Basic Research in Computer Science

BRICS RS-00-17 T. S. Hune: Modeling a Language for Embedded Systems in Timed Automata

Modeling a Language for Embedded Systems in Timed Automata

Thomas S. Hune

BRICS Report Series RS-00-17

ISSN 0909-0878 August 2000

Copyright © 2000, Thomas S. Hune.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

http://www.brics.dk
ftp://ftp.brics.dk
This document in subdirectory RS/00/17/

Modeling a language for embedded systems in timed automata*

Thomas Hune

BRICS**, Department of Computer Science
University of Aarhus, Denmark
baris@brics.dk

Abstract. We present a compositional method for translating real-time programs into networks of timed automata. Programs are written in an assembly like real-time language and translated into models supported by the tool Uppaal. We have implemented the translation and give an example of its application on a simple control program for a car. Some properties of the behavior of the control program are verified using the generated model.

1 Introduction

Reasoning about real-time systems can be very difficult and even more so if they consist of several concurrent processes. Tools for formal reasoning about such systems have been successfully developed [LPY97,HHWT95,Yov97] and applied with in a number of cases (see [LPY97,HHWT95,Yov97] for lists of case studies). Before applying such tools one has to define an appropriate model of the system in question. This can in many case be a time consuming and error prone process. Methods and tools for defining such models based on an (informal) description of the system or parts of the system are an important help in the process of modeling.

One can divide models of embedded system into two groups. The first consisting of systems where the control program (if any) and the physical systems are mixed into one description (the water level monitor and the leaking gas burner, see e.g. [ACH+95], are examples of this). Systems in the second group have a clear distinction between

* This work is partially supported by the European Community Esprit-LTR Project 26270 VHS (Verification of Hybrid systems)

** Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

the control program and the hardware/environment (some versions of the train gate controller, e.g. the one in [Hen96], belong to this group). Here we will consider systems belonging to the second group. More precisely we will consider a method for modeling the control programs of such systems.

We have defined and implemented a translation from control programs written in the RCX™ language to networks of timed automata [LPY95] used by Uppaal [LPY97]. Such a translation allows easier access to the verification power of a tool like Uppaal since the model of the program comes for free. The implementation has been tested on a number of programs and the properties of these programs have been verified by Uppaal.

The programs we are considering, are written in a language called the RCX™ language, which is an assembly like language with some highlevel features. The RCX™ language runs on a processor in the LEGO® RCX™ brick which is part of LEGO® MINDSTORMS™ and LEGO® ROBOLAB™. The RCX™ brick is basically a (big) LEGO® brick with a computer inside. The brick has three input and three output ports, a speaker and an infrared sender and receiver for communication. Four different types of sensors are available for the RCX™ brick: touch, light, temperature, and rotation. Programming in the RCX™ language takes place on a PC where the programs are translated into byte code and downloaded to the RCX™ for execution.

In Section 2 the RCX™ language is described in more detail and an example of a control program for a car is given. Section 3 describes how RCX™ programs are executed especially with respect to scheduling. The translation is described in Section 4. The correctness of the translation is shortly discussed in Section 5 and and some aspects of the implementation in Section 6. In Section 7 the example is revisited and Section 8 contains a conclusion and ideas for future work.

2 The RCX™ language

The language we are considering here is the RCX™ language running on the LEGO® RCX™ brick. It is a kind of assembly language but with some features from high level languages. The language is

mainly used as a target language for compilers form other languages like the ones in MINDSTORMS™ and ROBOLAB™. We have chosen to look at the RCX™ language for several reasons. First of all, it is a fairly simple language, but with most standard assembler operations. However, there is only one addressing mode making modeling a lot simpler. Programs consists of a fixed number of tasks running concurrently with a simple scheduling algorithm (see Section 3).

Even though the language is simple it can be used to write interesting control programs. Since the RCX™ is part of LEGO® one can build physical embedded systems for the control programs. This gives the opportunity of conducting experiments with complete embedded systems, and to study the relationship between the behavior of the complete embedded system and the formal model.

For these reasons we believe that the RCX™ language is suitable for a first try of defining an automatic translation from control programs to formal models.

2.1 Program structure

A RCX™ program consists of a number of tasks. The number is restricted to a maximum of ten tasks, numbered 0 to 9. There are other restrictions imposed by the language, the most sever one being that one can only use 32 integer variables for data (it is not possible to address more). The body of a task is defined between `BeginOfTask(i)` and `EndOfTask()`. During execution a task is either blocked or enabled, initially only task 0 is enabled. A task can be started by the command `StartTask(i)` and blocked by `StopTask(i)`. Whenever `StartTask(i)` is executed, task i is restarted from the beginning independent of the state of the task, so there is always at most one copy of each task.

Next we will give a small example of a control program for a car, and following that informally[1] present the part of the language we have considered. We present the instructions of the language in two groups, one containing instructions for control of flow and one for commands.

[1] No formal semantics is available for the language. An informal description of the language can be found in [LEG98]

2.2 Example

As an example we will look at a simple control program for a car equipped with one touch sensor on each side of the front and one motor on each side driving one wheel. At the front and the back there is a ball instead of a wheel to make turning smoother. The car is turned by the motors running in different directions. Figure 1 is a sketch of the car. The control program consists of three tasks. Task 0

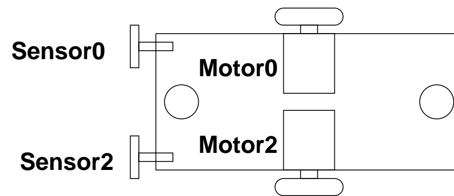

Fig. 1. Sketch of the car.

first sets up the sensors and starts the car. After having started the car the task enters an infinite loop waiting for the reading from one of the sensors to change from zero to one. The change of a sensor reading from 0 to 1 will be considered an event which should be handled. Depending on which of the two sensors changes, a task is started to handle the event.

```
BeginOfTask(0)
  SetFwd("02")             # Setup the output ports(motors)
  SetPower("02",2,4)       # to forward and power 4
  SetSensorType(0,1)       # Setup the input ports
  SetSensorMode(0,1,0)     # to touch sensor,
  SetSensorType(2,1)       # boolean mode
  SetSensorMode(2,1,0)
  SetVar(0,2,0)            # Var0:=0 (oldSensor0)
  SetVar(2,2,0)            # Var2:=0 (oldSensor2)
  On("02")                 # Start the motors
  Loop(2,0)                # Begin the infinite loop
    SetVar(3,9,0)          # var3:=Sensor0
    If(0,3,2,2,1)          # If var3 = 1
      If(0,0,3,2,1)        # If var0 <> 1
```

```
        StartTask(1)      # Start task 1
        SetVar(0,0,3)     # Var0:=Var3
      EndIf()
    Else                  # Sensor0 was 0
      SetVar(0,0,3)       # Var0:=Var3
    EndIf()
    SetVar(4,9,2)         # Var4:=Sensor2
    If(0,4,2,2,1)         # If var4 = 1
      If(0,2,3,2,1)       # If var2 <> 1
        StartTask(2)      # Start task 2
        SetVar(2,0,4)     # Var2:=Var4
      EndIf()
    Else                  # Sensor2 was 0
      SetVar(2,0,4)       # Var2:=Var4
    EndIf()
  EndLoop()
EndOfTask(0)
```

Task 1 and 2 are supposed to back the car a little and then turn away from the obstacle. The only difference between the tasks is the direction in which the car is turned, so only task 1 is shown.

```
BeginOfTask(1)
  Off("02")         # Stop the motors
  SetRwd("02")      # Set the motors to go backwards
  On("02")          # Start the motors
  Wait(2,40)        # Wait while the car goes backwards
  SetFwd("0")       # Make the car turn
  Wait(2,30)        # Wait while it is turning
  SetFwd("2")       # Go forward again
EndOfTask()
```

In line six of task 2 the argument for `SetFwd` is `‘‘2’’` and in line eight it is `‘‘0’’`.

2.3 Commands

The commands of the language can be divided into three categories. There are commands for manipulating variables, for setting up the sensors, and for controlling output.

The command for assignment is `SetVar(i,j,k)` where the number of the target variable (there are no symbolic names) is $i \in \{0, 1, \ldots, 31\}$, j is the type of the source of the assignment and k is the source. The most used sources (and the only ones we will consider), are variables (where k is the number of the variable), constants (where k is the value), sensor readings (where $k \in \{0, 1, 2\}$ is the number of the input port), and messages on the communication port[2] (where k does not have a meaning). All the commands for manipulating variables have this form, but the sensor reading and the message can only be used in assignment. The other possible manipulations are addition, subtraction, multiplication, division, bitwise conjunction and bitwise disjunction.

Two commands for setting up the type of the sensors exists. `SetSensorType(i,j)` specifies that input port i should be read as a sensor of type j, j being one of the four types described previously. One can also set the type of readings from a sensor by the `SetSensorMode(i,j,k)` command. Again i specifies which input port is set, j is the type of input, e.g. a raw ten bit integer value, a boolean or a percentage value. If boolean is chosen, k specifies how the value is calculated.

There are two types of output ports. Three ports for motors or lights (since we will concentrate on these, they will be called output ports) and one for the speaker. The most basic commands for controlling the output ports are `On(li)` and `Off(li)` where li is a list of output ports. These simultaneously turn on respectively turn off the ports specified by li. The commands `SetFwd(li)`, `SetRwd(li)` and `AlterDir(li)` can be used for changing the 'direction' of the output of the ports specified by li. Finally, one can change the power of the output from the ports by `SetPower(li,j,k)` where li specifies the ports and j and k specifies the power in the same way as j and k specifies a value in the commands for manipulating variables. The value for the power can only be chosen in the range $0, 1, \ldots, 8$. For using the speaker port the two commands, `PlaySystemSound(i)`, and `PlayTone(i,j)` are available. The first plays one of six predefined sounds or beeps, and the second plays a tone with frequency i for duration j, given in units of ten milliseconds.

[2] This will always be the last message received.

2.4 Flow control

Two kinds of iterations exist in the language. `Loop(j,k)` loops the number of times specified by j and k, where j is the type of the source and k is the source. As source only variables and constants can be used. If j and k specifies the constant zero, the loop is infinite. The other possibility is `While(i,j,k,l,m)` where k specifies a comparison operator from the set $\{<, >, =, \neq\}$, and i,j and l,m specifies the values to be compared. Here variables, constants and sensor readings can be used.

An *if* statement with an optional else part, `If(i,j,k,l,m)` having the same type of arguments as the *while* statement is also available.

Finally, one can block a task using the `Wait(j,k)` command. Here j and k specifies the time for which the task will be blocked in ten milliseconds units. This is the only 'real-time' command in the sense that it refers directly to time.

3 Execution and scheduling

The RCX™ is running a small operating system with processes for handling I/O and *one* process running an RCX™ interpreter. The process running the interpreter has the lowest priority. Since (almost) all the handling of I/O is periodic we assume that the interpreter gets a fixed portion of the CPU time.

Initially all tasks but task 0 are blocked. Each enabled task executes *one* instruction and then leaves control to the next task in a round robin fashion. One could imagine a number of other scheduling policies for RCX™ and experimenting with this would be interesting.

A task context switch takes place between interpretation of instructions. We have made some experiments measuring the timing of the execution of programs on the RCX™. Based on these we have concluded that the number of context switches does not depend on the number of tasks in a program. In a program with only one task, an instruction is interpreted approximately every 2 milliseconds.

The scheduling policy is very important when reasoning about the behavior of programs. If we want to guarantee a given response time of some action like pressing a touch sensor we must first of

all know how often the control program reads the input from the sensor[3]. As one can imagine, the response time will depend on the number of enabled tasks making it difficult to give precise bounds, though upper bounds can be given. We will return to this question in Section 7.

4 Modeling

The type of model we are using to model programs is networks of timed automata extended with integer variables which is supported by the tool Uppaal. Uppaal implements is a real-time model which we find appropriate for our purpose. The only real-time feature in the language (the wait command) could be handled by introducing a kind of tick, but we find it more natural to introduce a clock variable to model time. More importantly, we aim at modeling more than control programs. We hope to use our models of control programs together a with a model of the environment they are controlling and time will be needed for describing this. We might need more general constructions than clock variables to get a satisfactory model of the environment. However, the tools we know of supporting more general models does not seem to be mature enough yet.

A network of automata is a collection of automata running in parallel and communicating by handshake. Channels are either internal, input, or output. Internal channels do not have a name, while the other channels do. The names of input and output channels are suffixed with '?' or '!' respectively. Communication takes place between one output channel and one input channel. Internal channels do not synchronize. Figure 2 is an example of a network of two automata. Here the right automaton can communicate with the left on the a channel. After communicating both automata are in state S1 where the right automaton can send on channel c and the left on channel b. If a communication on the b channel occurs nothing more can happen, but if the c channel is used for communication the right automaton has the possibility to move back to state S1, using the internal channel. Here it can input on the b channel and

[3] This of course also depends on how often the underlying operating system polls the input ports but since we do not know this, we assume this is done 'often enough'.

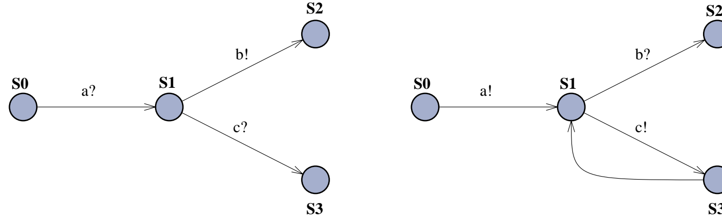

Fig. 2. A network of two automata

output on the c channel, but in this system, there is no automaton to communicate with so the system is deadlocked.

Time is added in the standard way [AD90,AD94] by introducing a number of real valued clock variables. In Figure 3 we have added clock variables x and y to the automata. Initially both clock variables

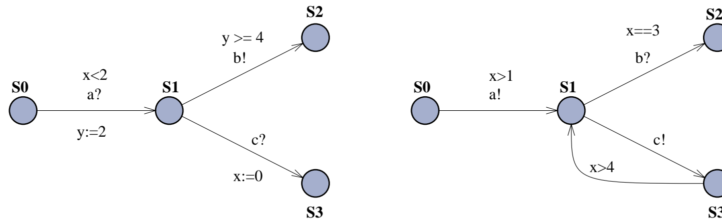

Fig. 3. A network of two timed automata

have value zero and they progress at the same speed during the execution. The a channel is enabled in the left automaton until two time units have passed and in the right after one time unit has passed from the beginning of the execution. After the communication has taken place the value of the clock variable y is set to two. This should give an idea how one can specify, when a channel is enabled by adding guards to the channel and how one can reset the value of clock variables. Only integer values are allowed in guards and resettings.

In Uppaal one also has integer variables which do not change with time but can be part of guards and assignments. On the right hand

side of assignments, expressions of the form $E ::= Var|Num|E \text{ OP } E$ where $\text{OP} \in \{+, -, *, /\}$ can be used. We will make heavy use of this, when modeling operations on variables in the language.

4.1 Structure of model

In our model of a program we will make one automaton for each task. All channel names are suffixed with the name of the task (we will assume that all instructions are from task 0 in the following figures if nothing else is stated). This makes it possible for the scheduler, also represented by one automaton, to control which task is allowed to execute. All information concerning time spent on interpreting instructions is handled by the scheduler.

Before we look at how single instructions are modeled, we have to look at the modeling of I/O. Since we are modeling a program and not a complete system our knowledge about the surroundings is very limited. The program gets readings from sensors after A/D conversion and some preprocessing but it cannot relate these readings to the true values in the environment[4]. Therefore the reading from a sensor is represented by an integer variable, named *Sensori*, where i is the number of the sensor. This is all the program has knowledge of. For output we do something similar. The status of each output port is modeled by three integer variables *Motor_Pow*, *Motor_Dir* and *Motor_On* representing the power, the direction and whether the port is turned on or not, respectively. The effect this has on what is connected to the port is not modeled. For the port to the speaker there is also three integer variables *SystemSound*, *Frequency*, and *Duration*.

In the following subsections we will describe the transformation, which is compositional, and hopefully it should be clear that implementing this has been straightforward. In all the figures the state named S0 is either the last state of the model of the previous instruction or the initial state of the automaton, if the instruction is the first of the task.

[4] Such readings might not make sense at all. If the programmer has specified that sensor 1 is a touch sensor and someone plugs a light sensor to port 1, how should the program relate such readings to the real values?

4.2 Commands

All the commands are modeled in a similar way by one channel and one new state. The name of the channel is the name of the command. The commands for setting up the sensors do not update any integer variables since the type and mode of the sensors are not used in the models we have defined so far. If needed, this may be included later. For a command like `On(''02'')` the variables *Motor_On0* and *Motor_On2* are set to 1 as shown in Figure 4. The other commands

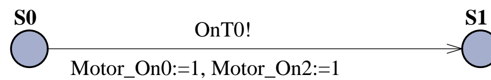

Fig. 4. The model for the single command `On(''02'')`.

for controlling the output ports are modeled similarly. Also the sound commands for the speakers are handled in this way.

The commands handling variables are modeled in almost the same way though here the type of the argument must be determined. If the command is `SetVar(23,2,0)` we should assign the constant 0 to variable number 23 (remember that the second argument specifies the type of the third argument, 2 means a constant). The command `SumVar(23,0,21)` for adding variable number 21 to variable number 23 will have the assignment $Var23 := Var23 + Var21$.

If we look at the sequence of commands like the first three commands from task 1 of the example we get the automaton in Figure 5. There is one channel for each command labelled with the name of the command. Each channel updates the value of the variables specified by the arguments of the command.

4.3 Flow control

When translating the instructions for flow control we need more that one new channel and one new state. In the model of some of these instructions there must be room for connecting models of the inner parts of the instruction (like the body of a loop). For modeling a *loop* statement like

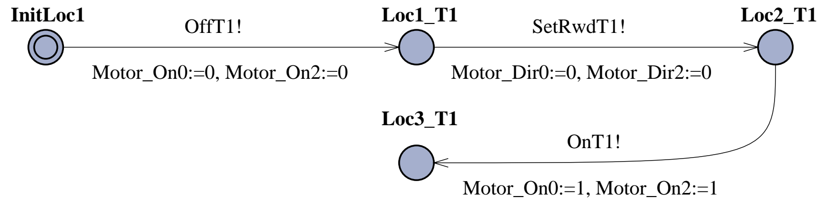

Fig. 5. The model of the first three commands of task 1.

```
Loop(2,5)
    .
    .
EndLoop()
```

where .. is the body of the loop, we use four new states. Figure 6 shows the model of such a loop statement. In the example we have a

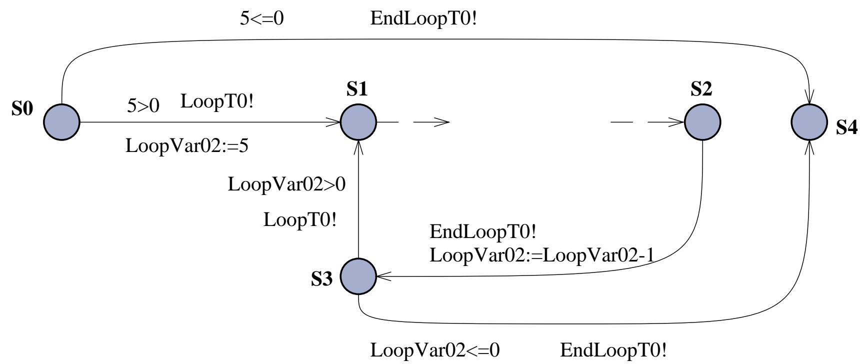

Fig. 6. The model for a *loop* statement.

loop always running five times but the number of times can also be specified by a variable. In the model this would mean changing the constant 5 to the specified variable. The states S1 and S2 are the initial and final state of the model of the body respectively. From the state S0 we have two channels, one for entering the loop and one for leaving it. Only one of these is enabled at a time. If the loop is

entered we assign the number of times the loop must be executed to a new loop variable. Every time the end of the body is reached, the loop variable is decremented and state S3 is entered. From S3 there is a channel leading to the beginning of the body and one to S4. Again only one of the channels is enabled based on whether the loop is finished or not. When the loop is finished state S4 is entered. Decrementing the loop variable and testing it could of course have been done in one step but in the implementation of the interpreter this is interpreted as two instructions. This means that the task needs to be scheduled twice to restart a loop.

The model of a *while* statement is very similar to that of a loop though there is no need for a new variable. An example of a *while* statement is

```
While(0,3,2,2,0)
    .
    .
EndWhile()
```

where the arguments means `While Var3==0`. Again .. is the body of the while. In Figure 7 the model of the *while* statement can be seen. As before the state S1 is the initial state of the body and S2

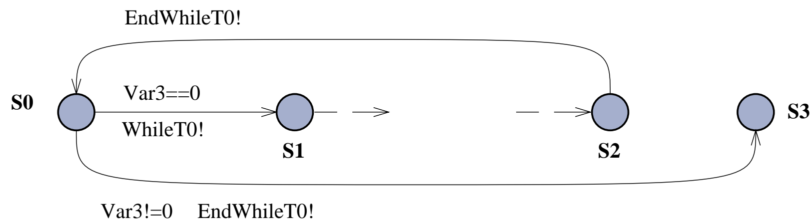

Fig. 7. The model for a *while* statement.

is the final state. From the state S0 we can move to the initial state of the body if the condition of the while is satisfied and otherwise the channel to the state S4 is enabled. Only one of these channels is enabled. From the final state of the body there is a channel to the S0 state where the condition is tested again. As for the *loop* this could have been done in one step (one channel) but the interpreter uses two steps.

An *if* statement (without an else) like

```
If(0,3,2,2,0)
    .
    .
EndIf()
```

is modeled as in Figure 8. Again the initial state of the body is S1

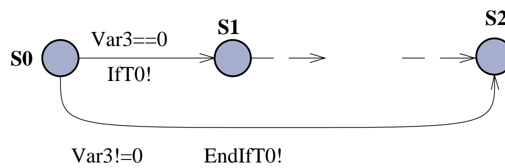

Fig. 8. The model for an *if* statement.

and the final state of the body is S2. From the S0 state a channel to S1 is enabled if the condition is satisfied. If this is not the case, a channel to state S2 is enabled.

The model of an *if-else* statement follows the same idea, though now there are two parts or bodies. A model of the *if-else* statement

```
If(0,3,2,2,0)
    .
    .
Else()
    .
    .
EndIf()
```

is shown in Figure 9. As in the model of the *if* statement there are two channels from the S0 state. One with label *IfT0* which is enabled if the condition is satisfied and one labelled *ElseT0* which is enabled otherwise. The initial state of the model of the *if* part (the first . .) is S1 and the final state is S2. From here there is an *EndIf* channel to S4 which is the final state of the model for an *if-else* statement. The initial state of the model of the *else* part (the second set of . .) is S3 and the final state is S4.

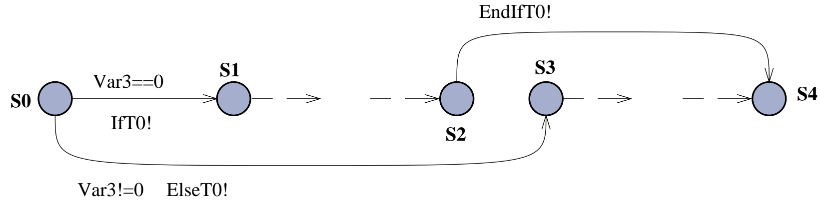

Fig. 9. The model for an *if-else* statement.

The *wait* statement is a little different from the others. This is the only instruction in the language referring directly to time and also the only part of the model of a task referring to time (represented by a clock). The model of a wait instruction, see Figure 10, consists of a channel from the S0 state to state S1 where a clock variable *xT0* is reset to zero, and a channel labelled *SkipT0* from S1 to itself. The *SkipT0* channel is only used to synchronize with the scheduler

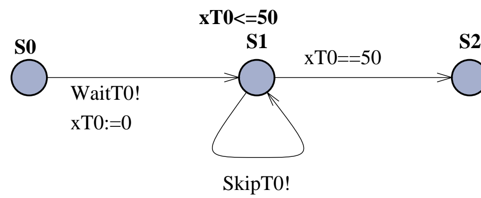

Fig. 10. The model for a *wait* statement.

without any time passing. There is also a channel without label from S1 to S2. With this construction the scheduler does not need to keep a list of tasks blocked by wait. The state S1 has an invariant forcing the automaton to leave the state when the task is no longer blocked by the wait. When the task is no longer blocked the channel to state S2 is enabled. The channel from S1 to S2 is not enabled when the task is blocked.

4.4 The scheduler

Applying the translation described so far we get an automaton for each task modeling the execution of that task. To get a model of

the execution of the complete program we must combine the executions of the individual automata according to the scheduling policy described in Section 3. We define one automaton controlling the execution of the other automata (by synchronizing with these) implementing the scheduling policy on the RCX.

When the `StartTask(i)` command is executed task number i is restarted. Therefore we need a way of getting to the initial state of a task from all the other states in the task. For this purpose we add a channel labelled $RSTi$ from all the states (including the initial state) to the initial state. The scheduler also needs to realize when a task has finished its execution, and hence we add a channel from the last state to itself with label $FinTi$.

As mentioned the scheduler lets each task which is not blocked execute one instruction in round robin. A task can be blocked if it has not been started (initially only task 0 is started), if it has finished, because of a *StopTask* statement, or because of a wait statement. In the first three cases our model of the scheduler will skip the task but in the case of the wait, the *Skip* channel with no delay will be used for communication. This means that the scheduler does not need to manage a list of tasks blocked by wait.

Our timing experiments suggest that the time spent on interpreting the different commands is almost the same for all commands. Since the time is almost independent on the parameters for the command we will not take this into account when modeling. The time measured for interpreting an instruction is less that 0.2 milliseconds which is less than the time spend on the context switch. In the model we have chosen milliseconds as our basic time unit. Therefore we will not model the time spent on interpreting each command but say that interpreting one command from each task in the program takes one millisecond, no matter how many tasks are enabled. The overhead of instructions for all the tasks is two milliseconds, so interpreting one instruction for all the tasks takes three milliseconds including context sitches. This limits the precision of our model since we are using three milliseconds steps. So the best guarantees we can give using the model is within three milliseconds.

Figure 11 shows the structure of the scheduler for a program with three tasks, where only task 0 can start the other tasks. The initial state in the figure is the one with a ring inside. In Uppaal one can de-

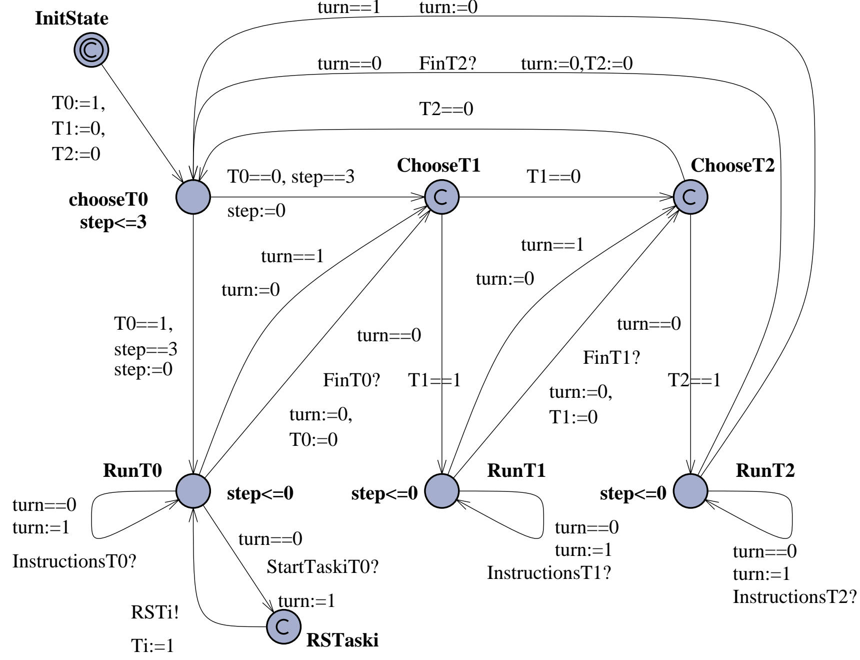

Fig. 11. Model of a scheduler.

fine a state to be *committed* which means that the automaton must leave the state before any other action takes place. Especially, this means that time cannot pass while an automaton is in a committed state. Committed states are marked by a 'C' inside state. The committed states in the scheduler can be seen as a kind of control states used for deciding who should be allowed to execute next.

For each task i there is an integer variable Ti taking values 0 or 1. If the task is enabled the value is 1, otherwise it is 0. The channel from the initial state to *ChooseT0* initializes these variables such that only task 0 is enabled. When the scheduler is in state *ChooseTi* the next task to execute according to the round robin schedule is task i. If this task is enabled the scheduler can move to state *RunTi* where it is possible to execute one instruction from the task. In case the task is blocked the scheduler can move to test the next task. Since all the *ChooseTi* states (except *ChooseT0*) are committed this does not take any time.

In state *RunTi* the next instruction of task i can be executed. The channel labelled *InstructionsTi* from *RunTi* to itself represents a number of channels, one for each kind of instruction in task i. All these channels have the same guard and assignment but different labels. The clock *step* is used to synchronize the execution such that each round through all the tasks takes three milliseconds. In state *ChooseT0* there is an invariant $step \leq 3$ and guards on the outgoing channels such that exactly three milliseconds must pass in this state. In the *RunTi* state there is an invariant $step \leq 0$ making sure that time does not pass in these states. We have chosen not to make these states committed because the environment should have the possibility of doing something. To make sure that a task only executes one instruction each time it gets the control, the integer variable *turn* is used. Depending on the value of *turn* it is possible to execute the next instruction of the task or leave control for the next task. Changing the model slightly would allow for different instructions to take a different amount of time. More instructions could be allowed by changing the guards involving *turn*.

The *StartTaskiT0* instructions is treated specially. This instruction executes as the other instructions but it must also restart task i. Therefore the *StartTaskiT0* channel ends in an intermediate state *RSTaski* which is committed. From this state there is a channel restarting task i and setting the variable *Ti* to one. After this control is back in the *RunT0* state, as if a normal instruction had been executed.

The scheduling policy on the RCX™ is very simple but with our approach one can model more complicated policies. As mentioned the number of instructions executed by each task could easily be changed. One could also define a time slice instead of counting instructions. Modeling a scheduler with fixed priorities is also possible. After a task finishes the scheduler should move control to the state allowing the task with the highest priority to start. If this task is enabled it will start, otherwise the other tasks should be checked according to their priority until one can be started.

4.5 I/O

Handling of the I/O is not part of the model. We have modeled each output port by three integer variables and the speaker port by three integer variables as well. How fast the motor goes or what sounds the speaker plays will not be part of our model, and the effects this might have on the environment and thereby the inputs, will not be modeled either. Note that this is important for modeling a complete system, but based only on the program, we cannot hope to do this automatically.

The input is not modeled either, though generating a simple model of this letting the sensor reading behave arbitrarily would be easy. Based on the sensor mode one can define an automaton which at any time can assign any value in the range to the integer variable representing the sensor reading. In most cases this can be determined by a static analysis of the program. Programs can change the sensor mode dynamically making such an analysis more difficult to realize automatically.

We have defined these automata by hand for the experiments we have made. In the example there are two touch sensors which can give the reading zero or one. The simplest way of representing the behavior of these two sensors is shown in Figure 12. This does not

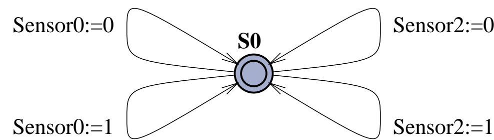

Fig. 12. Model of simple environment for two touch sensors.

place any restrictions on the readings from the sensors. If we had knowledge of how often the underlying operating system is polling the sensors, this would be a natural constraint to put on the channels.

5 Correctness

When addressing the correctness of the translation we must consider two different aspects. The relationship between the input of a program (values on the input ports) and the program variables and values written to the output ports. In our case this also depends on the scheduling of the processes. The second aspect is the timing information of an execution and in the model.

No formal semantics of the RCX™ language is available but for most of the commands the semantics is clear from the informal description given in [LEG98]. The few points which were not clear from the description has been clarified by some simple experiments with the language.

One could give a formal operational semantics to a task describing the output and changes of program variables with respect to old program variables, old output, and input. Assuming functions Var, In, and Out representing these environments, rules would have the form

$$(\mathsf{Var}, \mathsf{In}, \mathsf{Out}) \xrightarrow{\texttt{SetVar(3,0,5)}} (\mathsf{Var}[3 \mapsto \mathsf{Var}(5)], \mathsf{In}, \mathsf{Out})$$

With rules like this for all the instructions of the language, one can for each instruction in the language prove that the translation described in Section 4 satisfies these rules. Given a rule for sequential composition it would be possible to prove correctness of the individual tasks with respect to the semantic given. Properties of a program containing more than one task must be based on semantic rules taking the scheduling into account. If we disregard the `Wait` instruction this could also be done without complicated modifications. If we also want rules for the `Wait` instruction some notion of time is needed in the semantic. Relating such a notion of time in the semantic to the notion of time in Uppaal would be much harder than relating values of variables.

Proving the translation of the individual tasks correct should therefore be simple but tedious. Proving the scheduling of tasks without the wait should also be possible though more challenging.

With respect to the timing of the execution it might be more appropriate to talk about accuracy than correctness. Should we talk

about correctness we would need precise information about the operating systems and the how much time is spend on handling I/O. This might in the end depend on the input to the sensors. Also the precision of the clock would have to be taken into consideration. For these reasons we will talk about accuracy of the timing information. There are two immediate problems with the timing information in the model. First of all, the number of tasks enabled is not taken into consideration when calculating the time spend interpreting the commands. Secondly, time only passes in one state. In solving the second problem we would have to solve the first as well. To solve this the time unit would have to be changed, but this would enable models with better precision in general. With the current information we have on the timing of the execution of instructions, it does not make much sense to to allow for specification with such a precision. Our assumption about all commands taking the same time might not be valid with such a fine grained measure of time. Much more precise timing information would be needed for models and results obtained from these models to be useful.

Therefore we have chosen to keep milliseconds as the basic time unit and have the three milliseconds intervals when modeling the execution. With this as basic time unit, we find that there is a good correspondence between the timing specified in the model of a program and the actual execution of the program. However, since there is a small inaccuracy this can be added up during long executions. One should of course be aware of this when modeling and proving properties.

6 Implementation

From the description in Section 4 it should be clear that the translation can be implemented. We have made an implementation in ML which translates a RCX™ program file to a file containing a textual description of a network of timed automata (called *xta* format). The *xta* format is the format Uppaal uses for describing automata. There is no graphical information in the *xta* format but the newest version of the graphical interface to Uppaal can read a file in *xta* format and display the corresponding network of timed automata.

The program works in two phases. First the program file is parsed and a data type for the program is built. This type looks as one would expect with a statement being one of the instructions described in Section 2 and the body of the control statements consisting of statements.

The second phase is a recursive descent of the data type for the program. Since our translation is compositional a statement can be translated only knowing the last state of the model of the previous statement.

Along the way through the data type one also needs to collect the names of channels, clock variables, integer variables and states since these must be defined in the *xta* file.

We have successfully tested the translation program on a number of RCX™ programs.

7 Example revisited

We have used the translator to get a model of the control program in Section 2.2. Figure 13 shows the automaton for task 0. The loop testing the input from the sensors begins at state *Loc9_T0*, the channels before this state models the initialization. The state *Loc23_T0* is the final state which is not reachable since the loop is infinite. There are no restart channels in this model (except the one from the initial state) because this task is never restarted.

The models of task 1 and 2 are very similar so we will only show the model of task 1 (Figure 14). If one abstracts away from the restart channels (the channels labelled *RST1?*) it should be easy to follow the one path through the model. It should also be easy to see that this models the commands in the task.

We will not show the model of the scheduler since this looks very much like the model in Figure 11. The only difference being that the *InstructionsTi* channel has been replaced with a number of channels - one for each type of instruction in the task.

If we model the input from the sensors by the automaton in Figure 12 and use this together with the automata for the tasks and the scheduler we have a model of the complete system. We cannot use this model to reason directly about the movements of the car. What we can do, is reason about how the output ports react to

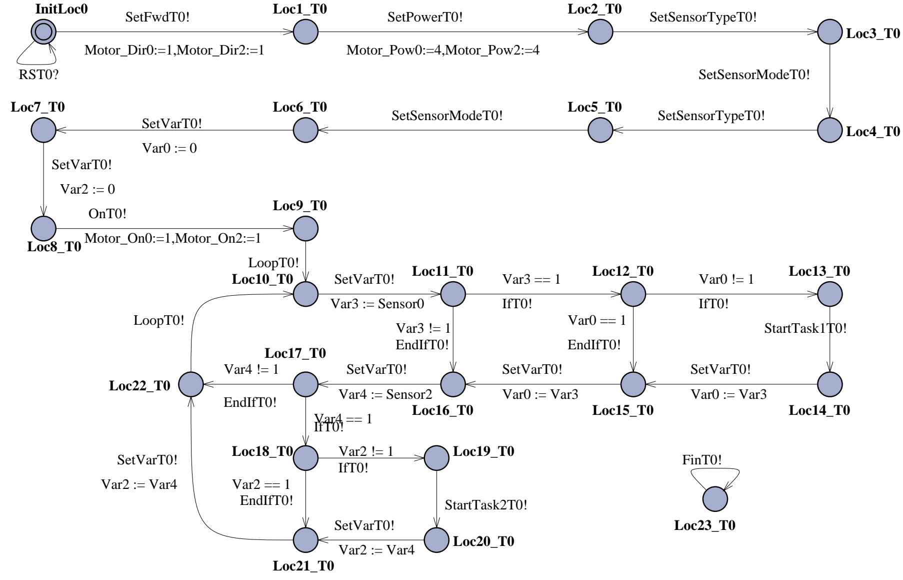

Fig. 13. Model of task 0.

input from the sensors. With this model we can first of all simulate the behavior of the program. Given the very liberal model of the environment we have defined, we can test how our program reacts under any possible kind of input.

We can also answer some very basic questions about the program such as whether it is possible for the output ports to be turned on or whether it is possible for the output ports to be in reverse. This is done by checking the formulas

$$E <> (Motor_On0 == 1 \ and \ Motor_On2 == 1)$$

and

$$E <> (Motor_On0 == 1 \ and \ Motor_On2 == 1 \ and \\ Motor_Dir0 == 0 \ and \ Motor_Dir2 == 0)$$

respectively. Both properties are satisfied and we get a trace leading to a satisfying state.

We can also try to find out how fast the program will respond when an input is read. This can not be done directly by writing one

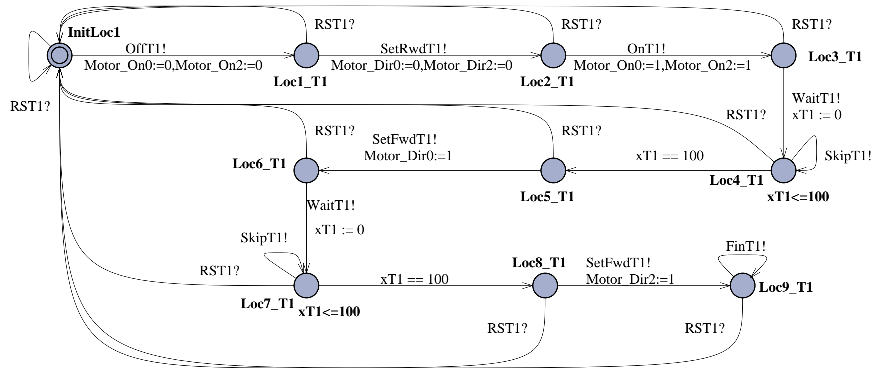

Fig. 14. Model of task 1.

formula in Uppaal. Instead we have to make what is called a test automaton. This is an automaton which only monitors the behavior of the system. Figure 15 shows a test automaton for testing whether the response time from sensor 0 has been read with value one by task 0 and until task 1 responds is less than 16 milliseconds. Some

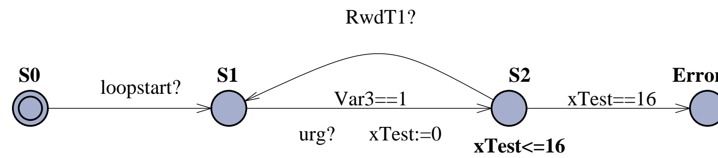

Fig. 15. A test automaton.

auxiliary channels are added to the model to synchronize with the test automaton. Firstly, we will only consider the program after the initialization is finished therefore the first channel. The channel *urg* is only used to make the channel what is called an *urgent* channel. An urgent channel is a special kind of channel which, when enabled, must be executed without any time delay. This channel is enabled when the program reads that the sensor has been pressed. If the output ports are set in reverse before 16 milliseconds have passed we

enter the state *S2* again and otherwise the *Error* state is entered. All channels but the one to the error state are urgent. We can now ask whether the *Error* state is reachable. If this is not the case the response always arrives within 16 milliseconds. In this case the *Error* state is not reachable but if 16 is changed to 15 then the *Error* state is reachable. If we wanted to test the response time from the sensor was touched we need a more complex test automaton taking into account that the reading of the sensor in this model can be set to one and then zero before the program reads it.

We might also want to verify that the output ports are set in reverse direction and turned on for a given time when the sensor reading changes to one. For this we need another model of the environment. When one of the readings change from zero to one the output ports are set in reverse. While the output ports are in reverse the reading from the other sensor might also change from zero to one. As a response to this the ports would be stopped. In general we will have to define a more precise model of the sensor readings if we want to prove more involved properties about our program or the movement of the car itself.

8 Conclusion

We have presented a method for translating RCX™ programs to networks of timed automata in a format readable by Uppaal. Applying this translation gives the possibility of reasoning formally about the behavior of the program using Uppaal. The translation have been implemented and tested on a number of examples with success.

Even though the method described here is specific to the RCX™ language, we believe that the principles can be carried over to most other assembly like (real-time) languages. There is a number of things one should take into consideration before trying to do this. Modeling other addressing forms like indirect addressing will be a lot more involved though it can be done. A detailed knowledge of the execution of programs or a formal semantics is needed for the model to make sense. If one wants to prove strong timing bounds for programs, precise timing information of the instructions will be needed in the model.

Exploring how good a relationship we can get between the behavior defined by timed automata and the behavior of LEGO® systems will be interesting. We cannot model the behavior of the physical system completely but we hope to be able to model it in such a way, that it makes sense to relate a number of properties of the formal model to the real system. In doing this we will have to define more detailed models of the environment.

Acknowledgements We would like to thank Michael Andersen from LEGO® for useful comments on the RCX™.

References

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:pages:3–34, 1995.

[AD90] R. Alur and D.L. Dill. Automata for modeling real-time systems. *Proc. of ICALP'90*, LNCS 433:pages 322–335, 1990.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126, 1994.

[Hen96] T. A. Henzinger. The theory of hybrid automata. *Proc. of LICS'96*, 1996.

[HHWT95] T. A. Henzinger, P.H. Ho, and H. Wong-Toi. A users guide to HYTECH. *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 1019, 1995.

[LEG98] LEGO. *Software developers kit*, November 1998. See `http://www.legomindstorms.com/`.

[LPY95] K. G. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. *In Proceedings of the 10th International Conference on Fundamentals of Computation Theory*, LNCS 965:pages 62–88, 1995.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *In Springer International Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.

[Yov97] S. Yovine. Kronos: A verification tool for real-time systems. *In Springer International Journal of Software Tools for Technology Transfer*, Vol. 1, October 1997.

Recent BRICS Report Series Publications

RS-00-17 **Thomas S. Hune.** ***Modeling a Language for Embedded Systems in Timed Automata*****. August 2000. 26 pp. Earlier version entitled *Modelling a Real-Time Language* appeared in Gnesi and Latella, editors, *Fourth International ERCIM Workshop on Formal Methods for Industrial Critical Systems*, FMICS '99 Proceedings of the FLoC Workshop, 1999, pages 259–282.**

RS-00-16 **Jiří Srba.** ***Complexity of Weak Bisimilarity and Regularity for BPA and BPP*****. June 2000. 20 pp. To appear in Aceto and Victor, editors, *Expressiveness in Concurrency: Fifth International Workshop EXPRESS '00 Proceedings*, ENTCS, 2000.**

RS-00-15 **Daniel Damian and Olivier Danvy.** ***Syntactic Accidents in Program Analysis: On the Impact of the CPS Transformation*****. June 2000. Extended version of an article to appear in *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, 2000.**

RS-00-14 **Ronald Cramer, Ivan B. Damgård, and Jesper Buus Nielsen.** ***Multiparty Computation from Threshold Homomorphic Encryption*****. June 2000. ii+38 pp.**

RS-00-13 **Ondřej Klíma and Jiří Srba.** ***Matching Modulo Associativity and Idempotency is NP-Complete*****. June 2000. 19 pp. To appear in *Mathematical Foundations of Computer Science: 25th International Symposium*, MFCS '00 Proceedings, LNCS, 2000.**

RS-00-12 **Ulrich Kohlenbach.** ***Intuitionistic Choice and Restricted Classical Logic*****. May 2000. 9 pp.**

RS-00-11 **Jakob Pagter.** ***On Ajtai's Lower Bound Technique for R-way Branching Programs and the Hamming Distance Problem*****. May 2000. 18 pp.**

RS-00-10 **Stefan Dantchev and Søren Riis.** ***A Tough Nut for Tree Resolution*****. May 2000. 13 pp.**

RS-00-9 **Ulrich Kohlenbach.** ***Effective Uniform Bounds on the Krasnoselski-Mann Iteration*****. May 2000. 34 pp.**

RS-00-8 **Nabil H. Mustafa and Aleksandar Pekeč.** ***Democratic Consensus and the Local Majority Rule*****. May 2000. 38 pp.**