such as HyTech [5] and UPPAAL [9]. More closely related is our own work [7]. Here, we address the dual, and more difficult problem of synthesizing RCX™ control programs from UPPAAL models. The work by Hune in [6] also deals with the exact same problem of synthesing UPPAAL models from RCX™ control programs. Compared to the work by Hune our modeling of the scheduling algorithm is simpler and our paper presents the first real application of the methodology.

In Section 2, we present the preliminaries of this paper, including a brief description of the programming language(s) used to program the RCX™ bricks, and a brief summary of UPPAAL. Section 3, describes our translation of control programs and modeling of the scheduler. Section 4 and 5 then applies our method to the modeling and verification of a sorting machine for LEGO® bricks[1]. UPPAAL is successfully applied to prove that indeed the bricks are sorted correctly. Additionally UPPAAL succeds in pointing out an error in a setting where the model of the environment does *not* satisfy the assumptions made by the control program.

2 Preliminaries

2.1 The RCX

We shall consider control programs executing on the LEGO® RCX™ brick [11], part of the LEGO®MINDSTORMS™ and LEGO® ROBOLAB® series. The RCX™ brick is basically a big LEGO® brick

[1]Ordinary, old-fashioned, LEGO® bricks.

```
task main{
   while( true ) {
      PlaySound( 1 );
      Sleep( 100 );
   }
}
```

Figure 3. A simple NQC program that periodically plays a sound.

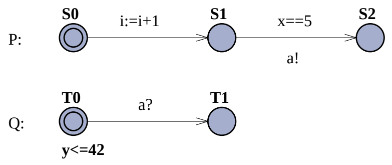

Figure 4. A simple UPPAAL model.

with a built in (small) processor. It has a speaker, a display, three sensor input ports, three actuator output ports, and an infrared port for communication between bricks and for easy downloading of programs from an external computer.

The RCX™ brick is equipped with an interpreter capable of handling programs with up to ten tasks and 32 integer variables. It interprets an assembly like low level language which we will call RCX™ byte code. A small RCX™ byte code program with one task that plays a beep sound every second is shown in Figure 2.

2.2 Not Quite C

The RCX™ brick is shipped with a graphical programming language. We have chosen to use a programming language with a textual C like syntax called Not Quite C, `NQC` [2]. It enables programming of the RCX™ at a higher level of abstraction than the RCX™ byte code language and is more appropriate for our purposes than a graphical language.

A `NQC` program, like a RCX™ byte code program, is restricted to ten tasks and 32 global integer variables. In addition, the `NQC` programs may be defined using sub routines. A `NQC` program always starts by executing a dedicated `main` task. The remaining tasks are started by other tasks. A simple `NQC` task with the same behavior as the RCX™ byte code program in Figure 2 is shown in Figure 3.

2.3 UPPAAL

UPPAAL[2] is a modeling, simulation, and verification tool for real-time systems modeled as networks of timed au-

[2]See the web site `http://www.uppaal.com/`.

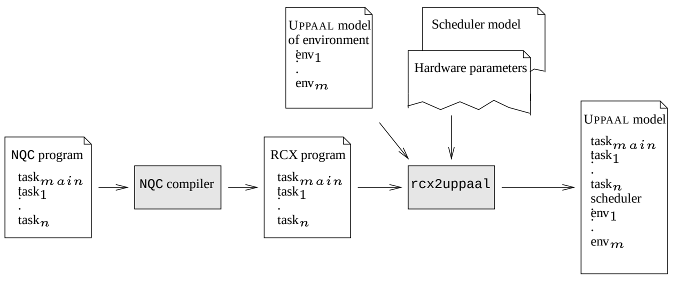

Figure 5. Overview of translation from NQC programs to timed automata.

tomata [1] extended with data types such as bounded integer variables, arrays etc. For a thorough description of UPPAAL see [9].

Figure 4 shows an UPPAAL model consisting of two parallel timed automata **P** and **Q** with two and three locations respectively (i.e. **S0** to **S2** and **T0** to **T1**). They use the two clocks `x` and `y`, an action channel `a`, and an integer variable `i`. Initially all clocks and integer variable values are zero.

When automata **P** takes the transition from **S0** to **S1** the integer variable `i` is incremented by one. For **P** to go from location **S1** to **S2** the clock `x` is required to be exactly 5 (by the guard `x==5`). On the same transition, the automaton must also synchronize on channel `a` with automata **Q**, as the edge is labeled with the action `a!`. Furthermore, automaton **Q** can not delay in location **S0** for more than 42 time units because of the location invariant `y<=42`.

UPPAAL can check reachability and invariance properties of boolean combination of automata locations, and clocks and integers constraints. In UPPAAL, `E<>`ϕ expresses that it is possible to reach a state satisfying ϕ. Dually, `A[]`ϕ expresses invariance of ϕ. For example, property `E<>`**Q.T1** specifies that automata **Q** can reach location **T1**. The property `A[]`(**y**>**42** `imply` **Q.T1**) states that automata **Q** is always operating in location **T1** when clock `y` is greater than 42. The two properties are both satisfied in the model.

3 From RCX™ to Timed Automata

In this section we describe an automated procedure for translating `NQC` control programs into the timed automata modeling language of the UPPAAL tool. The procedure is illustrated in Figure 5. It is performed in two steps.

The first step is to compile the input `NQC` program, consisting of a set of `NQC` tasks and (global) variables, into an RCX™ byte code program. This is done by the `NQC` compiler [2].

In the next step, the program `rcx2uppaal` uses the

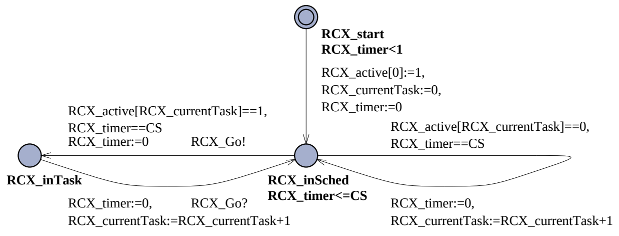

Figure 6. Timed automata model of a round-robin scheduler.

produced RCX™ byte code and a set of timing parameters specifying the timing requirements for performing the various run-time instruction of the RCX™ brick. From this input a network of timed automata is produced in which each task of the RCX™ program is modeled as a timed automaton reflecting its operational and timing behavior. The (global) variables are modeled as (global and bounded) integer variables. In addition, the `rcx2uppaal` program also takes as input timed automata models of the RCX™ scheduling algorithm and the components in the surrounding environment of the RCX™ brick, affecting the sensors and actuators connected to its input and output ports. The scheduler model is instantiated with timing parameters, merged with the model of the environment and the model of the RCX™ program to produce the final output: a network of timed automata in the input format of the UPPAAL tool.

In the reminder of this section we describe the scheduler model, the translation of RCX™ programs, and the model of interaction with the environment in more detail.

3.1 The RCX™ Scheduler

The RCX™ brick scheduler executes the tasks in round-robin order. The method described here is not restricted to a particular scheduling algorithm. We use a round-robin scheduler in the presentation as it is simple, well-known, and moreover the actual algorithm used in the RCX™ bricks.

A timed automata model of the round-robin scheduler is shown in Figure 6. It uses the bit array `RCX_active`, the (bounded) integer variable `RCX_currentTask`, and the clock `RCX_timer` to implement round-robin scheduling. The variable `RCX_currentTask` is always assigned the value of the executing task (or the task to be executed next if no task is executing). The bit array `RCX_active` has one element for each task indicating if the corresponding task is active (i.e. waiting to be executed) or not. We use $a[i]$ to denote the value of element i in array a. Task i is active if `RCX_active[`i`]`$= 1$, otherwise inactive.

When the scheduler is started, it first executes the special task$_{main}$ (or task 0). This is reflected in Fig-

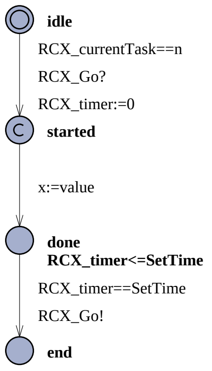

Figure 7. Timed automata model of the `Set` instruction.

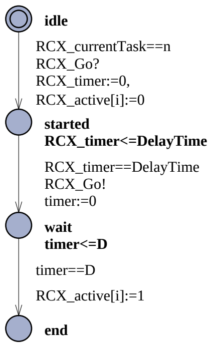

Figure 8. Timed automata model of the `Delay(D)` instruction.

ure 6 by the assignments `RCX_active[0]:=1` and `RCX_currentTask:=0` on the transition from location **RCX_start** to **RCX_inSched**, which activates task 0 and make it the next to be executed.

After initialization, the scheduler executes the active tasks in fixed order by repeatedly incrementing `RCX_currentTask` by one and executing the corresponding task. If task `RCX_currentTask` is active (i.e. if `RCX_active[RCX_currentTask]`=1), the scheduler takes the left loop (to location **RCX_inTask**) and starts the task by signaling the task `RCX_Go!` after a delay of `CS` time units, where `CS` is the overhead time required by the scheduler. When the task has finished executing a single instruction it signals `RCX_Go?` to the scheduler which proceeds by incrementing `RCX_currentTask` by one. If task `RCX_currentTask` is inactive the scheduler performs the similar but simpler right loop, in which the signaling on channel `RCX_Go` is not performed (i.e. the task is not executed).

The maximum number of tasks in the RCX™ brick is restricted to ten. If the number of tasks is i and $i < 10$, the tasks $i, \ldots, 9$ remain inactive throughout the execution. To ensure that the tasks are executed in round-robin order, the domain of the variable `RCX_currentTask` is $0, \ldots, 9$. In UPPAAL, integer variables are always assigned modulo their domain [3]. Therefore, incrementing `RCX_currentTask` by one results in zero if the value is nine, which is exactly the desired behavior.

[3]As a side-effect, the UPPAAL verifier generates a warnings when a variable is assigned a value outside its domains.

3.2 The RCX™ Program

An RCX™ program consist of a set of tasks and variables. The variables are compiled into (bounded) integer variables in the UPPAAL modeling language. The tasks are compiled into timed automata by replacing each RCX™ byte code instruction with a corresponding timed automata model of the instruction. In this section we describe how this is done by exemplifying with two instructions. For a detailed description of how all the RCX™ instructions are modeled as timed automata we refer to [10, 8, 6]. See also Figure 15 in the Appendix, which shows the timed automata generated from the RCX™ task in Figure 2.

Most instructions are translated into a structure similar to the `Set` instruction, show in Figure 7, which assigns variable `x` to a given value. The instruction is started on the first transition (to location **started**) were it synchronizes with the scheduler `RCX_Go` if `RCX_currentTask`=i. Thus, the starting time of the instruction is controlled by the scheduler. The actual effect of the instruction is modeled on the two next transitions were the variable `x` first is assigned. The automaton then delays in location **done** for the time required to perform the `Set` operation in the RCX™ brick. Finally, in the last transition, the instruction returns the control to the scheduler and waits to perform its next instruction. Thus, the **end** location coincides with the **start** location of the next instruction.

Perhaps the most interesting RCX™ instruction is `Delay(D)`, which delays the task for `D` time units. Its timed automata model for task i is shown in Figure 8. The instruction is executed in two stages. In the first stage (the two first transitions in Figure 8) the task in-activates itself by resetting its entry (i.e. entry i) in the `RCX_active` bit array to zero. It then delays in location **started** for the time required to perform these operations (i.e. `DelayTime`), resets the (local) clock `timer`, and returns the control to the scheduler. In the second stage, in location **wait**, the task delays for `D` time units before (re-)activating itself (i.e. assigning 1 to the variable `RCX_active`[i]) and proceeding to location **end**.

Another 30 RCX™ instructions occur in the `NQC` compiler output[4]: the arithmetic and logical instructions are similar to the `Set` instruction; the control flow instructions, such as `Test`, `SetLoop`, and `Jump` are straight forwardly modeled as branching timed automata; and the control instructions, such as `StartTask` and `StopTask` are variations of the `Set` instruction automaton updating the `RCX_active` array. The only instructions not covered by our `rcx2uppaal` program are the two RCX™ datalog instructions `SetLog` and `DataLog` used to log variable values during program execution [5]. However, for our purposes it is sufficient to model them as empty instructions consuming the correct time, as the datalog can not be accessed by the program or have any affect on the environment.

3.3 The Environment Interface

In addition to the RCX™ program, the `NQC` compiler also generates a set of global variables for the interface between the tasks and the environment of the RCX™ brick. The `rcx2uppaal` program compiles these integers to variables in the timed automata modeling language extended with (bounded) integers used in UPPAAL.

Each port is typically modeled by a tuple of variables, e.g. input port 1 is described by the triple (`RCX_InType_1`, `RCX_InMode_1`, `RCX_IN_1`), where `RCX_InType_1` specifies the type of sensor connected to port 1, `RCX_InMode_1` how the sensor values are to be interpreted, and `RCX_IN_1` is the actual sensor value. Thus, if a light sensor is connected to port 1, `RCX_InType_1=SENSOR_TYPE_LIGHT`, `RCX_InMode_1=SENSOR_MODE_PERCENT`, and `RCX_IN_1=50` should be interpreted as a light intensity of 50%.

All communication between the RCX™ program and its environment goes through the tuples modeling the environment interface. The interaction is implemented using the ordinary instructions, such as the `Set` instruction described above.

4 LEGO® Example: a Brick Sorter

To demonstrate our techniques we have built a sorting machine for LEGO® bricks. A conveyor belt equipped with a color sensor in one end and a kicking arm at the other end

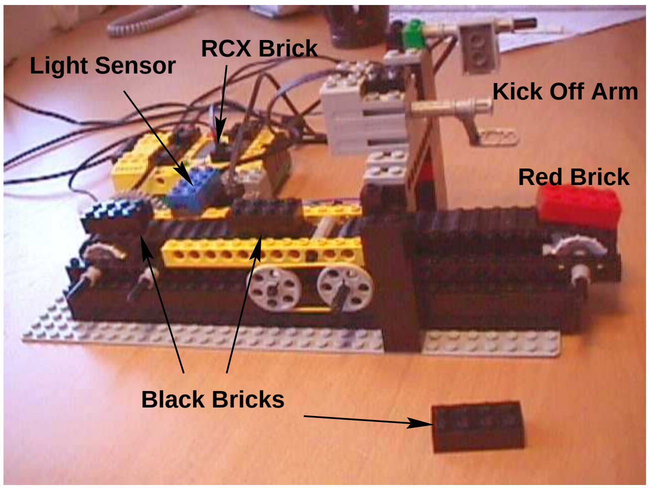

Figure 9. The LEGO® brick sorter.

is carrying black and red LEGO® bricks from left to right, (see Figure 9). The idea is that the red bricks should pass uninterrupted by the kick arm and thus run all the way to the very end of the belt, while all the black ones should be kicked off by the arm.

4.1 Overview of Brick Sorter

The LEGO® test setting is sketched in Figure 10. The conveyor belt is running from left to right driven by Motor A. Shortly after being placed at the very left end of the belt a brick will pass a light intensity sensor (Sensor 1). When the brick passes the sensor its color is registered in the following way: the undisturbed sensor displays a light intensity of 50% (during daytime). Now, a black brick in front of the sensor reduces the light intensity to a value of 35%, while a red brick causes no significant change in light intensity. Thus, by comparing the readings of the sensor

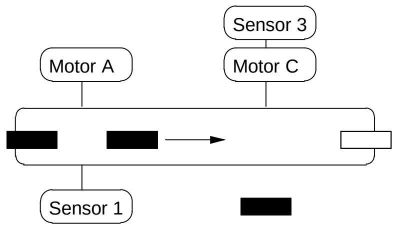

Figure 10. The brick sorter (top view). The positions of black and red bricks correspond to those in Figure 9 .

value with a suitable threshold, say 42%, we will be able to observe when a black brick passes Sensor 1. Notice that in this way the red bricks are ignored, but this is just fine,

[4] There are more than 32 RCX™ byte code instructions but all of them do not occur in the output of the `NQC` compiler.

[5] The datalog instructions are mainly used for debugging purposes.

since precisely these bricks are *not supposed* to be kicked of the belt.

When the presence of a black LEGO® brick is observed by Sensor 1 a timer will be reset, such that the kick arm (situated at Motor C) will be invoked at just the right moment. For simplicity we have restricted the model to handle at most two black bricks at a time between Sensor 1 and the kick arm, and thus we can do with two timers (`Timer(1)` and `Timer(2)`).

4.2 Control Programs: Observe and Kick

There are two control tasks, `main` and `kick_off`, (see Figure 11 and 12). Initially the `main` task configures Sensor 1 (using the integer `IN_1`) and Sensor 3 (using the integer `IN_3`), then starts the conveyor belt (Motor A) and invokes the `kick_off` task. Hereafter it enters a loop where it indefinitely waits for a black brick to pass Sensor 1. The role of the boolean `b` is to keep track of which timer to reset (`Timer(1)` or `Timer(2)`). The role of the booleans `active1` and `active2` is to let the `kick_off` task know which timer to trigger on next. At each appearance of a black brick a timer is reset. The last wait-statement ensures that Sensor 1 will postpone looking for a new brick until the current one has passed completely. Without this wait statement the sensor might observe the same brick several times, and hereby lead the kick_off task into the misunderstanding about the number of bricks on the conveyor belt.

The `kick_off` control task is triggered by timers exceeding the value 25 (corresponding to 2.5 seconds), which is the estimated time for a black brick to move from Sensor 1 to the position of the kick arm. The additional tests on booleans `active1` and `active2` are necessary because the two timers are always running, and thus a timer value above 25 is not in itself enough information to determine that there is a black brick in front of the kick arm. In fact, without `active1` (or `active2`) the kick arm would be kicking back and forth in the air constantly even with no bricks on the belt. Now, `kick_off` will run Motor C for 0.6 seconds causing the kick arm to pass by the belt (hereby hopefully hitting a black brick). Then the direction is reversed and Motor C is run again until the kick arm hits Sensor 3 (a touch sensor). This use of a touch sensor ensures that the kick arm is brought to halt in exactly the same position for all kicks it performs. If alternatively, the arm was just stopped after some part of the kick, the arm might come out of adjustment after a number of kicks.

The timed automata models generated by our program `rcx2uppaal` from the two tasks `main` and `kick_off` are shown in Figure 16 and 17 of the Appendix[6].

[6]The figures are mainly intended to illustrate the size of the generated automata.

```
int b=0, active1=0, active2=0;
int DELAY=25;
int LIGHT_LEVEL=42;

task main{
  Sensor(IN_1, IN_LIGHT);
  Sensor(IN_3, IN_SWITCH);
  Fwd(OUT_A,1);
  start kick_off;
  while(true){
    wait(IN_1<=LIGHT_LEVEL);
    if(b==0){
      ClearTimer(1);
      active1=1;
    }
    if(b==1){
      ClearTimer(2);
      active2=1;
    }
    b=-b+1;
    wait(IN_1>LIGHT_LEVEL);
  }
}
```

Figure 11. The `main` control task, which is responsible for observing LEGO® bricks running on the belt and resetting appropriate timers in order for the `kick_off` task to know when to invoke the kick arm.

4.3 Environment: Bricks and Arms

The timed automaton in Figure 13 models a black LEGO® brick. This particular brick will start to pass by the light intensity sensor no later than 100.000 time units (corresponding to one second) after system initiation. This is done by setting the integer representing the light intensity, `IN_1`, to the value 35% which is precisely what happens when a black LEGO® brick start to pass by Sensor 1. The brick will remain in this location for 25.000 time units corresponding to a quarter of a second whereafter the light intensity is set back to 50%, this modeling the duration for the brick to pass the sensor. Next the black brick will let time pass until a total time of 250.000 time units, (2.5 seconds), since it was first noticed by the sensor has elapsed. Hereafter, the brick will during 20.000 time units (0.2 sec) accept a synchronization on action `kick` with the kick arm hereby being pushed off the belt. In case no synchronization is made the brick will time-out modeling that it has finally past the kick arm and eventually will fall off at the end of the belt.

All black bricks can be modeled as in Figure 13, the only difference being the starting interval modeled by the invari-

```
task kick_off{
  while(true){
    wait(Timer(1)>DELAY && active1==1);
    active1=0;
    Fwd(OUT_C,1);
    Sleep(6);
    Rev(OUT_C,1);
    wait(IN_3==1);
    Off(OUT_C);
    wait(Timer(2)>DELAY && active2==1);
    active2=0;
    Fwd(OUT_C,1);
    Sleep(6);
    Rev(OUT_C,1);
    wait(IN_3==1);
    Off(OUT_C);
  }
}
```

Figure 12. The `kick_off` task which will alternate between reacting on values of `Timer(1)` and `Timer(2)`. The presence of booleans `active1` and `active2` is necessary since the timers are running all the time, also when no bricks are on the belt.

ant in location `B0` and the guard on the transition to location `B1`. Also red bricks follow this skeleton, the only difference from black bricks being that the light intensity is set to a value larger than 42. The parameterized template facility in UPPAAL provides an easy way to create a number of black and red bricks. It is important to note that due to progression of time a brick will eventually reach one of the two locations **Kicked_off** and **Passed**. Which one of these two locations the brick reaches depends on it being kicked away or not. The properties to check in the verification presented in the next section will then be very simple: namely that black bricks will never reach the location **Passed** and that red bricks will never reach **Kicked_off**.

The timed automaton in Figure 14 models the physical kick arm. It will wait for three integer variables to be set to appropriate values after which it starts moving. After 6.000 time units it is willing to kick off a brick during 500 time units. Now, the arm will either reverse direction if told, or it will crash into the gate it is attached to (see Figure 9). It will be part of the verification to check that the kick arm under no circumstances will crash into the gate.

5 Verification

The purpose of the brick sorter is to have all black bricks (and only those) kicked off the belt. Thus, it should hold for a black brick that the location **Kicked_off** is reachable

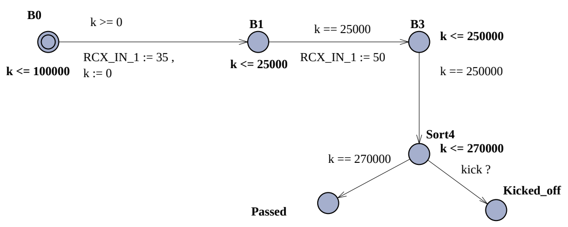

Figure 13. A timed automaton modeling a black brick. This brick will enter the area of Sensor 1 **no later than 100.000 time units (corresponding to one second) after the system initiation.**

and that the location **Passed** is not reachable. That is, for a black brick named BlackBrick1 the following properties should hold:

```
E<>( BlackBrick1.Kicked_off )
A[]not( BlackBrick1.Passed )
```

We have verified both these properties although our knowledge of the time progression in the brick timed automaton tells us that checking the last property would suffice. Dually, for the red brick we will have to verify the properties

```
E<>( RedBrick.Passed )
A[]not( RedBrick.Kicked_off )
```

As for the black brick, the satisfaction of this property together with the progression of time ensures that the brick will eventually reach the end of the belt.

5.1 Performance Results

We have analyzed several versions of the UPPAAL model of the brick sorter. The most simple scenarios are when just a single brick (either red or black) is placed on the conveyor belt. These scenarios were checked by UPPAAL within 10 seconds (and fortunately the answers were the expected ones as well!).

A more interesting scenario is when two black bricks may simultaneously be on the conveyor belt. We will assume that the first brick passes Sensor 1 at time 0. As for the the second brick we only make the assumption, that it will appear on the belt no later than 250.000 time units after the first brick[7]. Although UPPAAL is insensitive to large constants in clock guards the presence of the scheduler has a

[7] In case the second brick is put on the conveyor belt later than 250.000 time units after the first brick, the experiment would correspond to two one–brick–scenarios.

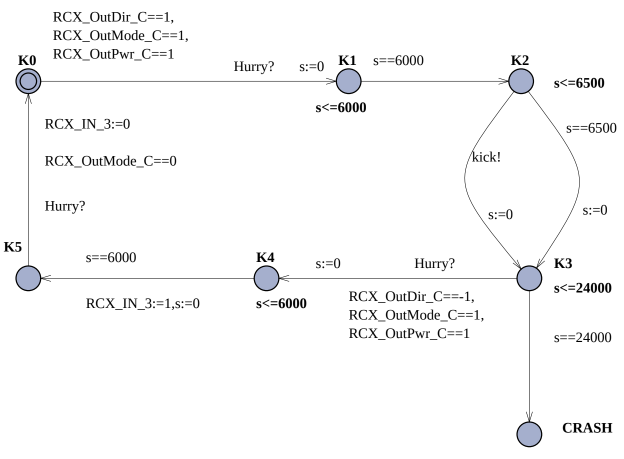

Figure 14. A timed automaton modeling the physical kick arm.

somewhat negative influence on the performance of the verification. The problem is that the scheduler chops the time into small tiny pieces of length 18 time units thus yielding a remarkable number of states that made verification impossible in practice. However, by limiting the entrance time for the second black brick to an interval of 50 time units verification can be done within a few seconds. Consequently, we chopped the time interval from 25.000 to 300.000 into small pieces of duration 50 time units and verified the system for all these during less than 10 hours.

Using the convex hull optimization in UPPAAL we have managed to reduce the verification time even further without chopping the large time interval. Although this technique gives an over approximation of the state space, this is still correct when considering safety properties. Now, only 1 hour and 10 minutes were needed to verify the system.

5.2 Finding an Error

As mentioned earlier the control program is developed to handle at most two black bricks on the conveyor belt at a time, and as such we have no reason to expect correct behavior in other scenarios. On the other hand, it may be of interest to figure out exactly what will happen if a third black brick is added on the belt, in a situation where two black bricks are already there. We therefore extended the UPPAAL model of the environment by a third black brick[8] and to our surprise the result of the verification was that the extra brick was indeed kicked off the belt, whereas the two first bricks were wrongly doomed to pass the kick arm without ever being hit by it. The diagnostic trace facility of UPPAAL was then used in the process of understanding this behavior, the reason being as follows. Brick 1, 2 and 3 set timers 1, 2 and 1, and thus the first brick is 'forgotten' by the control program since the third brick overwrites `Timer(1)`. Now, `Timer(2)` will time out first, and ideally the second brick ought to be kicked off the conveyor belt correctly. However, the `Kick_off` task (see Figure 12), is waiting for `Timer(1)` (connected to the third brick) to time out first and consequently the third brick (and only this one) is kicked off the belt. This erroneous behavior predicted by UPPAAL, matches the experimentally observed behavior.

[8]Again, the third black brick having a starting time less than 250.000 time units after the first brick.

6 Conclusion

In this paper, we have presented a method for automatic verification of real-time control programs running on LEGO® RCX™ bricks using the verification tool UPPAAL. The control programs — consisting of a fixed number of tasks running concurrently — are automatically translated into the timed automata model of UPPAAL. The fixed scheduling algorithm used by the LEGO® RCX™ processor is modeled in UPPAAL, and supply of similar (sufficient) timed automata models for the environment allows analysis of the overall real-time system using the tools of UPPAAL.

To illustrate our techniques we have constructed a machine for sorting LEGO® bricks by color. UPPAAL is successfully applied to prove that indeed the bricks are sorted correctly. Additionally UPPAAL succeds in pointing out an error in a setting where the model of the environment does *not* satisfy the assumptions made by the control program. In its present form, our methodology leads to disappointingly poor time-performance in the verification effort, even when an over-approximating analysis is performed. A plausible explanation of this phenomena is the direct modeling of the busy-waiting behavior of the RCX™ scheduler. In situations where all tasks are idling, this results in an extremely fine-grained, and seemingly unnecessary partitioning of the symbolic state-space. We are currently working on alternative models of the scheduler which will avoid this problem, and hence allow for our method to scale up.

Finally, we are convinced that our method is applicable to real-time control programs and systems in general and not only to RCX™ programs and LEGO® MINDSTORMS™ systems.

References

[1] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. *Theoretical Computer Science*, 126(2):183–236, April 1994.

[2] David Baum. *NQC — Not Quite C.* Web site at `http://www.enteract.com/~dbaum/nqc/`.

[3] Lars Björnfot. Ada and Timed Automata. In *Proc. of Ada in Europe*, number 1031 in Lecture Notes in Computer Science, pages 389–405. Springer–Verlag, 1995.

[4] James C. Corbett. Modeling and Analysis of Real–Time Ada Tasking Programs. In *Proceedings of the Real-Time Systems Symposium*, IEEE Computer Society Press, December 1994.

[5] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: The Next Generation. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 56–65. IEEE Computer Society Press, December 1995.

[6] Thomas Hune. Modelling a real-time language. In *Proc. of the 4*th *Workshop on Formal Methods for Industrial Critical Systems*, 1999.

[7] Thomas Hune, Kim G. Larsen, and Paul Pettersson. Guided Synthesis of Control Programs Using UPPAAL. Accepted for publication at DSVV'00, 2000.

[8] Torsten K. Iversen and Cris. B. Thomasen. Automatic Verification of LEGO RCX Systems Usin UPPAAL. Technical report, Institute of Computer Science, Aalborg University, 1999.

[9] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[10] Morten Laursen, Rune G. Madsen, and Steffen K. Mortensen. Verifying Distributed LEGO RCX Programs Using UPPAAL. Technical report, Institute of Computer Science, Aalborg University, 1999.

[11] LEGO. *Software developers kit*, November 1998. See `http://www.legomindstorms.com/`.

Appendix

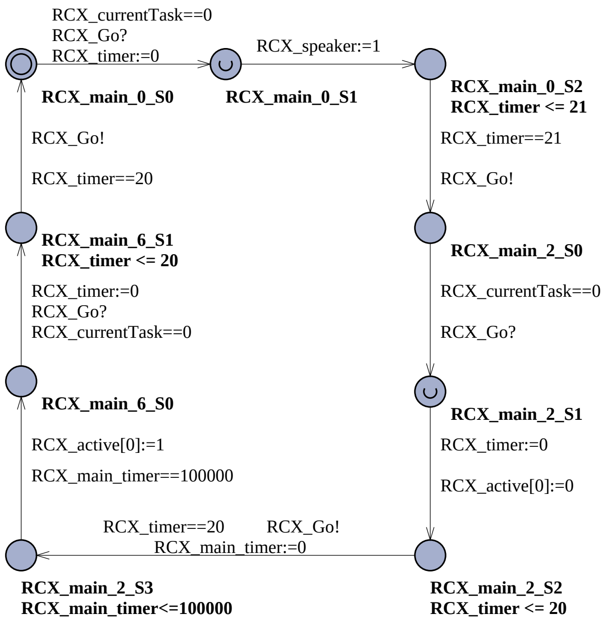

Figure 15. The timed automaton model generated by `rcx2uppaal` for the programs shown in Figure 2 and 3.

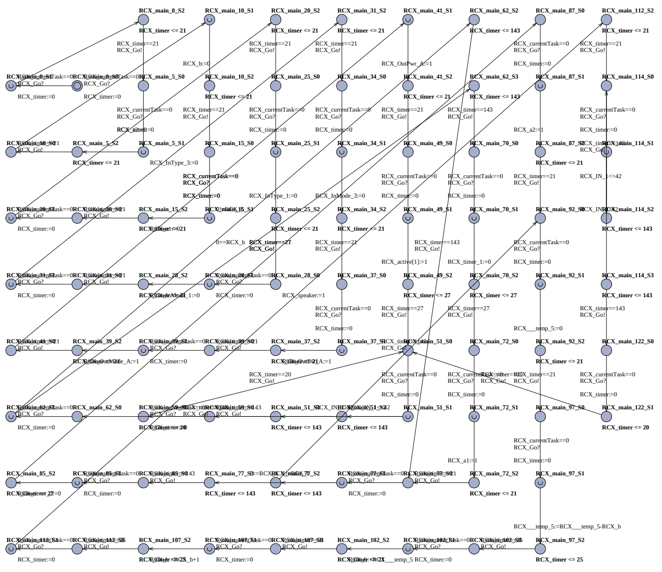

Figure 16. The timed automaton model of the `main` task shown in Figure 11.

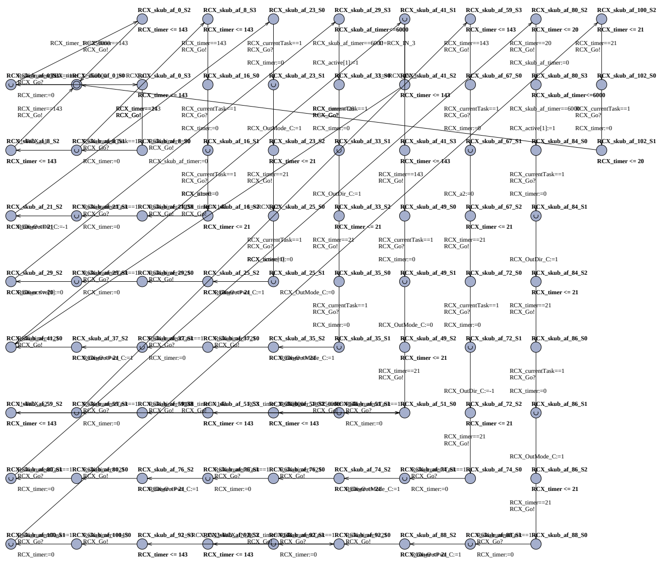

Figure 17. The timed automaton model of the the `kick_off` task in shown Figure 12.