Translating Uppaal to Not Quite C

M. Hendriks

Computing Science Institute/

CSI-R0108 March 2001

Computing Science Institute Nijmegen
Faculty of Mathematics and Informatics
Catholic University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

Translating Uppaal to Not Quite C*

Martijn Hendriks

Subfaculty of Computer Science, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
martynhe@sci.kun.nl

Abstract. This project presents a simple translation from Uppaal models of real-time controllers to NQC programs. The modeling of these controllers in Uppaal provides a way to verify the requirements on these controllers. The user directs the translation by defining a type for each variable used in the model and by assigning each automaton in the model to a controller. The translation, that has been implemented in the tool `uppaal2nqc`, results in a set of NQC programs that, when all NQC programs are run concurrently, approximately realizes a subset of the executions of the model. An Uppaal model of controllers of an experimental LEGO setup has been translated and the resulting NQC programs have been run in this setup to validate the translation.

Keywords: Real-time, timed automata, compiling, model-checking, synthesis.
AMS Subject Classification: 68Q60.
CR Subject Classification: D.2.4, D.3.4.

1 Introduction

Model checkers emerge as practical tools for the modeling, validation and verification of real-time systems. They support the development of formal models of real-time controllers and they provide an easy way to compute properties of these models. All requirements the real-time controller must satisfy can be formulated as properties of the model. A controller of a railroad crossing for example, must satisfy some safety and liveness requirements. Such a safety requirement could be that the gate must be closed if a train is near. An example of a liveness requirement is that the gate must eventually open so that other traffic can cross the railroad. If a model possesses all properties, and thus satisfies all requirements, it is assumed to be correct and the next step is then to realize this formal model. This realization however, is in general a manual process in which the semantics of the model could be misinterpreted, possibly resulting in a badly behaving real-time controller.

Hune, Larsen and Pettersson describe in *Guided Synthesis of Control Programs Using Uppaal* how a scheduling problem for a batch plant is solved using the model checker Uppaal and how a realization of the controller of the plant is automatically obtained [1,2,3]. The problem they face is to synthesize a controller such that the plant produces a certain amount of batches of different qualities of steel within a given amount of time. Hune et al. start with

* This report has been written in the context of the course *Research Lab 1* in the fall of 2000.

an Uppaal model of the plant that models *all* it's possible behaviors. The *good* behavior of the plant, that will produce the batches in time, has been translated to a reachability question that is solvable using Uppaal. The result of the reachability analysis is a trace of actions of the model with timing information of those actions. Because Hune et al did not expect to be able to validate their approach in the real steel plant, a LEGO plant, controlled by several LEGO RCX computers, has been used [4]. Therefore the timed trace has been projected to LEGO RCX control programs. The validation of these control programs in the physical LEGO plant turned out to be successful. A drawback is that the RCX control programs cannot handle input, since only *one* possible execution of the Uppaal model is realized.

In this project is assumed, in contrast to the approach of Hune et al, that a formal model of a real-time controller is already available. The problem is then to find a way to automatically obtain control programs that realize this model. The relevance of a solution of this problem is twofold. First, time is gained if one can automatically obtain a realization of a formal model instead of realizing the model by hand. Second, the realized real-time controllers will, under certain assumptions, function as specified in the requirements, if these requirements have been verified by the model checker *and* if the automatic realization preserves the semantics of the formal model.

More specific, this project proposes a translation of Uppaal models to the programming language Not Quite C (NQC) for the LEGO RCX computer. Two problems arise during the modeling. First, an Uppaal model models in general several RCX computers and these models should thus result in several NQC programs. Second, a way must be found to model all specific aspects of the RCX in Uppaal. For example, the user must be able to model the sensor value of some RCX sensor in Uppaal and this must be projected to NQC. These problems are solved by letting the user insert *compiler directives*, that direct the translation, in the Uppaal model.
Ideally, the translation should preserve the semantics of the Uppaal model. As we will indicate, this is not always the case. The control structure of the NQC programs assures that a certain subset of the executions of the Uppaal model is approximately realized. The translation has been implemented in a compiler called `uppaal2nqc`. This compiler has been used to translate an Uppaal model of an experimental LEGO setup to NQC. Various test runs showed that the automatically obtained control programs behaved like expected.

The rest of this report is organized as follows: First the properties of the target platform, the RCX computer and the programming language NQC are studied in section 2. In section 3 the model checker Uppaal is introduced. In section 4 is explained why and how the user must include some additional information in the Uppaal model. Also an example of an Uppaal model, the model of an experimental LEGO setup available at the KUN, is given. Then, in sections 5 and 6, a translation of Uppaal models to NQC programs is proposed. The constraints that this translation places on the Uppaal models are described in section 7. Then the relation between the translation and the model is explained in section 8. Finally, the translation and future work are discussed.

2 The RCX platform and NQC

The LEGO RCX 1.0 computer is a big LEGO brick with a processor inside. It is capable of interpreting an assembly like low level language. Uppaal models will be translated to the C like programming language NQC. Dave Baum's compiler will then translate this NQC program to the assembly language and download the assembly to the RCX [5]. The RCX computer has been chosen as target platform for the translation because an experimental LEGO setup, controlled by three RCX computers, was readily available for testing purposes. Uppaal models are not translated directly to the assembly language of the RCX because NQC is expressive enough and is easier to understand and learn than the assembly language. This last point presumably makes the construction of the translation easier. Though an argument in favor of the assembly language is that the resulting code will probably be more efficient, the extra time needed to learn this language was not available.

In the following subsections the RCX hardware and the various RCX API calls used for interfacing with the hardware are described. These are not all features of the RCX and NQC, but they are the ones used for the translation.

2.1 Sensors and actuators of the RCX

A RCX can use three sensors and three actuators of various kinds. Possible sensors are light sensitive sensors and touch sensitive sensors. Three *attributes* are associated with each physical sensor. Two of these attributes, the *type* and the *mode* of the sensor must be configured before using the sensor. After this configuration, the sensor attribute *value* can be read. These tasks can be accomplished by the RCX API calls of which the C like prototypes are stated below:

`void SetSensorType(`*sensor*`,`*type*`);`	Configures the sensor type.
`void SetSensorMode(`*sensor*`,`*mode*`);`	Configures the sensor mode.
`int SensorValue(`*sensor*`);`	Reads the sensor value.

NQC provides C like macro definitions. The arguments *sensor*, *type* and *mode* of the three functions above must be such NQC macro's, namely the ones listed in the following tables.

sensor
SENSOR_1 or 0
SENSOR_2 or 1
SENSOR_3 or 2

type
SENSOR_TYPE_TOUCH
SENSOR_TYPE_TEMPERATURE
SENSOR_TYPE_LIGHT
SENSOR_TYPE_ROTATION

For every possible value of the *mode* argument, the domain of the value returned by the call `SensorValue(x)` is given in the following table:

mode	domain of sensor values
SENSOR_MODE_RAW	[0,1023]
SENSOR_MODE_BOOL	[0,1]
SENSOR_MODE_PERCENT	[0,100]
SENSOR_MODE_ROTATION	16 ticks per revolution

In NQC, every argument of such an API call must be one of the listed macro's. For example, the second line of the following code is not correct:

```
int x = SENSOR_MODE_BOOL;
SetSensorMode(SENSOR_1,x);
```

This code can be replaced by the following, correct, NQC code:

```
SetSensorMode(SENSOR_1,SENSOR_MODE_BOOL);
```

Possible actuators that can be used with the RCX are motors and lights, though many more actuators can be build. Every actuator has three attributes that must be configured: the *mode*, the *direction* and the *power*. This configuration can be accomplished by the RCX API calls of which the C like prototypes are stated below:

`void SetOutput(`*output*`,`*mode*`);` Switches the actuator on or off.

`void SetDirection(`*output*`,`*dir*`);` Switches the rotation direction of a motor.

`void SetPower(`*output*`,`*power*`);` Sets the power; 0 is min. and 7 is max. power.

The possible values of the attributes *output*, *mode*, *dir* and *power* are listed below:

output
OUT_A
OUT_B
OUT_C

mode
OUT_OFF
OUT_ON
OUT_FLIP

dir
OUT_FWD
OUT_REV
OUT_TOGGLE

power
$\in [0, 7]$

As with the input attributes, the RCX API calls are only syntactical correct if the arguments, with exception of the *power* argument, are the macro's listed in the tables above.

The sensor and actuator attributes have default values that are used if the attributes are not configured by the programmer. For these default values, see Dave Baum's *NQC programmer's guide* [5].

2.2 Infrared messages

The RCX can use an infrared transmitter and receiver to communicate with other RCX's and to enable the easy downloading of control programs. The buffer for the incoming mes-

sages has a size of one in which the most recent message is stored. A message is an integer and can have a value in the range [0,255]. The default value of the message buffer is 0. The following API calls concerning messages are available (again in C like prototypes):

`void ClearMessage();`	Sets the value of the message buffer to 0.
`void SendMessage(`n`);`	Sends a message with value $n \in [0, 255]$.
`int Message();`	Returns the value of the message buffer.

The `SendMessage(n)` call sends a message to *all* RCX's that are "close" to it, including itself. Note that if the `ClearMessage()` call is not used by a RCX in a certain system, then the value of the message buffer can be regarded as a global value. Every RCX can read and write this value and all message buffers of the RCX's contain the same value.

2.3 Timers

A RCX can handle a maximum of four timers or clocks. Each timer is identified by a unique natural number in the range [0,3]. The timer has a resolution of 100 ms. That means that the timer value is increased by one every 100 ms. The following API calls are available (again in C like prototypes):

`void ClearTimer(`n`);`	Resets timer $n \in [0, 3]$ to zero.
`int Timer(`n`);`	Returns the value of timer $n \in [0, 3]$.

2.4 Variables, tasks and subroutines

A RCX can handle a maximum of 32 16-bit signed integer variables. This thus means that a NQC program can use at most 32 variables.

The RCX has the feature that a maximum of 10 tasks, that will be scheduled in a round-robin way, can be defined. Every NQC program must at least have one task, the main task. This main task can start the other tasks of the NQC program. A RCX can also use a maximum of 8 subroutines. For details about these tasks, subroutines and NQC in general, see Dave Baum's *NQC programmer's guide* [5].

3 Uppaal

Uppaal uses the theory of timed automata to model, simulate and verify systems (e.g. real-time controllers). The ease of usage was the main reason to choose for this tool. A brief introduction of Uppaal, based upon definitions and descriptions given in *Uppaal in a*

Nutshell [2], is given is this section. When in the following sections is referred to Uppaal, the current version at the time of this report, that is version 3.0.41, is meant.

An Uppaal model consists in general of a network of timed automata with a finite control structure and real valued clocks that communicate through channels and/or shared variables. Such a timed automaton A is a tuple $(L, L^0, Act, I, C, Inv, E)$, where

- L is a finite set of locations,
- L^0 is the initial location,
- Act is a finite set of *sending* and/or *receiving* actions and the internal action τ,
- I is a finite set of integer valued variables,
- C is a finite set of real valued clocks,
- Inv is a partial mapping that assigns location invariants over C to locations is L,
- $E \subseteq L \times G \times Act \times \mathcal{P}(R) \times L$ corresponds to the set of edges. G is the set of all guards over C and I and R is the set of assignments over C and I.

Actions are used for synchronization between *two* automata in the system. An action `s` can be a sending action, in Uppaal denoted by `s!`, or a receiving action, in Uppaal denoted by `s?`. If, for example, automaton A has an enabled edge with action `s!` and automaton B has an enabled edge with action `s?`, then they can both execute that transition in *one* step. For details of actions and synchronization, see *Uppaal in a Nutshell* and *Timed automata* [2,7].

A location invariant over C has the following form: $x \sim n$ where $x \in C$, $\sim \in \{<, \leq\}$ and $n \in \mathbb{N}$. Control can only be in a certain location, if the location invariant of that location is *not* violated. Therefore location invariants insure progress. The default location invariant is *true*.

A guard over the sets C and I is a conjunction of timing and data constraints. A timing constraint is of the form $x \sim n$ or $x - y \sim n$, where $x, y \in C$, $n \in \mathbb{N}$ and $\sim \in \{<, \leq, \geq, >, =\}$. A data constraint is of similar form $i \sim k$ or $i - j \sim k$, where $i, j \in I$, $k \in \mathbb{Z}$. The default guard of an edge is *true*.

An assignment over C or I is of the form $w := e$, where w is a clock or integer variable and e is an expression. In Uppaal, clock assignments or resets must have the simple form $x := n$, where $x \in C$ and $n \in \mathbb{N}$. The integer assignments may have more complex forms, see the help menu in the tool Uppaal. The definition uses the power set of all assignments. This means that an edge can be labeled with an arbitrary number of these assignments.

The semantics of an Uppaal model are defined by the underlying transition system of the timed automata. Two kinds of transitions are possible:

action transitions If two automata can synchronize in a certain state, that means that both are in a location from which an edge is enabled *and* these two enabled edges contain complementary actions, then they can both take that edge, leading in a single step to a new state. If an automaton has an internal edge enabled, that is an edge labeled with action τ, that edge can be taken without any synchronization.

delay transitions In a certain state, as long as none of the automata violates the location invariant of it's current location, time may progress without affecting the current

location of the automata and with all clock values incremented with the elapsed duration of time.

Uppaal also provides some syntactical and semantical additions to these definitions. These are the notions of *urgent* channels/actions, *urgent* locations and *committed* locations. The meaning of an urgent channel is that there can be no delay transition if the synchronization action by that urgent channel is enabled. The meaning of an urgent location is that there is an extra clock x that is reset to zero on all in going transitions to that location. An extra guard, $x \leq 0$, is added to the location invariant. Intuitively this means that there cannot be any delay in that location. The meaning of a committed location in a certain automaton is that it is an urgent location, *and* any action transition *must* involve that particular automaton.

4 How to model RCX controllers in Uppaal

In this project we build a real-time system by adding control software to existing hardware. The Uppaal model that will be translated to the control software is thus based upon the existing situation. Now consider the case of a LEGO real-time system with a certain number of RCX computers. This situation will in general exist because of two reasons. The first reason is that the RCX computer has only limited resources, like a maximum of three sensors and three actuators. The second reason is that a real-time system might consist of several independent subsystems that must not be physical connected to each other, but that must interact with each other. The Uppaal model should then be translated to an equal number of NQC programs: one NQC program for every RCX computer.

Moreover, when a controller is to be realized on a certain platform, the user must be able to include all aspects and properties of that platform in the model. If the target platform is the RCX, then it should be possible to model all sensor and actuator attributes and the IR buffer, explained in section 2.1 and 2.2.

These considerations give rise to the following problems:

(1) *Distribution*: A LEGO real-time system consists in general of n RCX's. The translation of the Uppaal model of this system should result in n NQC programs: one for every RCX. The problem thus is to find out which automata in the Uppaal model to use for which NQC program. Furthermore, the Uppaal model can contain automata that model the environment of the system for verification and simulation purposes. These environment automata must not be translated to NQC. How are these automata recognized?

(2) *RCX specific properties*: The user must be able to model all aspects of the RCX in Uppaal. It is natural to model all attributes, like the sensor values and actuator modes, as integer variables. Now the problem is to find out which variable in the Uppaal model models which aspect of the RCX.

An annotation method is introduced to solve these problems. The user should add certain Uppaal comments above the integer variable declarations and above the process definitions

of the Uppaal model. If placed above an integer declaration, the comment tells the compiler which aspect is modeled by this integer variable. If placed above a process definition, the comment tells the compiler whether or not that automaton models the environment and, if not, for which NQC program it is to be used. These *compiler directives*, recognizable by the keyword `RCX`, are more formally explained in sections 5.1 and 5.2.

4.1 Trains and Gates: an Uppaal example

An experimental LEGO setup has been built at the KUN by Jeroen Kratz [6]. This setup consists of two interwoven tracks: a circular railroad track with a train and an other circular track that is followed by a car. These tracks intersect each other twice. The first intersection is a bridge and the second intersection is a railroad crossing guarded by gates. The setup uses three RCX's. One RCX is used for controlling the train that drives in circles over the railroad track. An other RCX is used for controlling the railroad crossing. The third RCX is used for controlling a car that follows the track and thus once in a while encounters the railroad crossing. In this project only the RCX's that control the train and the crossing are considered.

The RCX that controls the crossing uses a motor to lower and raise the gates. It also uses two warning lights that signal the car to stop when the gates are lowered. This RCX receives input from two light sensors that signal the approach or departure of a train. It also receives input from a touch sensor to determine whether or not the gates were lowered successfully. The RCX that controls the train only uses two motors for it's driving. The two RCX's communicate using the infrared channel. For example, if the gates are not lowered successfully, then the train is warned of this dangerous situation by a *failure* message. If the train has passed the gates, the RCX that controls the crossing signals the train that it can speed up again by a *ok* message.

The control programs for these two RCX's must exclude certain "dangerous" situations such as collisions between the train and the car. In order to obtain such programs, the controller software to run on the RCX's that control the train and the crossing are modeled in Uppaal and this Uppaal model is then compiled to NQC programs. The Uppaal model is given below.

The process definition section and the system definition have been displayed below. The comments above the process definitions of `Train` and `TrainAlarm` direct these two automata to the NQC program `Train`. The comment above the process definition of `Gate` directs this automaton to the NQC program `Gate`. The comment above the process definition of `Env` defines this automaton as an automaton that models the environment.

```
// RCX block Train
Train := aTrain ( failure, ok );

// RCX block Train
TrainAlarm := aTrainAlarm ( );
```

```
// RCX block Gate
Gate := aGate ( failure, ok, sns1v, sns2v, sns3v );

// RCX environment
Env := anEnvironment ( sns1v, sns2v, sns3v );

system Train, TrainAlarm, Gate, Env;
```

The global declarations section of the Uppaal model has been displayed below. The second comment, being `// RCX ir`, is a compiler directive. It means that the global variable `ir` models the IR buffer of the RCX's in the system. The third, fourth and fifth comment define that the three associated global variables model sensor values.

```
// Message constants:
const failure 1;
const ok 2;

// RCX ir
int[0,255] ir:=0;

chan start_alarm, stop_alarm;

// RCX sns_1_value
int sns1v;
// RCX sns_2_value
int sns2v;
// RCX sns_3_value
int sns3v;
```

Now only the templates `aTrain`, `aTrainAlarm`, `aGate` and `anEnvironment` that are used in the process definition section to define the automata in the system, must be given.

First, the template `aTrain` is defined. Therefore it's local declarations are given below. The variables `l_pow` and `r_pow` model the power attributes of output A and output C of the RCX that controls the train. The variables `left_dir` and `right_dir` model their directions and the variables `left` and `right` model the mode of these outputs.

```
const OUT_ON  1;
const OUT_REV 0;

// RCX out_A_power
int l_pow:=7;
// RCX out_C_power
```

```
int r_pow:=7;

// RCX out_A_direction
int left_dir:=OUT_REV;
// RCX out_C_direction
int right_dir:=OUT_REV;

// RCX out_A_mode
int left:=OUT_ON;
// RCX out_C_mode
int right:=OUT_ON;
```

Figure 1 shows the template that models the controller of the train. The train is a very simple automaton. It has two transitions that both contain a synchronization action to start or stop the alarm. The alarm is possibly started if the variable that models the IR buffer of the RCX's contains the value `failure`. The alarm can be stopped if the variable that models the IR buffer of the RCX's contains the value `ok`.

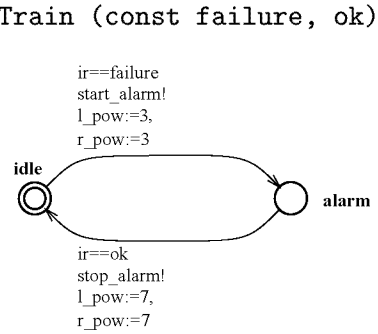


Figure 1: Template `aTrain`.

Second, the template `aTrainAlarm` is defined. Therefore it's local declarations are given below. The variable `alarm_light` models the mode attribute of output B of the RCX that controls the train. This is the alarm light of the train that will flash if a failure message has been received.

```
clock x;

int light;

const OUT_OFF 0;
const OUT_ON  1;
```

```
// RCX out_B_mode
int alarm_light:=OUT_OFF;
```

Figure 2 shows the template that models the alarm of the train. The job of this automaton is to let a light flash. This is modeled by switching the value of variable `alarm_light`, that models the output mode of actuator B, between the values `OUT_ON` and `OUT_OFF`.

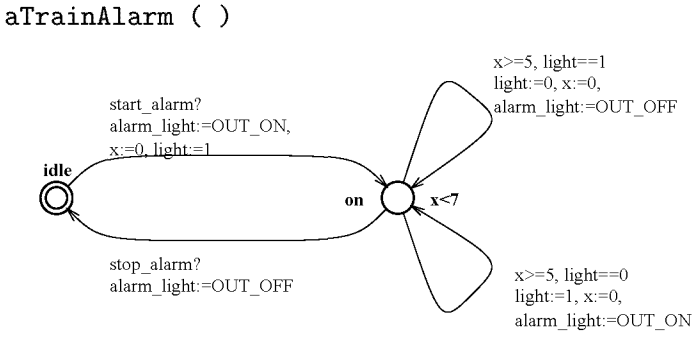


Figure 2: Template `aTrainAlarm`.

Third, the template `aGate` is defined. The automaton `Gate`, that is an instance of this template, controls the railroad crossing on its own: it is directed to an other NQC program than the automata `Train` and `TrainAlarm`. All the local declarations are given below. The variables that are linked by the compiler directives to output A of the RCX, being `md`, `mp` and `m`, model the motor that raises and lowers the gates. These values are initialized in such a way that the motor is turned on and the gates are raised. The variables that are linked to output B and C of the RCX, being `l1p`, `l1`, `l2p` and `l2`, model the warning lights of the gate. These are initialized in such a way that they are off. The variables `s1t` and `s1m` model the type and mode attributes of the touch sensor that senses whether or not the physical gates are lowered. The variable that models it's value is an parameter of the template and it is declared globally. In this way the variable can be shared between this automaton and the environment automaton, which is necessary for verification and simulation purposes. The same holds for the light sensors that detect the approach or departure of a train.

```
clock z;

// Threshold value train approach:
const TR_APP 40;
// Threshold value gate closed:
const GT_CL 45;

// These constants are also NQC macro's:
```

```
const OUT_OFF 0;
const OUT_ON  1;
const OUT_REV 0;
const OUT_FWD 1;
const SENSOR_TYPE_TOUCH 1;
const SENSOR_TYPE_LIGHT 2;
const SENSOR_MODE_RAW     1;
const SENSOR_MODE_PERCENT 2;

// RCX out_A_direction
int md:=OUT_FWD;
// RCX out_A_power
int[0,7] mp:=2;
// RCX out_A_mode
int m:=OUT_ON;

// RCX out_B_power
int[0,7] l1p:=7;
// RCX out_B_mode
int l1:=OUT_OFF;
// RCX out_C_power
int[0,7] l2p:=7;
// RCX out_C_mode
int l2:=OUT_OFF;

// RCX sns_1_type
int s1t:=SENSOR_TYPE_TOUCH;
// RCX sns_2_type
int s2t:=SENSOR_TYPE_LIGHT;
// RCX sns_3_type
int s3t:=SENSOR_TYPE_LIGHT;

// RCX sns_1_mode
int s1m:=SENSOR_MODE_PERCENT;
// RCX sns_2_mode
int s2m:=SENSOR_MODE_PERCENT;
// RCX sns_3_mode
int s3m:=SENSOR_MODE_PERCENT;
```

Figure 3 shows the template that models the controller of the gate. In the initial location, it assumes that the train is far away and the gates are raised by assigning the right value to the variables modeling the actuators of the gate; see the model and the local declarations. While in the location `idle`, the gate waits until one of it's two light sensors senses the

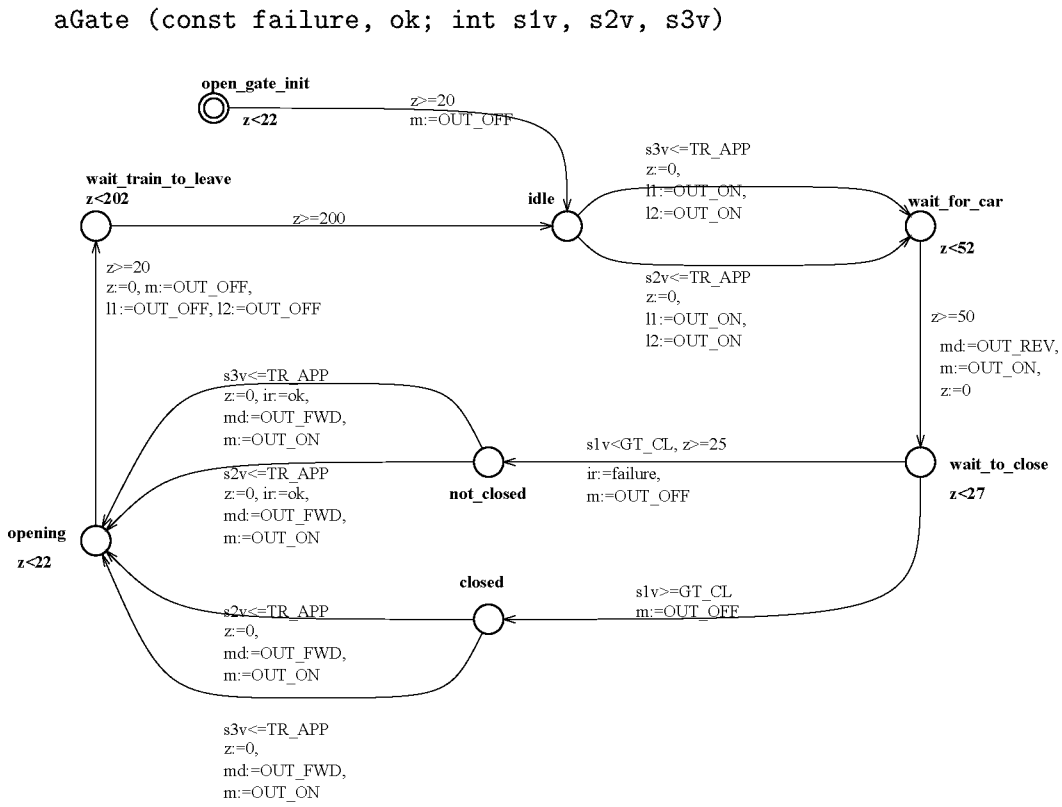


Figure 3: Template `aGate`.

approach of a train. The transitions to the location `wait_for_car` switch on the warning lights that stop the car at the gate. The control remains between 50 and 52 time units in the location `wait_for_car` to possibly let the car move off the railroad before closing the gate. The transition to `wait_to_close` activates the actuator that closes the gate. If the gate is closed properly, sensed by the touch sensor `s1v`, the actuator that closes the gate is turned off and control is passed to `closed`. If more than 26 time units elapse without the sensor reporting a properly closed gate, control is passed to `not_closed` and the actuator that closes the gate is turned off. Also the `failure` is send. From the locations `not_closed` and `closed` there are two transitions to `opening` as the train can leave in two different directions. If that happens from the location `not_closed`, the `ok` is send. Control remains in the location `opening` to let the gate open before control is passed to `wait_train_to_leave`. The effect of this location is that for a period the sensors are not used. This is necessary due to the possible slowness of the physical train. The timing constants that occur above result from experiments of Jeroen Kratz.

Finally, the template `anEnvironment` is defined. It has no local declarations, but has three variable arguments. These arguments model the value attribute of the sensors that are used

by the RCX that controls the crossing. They are also used by the template `aGate`. The job of this automaton is to provide sensor input. Figure 4 shows the template that models the environment of the system.

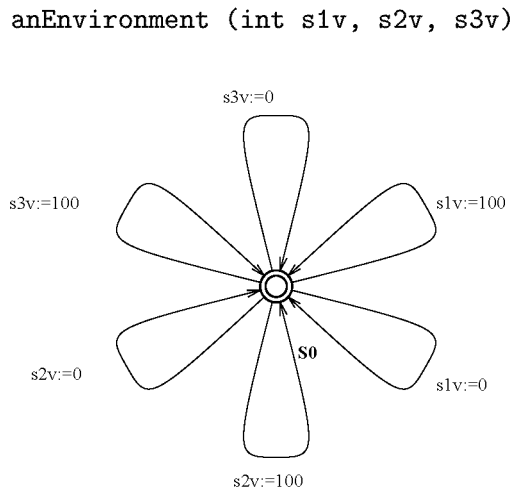


Figure 4: Template `anEnvironment`.

For verification purposes, some properties of this model have been proved by Uppaal. Property (1) states that if the controller for the gate is in the location `not_closed`, that means that the physical gate probably is not closed while it should be closed, then the variable `ir`, that models the IR buffer of the RCX's, has the value `failure`.

```
A[] (Gate.not_closed imply ir==failure)        (1)
```

Property (2) states a liveness property. Some executions of the model reach the location where the controller for the gate is in the location `opening`.

```
E<> (Gate.opening)        (2)
```

Property (3) states that if the controller for the gate is in the location `closed`, that means that the physical gate is closed, then the variable `ir`, that models the IR buffer of the RCX's, has the value `ok` or 0. This variable is 0 when no IR messages have been sent yet.

```
A[] (Gate.closed imply (ir==ok or ir==0))        (3)
```

Property (4) states that if the controller for the train is in the location `alarm`, then the alarm light is flashing and the train is moving slow.

```
A[](Train.alarm imply
(TrainAlarm.on and Train.l_pow==3 and Train.r_pow==3))
```
(4)

These properties all seem relatively straightforward. It is however not difficult to imagine models of real-time controllers where things are very complicated. In those cases Uppaal provides an easy way, *if* the model is not too large, to prove, or disprove, properties.

5 Parsing Uppaal system definition files

The first step in realizing an Uppaal model, is the parsing of the `xta` file that contains the system definition. The UNIX tools `yacc` and `lex` have been used to construct a parser for the `xta` file format. The parser does not recognize conditional assignments, i.e. `id:=(x<10?0:1)`. These can be avoided by using two transitions. It must be noted that the assumed grammar is probably not completely correct because it is derived from the structure of some `xta` files and of the BNF grammar in the help menu of the tool Uppaal itself. This BNF grammar does not contain all rules to generate `xta` files. In the testing of the tool `uppaal2nqc` however, the parser did a good job.

The set of all timed automata of the system, these are listed in the system definition section of the `xta` file, must be extracted. Therefore, the templates are instantiated with the corresponding arguments, listed in the process definition section. During this process, some integer variables, constants and clocks of the model will be renamed. Constants that are NQC macro's, see section 2.1, are never renamed. Integer variables, other constants and clocks that are globally declared in the Uppaal model will keep their names. Integer variables, other constants and clocks that are local to an automaton in the Uppaal model, will be prefixed with the name of the automaton and an underscore. Integer variables, other constants and clocks that are arguments of an automaton, will be resolved to the name, or in case of constants to the value of the global parameter.

The successful parsing of a `xta` file will result in a set of timed automata as defined in section 3, extended with some extra information:

(1) Extra information originating from Uppaal are the notions of constants, arrays, urgency and commitment.

(2) As mentioned in section 4, the user must also include extra information to make the realization of the controller(s) in the model not too difficult: First a so-called *type mapping* must be defined; this is explained in section 5.1. Second, the user must assign each automaton to a NQC program. This is explained in section 5.2.

5.1 Constructing the type mapping

In section 4 was pointed out that each sensor and actuator attribute and the IR buffer of a RCX can be modeled in the Uppaal model. To realize the model, it is necessary to know which variable models which attribute and which variable models the IR buffer. This

information must be included in the Uppaal model in the form of comments above variable declarations. A *type mapping* t must thus be constructed that maps each variable in the Uppaal model, this set is denoted by I_A, to a type:

$$t : I_A \rightarrow T \text{ where } T = \{ \begin{array}{l} \texttt{sns\_1\_type, sns\_1\_mode, sns\_1\_val,} \\ \texttt{sns\_2\_type, sns\_2\_mode, sns\_2\_val,} \\ \texttt{sns\_3\_type, sns\_3\_mode, sns\_3\_val,} \\ \texttt{out\_1\_mode, out\_1\_dir, out\_1\_power,} \\ \texttt{out\_2\_mode, out\_2\_dir, out\_2\_power,} \\ \texttt{out\_3\_mode, out\_3\_dir, out\_3\_power,} \\ \texttt{ir, none}\} \end{array}$$

The first 18 elements of T are used to express which sensor or actuator attribute is modeled. The type `ir` is used to express that a variable models the IR buffer of all RCX's in the system. As mentioned in section 2.4, the values of *all* IR buffers are the same if no NQC program in the system uses the `ClearMessage()` call. Therefore all IR buffers can then be modeled by one variable. The type `none` is used to express the fact that a Uppaal variable does not model a sensor or actuator attribute and that it does not model the IR buffer. Such variables are called internal variables.

These integer variable annotations, or compiler directives, must be placed above the variable declarations in the Uppaal model by the user. These compiler directives are regarded by Uppaal as comments, but the parser can recognize and process them. The general form of the compiler directives for the integer variables is:

$$\texttt{// RCX } p \qquad \text{where } p \in (T \backslash \{\texttt{none}\})$$

If a this compiler directive is stated above the declaration of variable n, then the type mapping t is updated such that $t(n) = p$. If an integer variable has no compiler directive, it is assumed that it's type is `none`.

5.2 Controller and environment automata

Section 4 also pointed out that there exist compiler directives for the process definitions in the Uppaal model. The first of these is the environment directive:

```
// RCX environment
```

This compiler directive means that the automaton that is defined directly below, is an environment automaton and it will therefore not be added to the set of timed automata that is the result of this parsing step.

As mentioned in section 4, an Uppaal model should in general result in several NQC programs. To define which automata are to be used for which program, the following compiler directive is used:

```
// RCX block <id>
```

This compiler directive means that the automaton defined directly below must be used to construct the NQC program with name `<id>`.

These two compiler directives must be placed above a process definition. All the processes defined in the process definition section of an Uppaal model must either have an environment compiler directive or a block compiler directive.

6 Realization of the controllers

The result of the previous step, the parsing of a `xta` file, is a set of timed automata extended with the extra information as specified in section 5. Now all automata that are directed to the same NQC program by the user, must be translated to a single NQC program. In the example, the automata `Train` and `TrainAlarm` are used to construct one NQC program and the automaton `Gate` is used to construct another NQC program. In general, the result of the parsing step is a set of timed automata $A = \{A_1, ..., A_n\}$ that can be partitioned into blocks by the compiler directives described in section 5.2. Every block B_i is then translated into one NQC program.

This set of NQC programs must simulate the underlying transition system of the Uppaal model for that system defines the semantics of the model [2,7]. If every NQC program simulates the transition system defined by the automata in a certain block and all blocks are translated to NQC, then the Uppaal model is approximately realized by the concurrent execution of these programs.

In the next subsections the translation of a block to a NQC program is described in a top-down way.

6.1 General control structure

To simulate the transition system the following execution model is implemented in NQC. Only the main task and two subroutines, explained in sections 6.4.2 and 6.5, are used. In the main task an infinite while-loop is used to interleave the transitions of the automata of the block in the finest possible way. Every automaton can execute zero or one action transition in one iteration of the body of the while loop. Before an automaton can determine whether or not to take an action transition, all input is read by a call to the subroutine `read_input`. An alternative is to read input only when it is needed in the evaluation of guards of edges. This can result in a scheme where the choice of which edge to take is based on possibly different input values. On top of that, it is probably more efficient to keep a local copy of the input values and to use those copies during the evaluation instead of reading the input every time it is needed from the hardware.

In order to implement this execution model, the "active" location of every automaton is held in a global variable, the program counter of that automaton. The name of this variable is the name of the automaton prefixed with `pc_`. The program counter is initialized to the initial location of the corresponding automaton during it's declaration. The program counter can have the values defined by all location names of that automaton and it can also have a special value, `deadlocked`, see section 6.5. The location names are defined by macro's in the NQC program as is explained in section 6.2. For example, the global declarations (and initializations) of the program counters and the global structure of the main task of block `Train` of the example of section 4.1 are depicted below:

```
/* Definitions of macro's ... see section 6.2 */

/* Declaration and initialization of the program counter(s): */
int pc_Train = idle,
    pc_TrainAlarm = idle;

/* Clocks, input variables and subroutines ...
   see section 6.3, 6.4.2  and 6.5 */

task main()
{
    /* Declaration of local variables ... see section  6.4.3 */
    /* Initialization of output attributes ... see section 6.4.1 */
    /* Reset all timers ... see section 6.3 */

    while(1)
    {
        /* Transitions for automaton Train: */
        read_input();
        if (pc_Train == idle)
        {
            /* Transitions ... see section 6.6 */
        }
        else if (pc_Train == alarm)
        {
            /* Transitions ... see section 6.6 */
        }

        /* Transitions for automaton TrainAlarm: */
        read_input();
        if (pc_TrainAlarm == idle)
        {
            /* Transitions ... see section 6.6 */
```

```
        }
        else if (pc_TrainAlarm == on)
        {
            /* Transitions ... see section 6.6 */
        }
    }
}
```

In the example above can be seen that if an automaton 's program counter has the value `deadlocked`, it never executes action transitions anymore. Otherwise, zero or one action transitions can be executed by the automaton before letting the other automata of the block execute zero or one action transitions.

This execution model also implicates that action transitions are taken as soon as they are enabled, with a certain timing uncertainty that is discussed in section 8.

6.2 Translation of Uppaal constants and location names

The NQC compiler provides the possibility to use C like macro definitions and these have been used to translate the constants that are used in the Uppaal model. Let us consider a constant c with value n that is used in some automaton in a block. The arguments of the API calls of section 2.1 are predefined in NQC. Therefore, if the string c is equal to such an argument, nothing has to be done because the NQC compiler will handle this macro. Otherwise, a line of the following form must be added to the NQC program:

```
#define c    n
```

In section 6.1 has been mentioned that every location of a timed automaton in the block is mapped to a natural number by a macro definition to the effect that the location names can then be used in the NQC program. Let $Locations_i = \bigcup_{A_k \in B_i} L_k$ be the set of all locations of a block B_i. Then a one-to-one mapping $N_l : Locations_i \rightarrow \mathbb{N}$ is constructed. For all $l \in Locations_i$ a line of the following form must be added (where $\mathtt{n} = N_l(l)$):

```
#define l    n
```

Also a macro has to be defined for the *deadlocked* state of an automaton. Therefore a constant d in the natural numbers is chosen that is not equal to a value used in the mapping N_l. The last macro is thus:

```
#define deadlocked    d
```

The macro's defined in the NQC program that results from the translation of block `Train` of the example are the following:

```
/* The constant(s) of block Train: */
#define failure   1
#define ok        2

/* The location(s) of block Train: */
#define idle        1
#define alarm       2
#define on          3
#define deadlocked  4
```

Note that the constants `OUT_ON`, `OUT_OFF` and `OUT_REV`, these are NQC macro's, are not defined.

6.3 Translation of Uppaal clocks

It is an easy and natural way to translate the clocks used by the automata in a certain block with use of the hardware timers of the RCX. These are hardware timers identified by a natural number and therefore a one-to-one mapping $c_i : CB_i \rightarrow \{0, 1, 2, 3\}$, where CB_i is the set of all clocks of the block B_i must be constructed for each block B_i to assign an unique number to each clock. In Uppaal the clocks all have the value 0 when the execution begins and therefore all clocks are reset to zero just before the start of the infinite while loop. This is achieved by adding the following line to the main task before the start of the while-loop: `ClearTimer(`$c_i(x)$`);` for every clock $x \in CB_i$. For the exact position of these lines, see section 6.1.

In the example Uppaal model, the translation of block `Train` gives following clock resets:

```
/* Reset the timer(s) to zero: */
ClearTimer(0);
```

6.4 Translation of Uppaal integer variables

The set of variables used in an Uppaal model can be partitioned into the input variables, the output variables and the internal variables. These three sets are treated different in this translation.

6.4.1 Output variables

The Uppaal integer variables that are *not* mapped to `sns_X_value`, `ir` or `none` are used for the modeling of output attributes. These variables will not be declared in the NQC program. This choice has been made because it saves a significant number of variables. Remember that a NQC program can only use 32 variables.

These output variables might be initialized in the Uppaal model. If so, then this initial value of the output attributes must be expressed before the start of the infinite while loop. For example, let n be such a variable that is initialized in the Uppaal model to value `x`. Then the translation adds a line of the following form to the main task, before the while loop:

- if $t(n) =$ `sns_`m`_mode`, add `SetSensorMode(SENSOR_`m`,x);` where $m \in \{1, 2, 3\}$.
- if $t(n) =$ `sns_`m`_type`, add `SetSensorType(SENSOR_`m`,x);` where $m \in \{1, 2, 3\}$.
- if $t(n) =$ `out_`m`_mode`, add `SetOutput(OUT_`m`,x);` where $m \in$ `{A,B,C}`.
- if $t(n) =$ `out_`m`_direction`, add `SetDirection(OUT_`m`,x);` where $m \in$ `{A,B,C}`.
- if $t(n) =$ `out_`m`_power`, add `SetPower(OUT_`m`,x);` where $m \in$ `{A,B,C}`.

For the exact position of these lines, see section 6.1.

In the example Uppaal model, the translation of block `Train` gives following output attribute initializations:

```
/* Initialize output variable(s): */
SetPower(OUT_A,7);
SetPower(OUT_C,7);
SetDirection(OUT_A,OUT_REV);
SetDirection(OUT_C,OUT_REV);
SetOutput(OUT_A,OUT_ON);
SetOutput(OUT_C,OUT_ON);
SetOutput(OUT_B,OUT_OFF);
```

6.4.2 Input variables & input subroutine

The values of the integer variables that are mapped to type `sns_X_value` or `ir` and the clock values are those that are continually changing due to the environment and the elapse of time. The input of the program is read in the infinite while-loop of the main task. For the reasons mentioned in section 6.1, a copy of every input value is kept in an integer variable of the NQC program. These values are updated by a call to the subroutine `read_input`. Because of the fact that subroutines cannot handle arguments and return values, the variables that contain these input values must be declared globally. Let IB_i be the set of all integer variables in block B_i. A global integer variable declaration must be added for every $x \in IB_i$, if $t(x) \in$ `{sns_X_value, ir}`, where `X` $\in$ `{1,2,3}`. Also such a declaration of a variable must be added for every $c \in CB_i$ (see section 6.2). For the exact position of these declarations, see section 6.1.

The following lines are added to the body of the subroutine `read_input` to update the values of the input variables:

- for every $x \in CB_i$ add the line `x = Timer(`$c_i(x)$`);`
- for every $n \in IB_i$:
  - if $t(n) =$ `sns_`m`_value`, add `n = SensorValue(`$m-1$`);` where $m \in \{1, 2, 3\}$.
  - if $t(n) =$ `ir`, add the line `n = Message();`

In the example Uppaal model, the translation of block `Train` gives following additional global declarations and the following input subroutine:

```
/* Declaration of the clock(s): */
int TrainAlarm_x;

/* The IR variable: */
int ir;

sub read_input()
{
    /* Read the rcx timer(s): */
    TrainAlarm_x = Timer(0);

    /* Read the IR value: */
    ir = Message();
}
```

6.4.3 Internal variables

Then there is the set of variables with type `none`. These *internal variables* are translated to variables local to the main task. The initial value is assumed to be zero if they are not initialized in the Uppaal model, because Uppaal also assumes this. For the exact position of these declarations, see section 6.1.

In the example Uppaal model, the translation of block `Train` gives following declaration:

```
/* Declaration of local variable(s): */
int TrainAlarm_light = 0;
```

6.5 Deadlock subroutine

It is imaginable that the realization of an Uppaal model can at a certain point in time *not* satisfy some location invariant. This, for example, can occur if more action transitions are forced in some time interval than the RCX can handle. In this translation has been choosen to *stop* an RCX that cannot satisfy some location invariant. In order to achieve this, a second subroutine, named `deadlock_automata`, is created. This subroutine sets all program counters to the `deadlocked` location, that has no outgoing transitions, and switches all motors of the RCX off.

In the example Uppaal model, the translation of block `Train` gives following deadlock subroutine:

```
sub deadlock_automata()
{
    pc_Train = deadlocked;
    pc_TrainAlarm = deadlocked;
    SetOutput(OUT_A,OUT_OFF);
    SetOutput(OUT_B,OUT_OFF);
    SetOutput(OUT_C,OUT_OFF);
}
```

6.6 Translation of transitions

The next step is to translate the action transitions of the Uppaal model. These are separated into the action transitions that do not use a synchronization action and into those that do use a synchronization action. Let l be the location in automaton A_k, named `K`, in block B_i of which al outgoing transitions must be translated.

Let (l, a, g, r, l') be a transition where $a = \tau$, thus a transition without synchronization action. This transition can be straightforward translated to NQC code, where `G` is the translation of the guard set g, `A` is the translation of the assignment set r and `new_location` is the name of l':

```
(else) if (G)
{
    A
    pc_K = new_location;
}
```

If another transition has already been translated for this location, the parenthesis around the keyword `else` must be removed. Otherwise, the parenthesis and the keyword `else` must be left out. This is to make sure that only one action transition starting in location l will be executed. The global control structure explained in section 6.1 assures that the whole automaton executes a maximum of one action transition.

Now, let $a \neq \tau$, thus a transition with synchronization action. All other, matching transitions in the other automata of block B_i must be found. Thus for all transitions (h, a', g', r', h') of an automaton $A_m \in B_i$ where $m \neq k$: if $a = a'$ *and* they match, then the following code must be added to the translation of the transitions of location l:

```
(else) if (pc_M == H && G1 && G2)
{
    A1
    A2
```

```
        pc_K = new_location_K;
        pc_M = new_location_M;
    }
```

In this code, `H` is the name of location h, `G1` is the translation of the guard set g, `A1` is the translation of the assignment set r and `new_location_K` is the name of l'. Symmetrically `G2` is the translation of the guard set g', `A2` is the translation of the assignment set r' and `new_location_M` is the name of h':

What rests is the translation of the location invariants, guards and assignments. The location invariants and guards of an Uppaal model can literally be included in the NQC program.

The translation of an assignment $n := x$ in some assignment set is as follows:

- if $t(n) =$ `sns_`m`_mode`, add `SetSensorMode(SENSOR_`m`,x);` where $m \in \{1, 2, 3\}$.
- if $t(n) =$ `sns_`m`_type`, add `SetSensorType(SENSOR_`m`,x);` where $m \in \{1, 2, 3\}$.
- if $t(n) =$ `out_`m`_mode`, add `SetOutput(OUT_`m`,x);` where $m \in$ `{A,B,C}`.
- if $t(n) =$ `out_`m`_direction`, add `SetDirection(OUT_`m`,x);` where $m \in$ `{A,B,C}`.
- if $t(n) =$ `out_`m`_power`, add `SetPower(OUT_`m`,x);` where $m \in$ `{A,B,C}`.
- if $t(n) =$ `ir`, add `SendMessage(x);`
- if $n \in CB_i$, add `ClearTimer(`$c_i(n)$`);`
- if $t(n) =$ `none`, add `n = x;`

The transitions of automaton `TrainAlarm` from location `on` of the example are translated as follows:

```
else if (pc_TrainAlarm == on)
{
    if (!(TrainAlarm_x<7))
        deadlock_automata();
    else if (ir==ok && pc_Train == alarm)
    {
        SetOutput(OUT_B,OUT_OFF);
        SetPower(OUT_A,7);
        SetPower(OUT_C,7);
        pc_TrainAlarm = idle;
        pc_Train = idle;
    }
    else if (TrainAlarm_light==1 && TrainAlarm_x>=5)
    {
        TrainAlarm_light = 0;
        ClearTimer(0);
        SetOutput(OUT_B,OUT_OFF);
        pc_TrainAlarm = on;
```

```
    }
    else if (TrainAlarm_light==1 && TrainAlarm_x>=5)
    {
        TrainAlarm_light = 1;
        ClearTimer(0);
        SetOutput(OUT_B,OUT_ON);
        pc_TrainAlarm = on;
    }
}
```

This fragment shows the translation of an edge that contains a synchronization. Note that the "same" transition exists for automaton `Train` in the NQC code.

For the complete translation of the Uppaal model, see the appendix.

6.7 Urgency, commitment and arrays

These aspects of an Uppaal model cannot be translated yet.

7 Restrictions on Uppaal models

In section 6 an easy and straightforward translation of Uppaal models to NQC programs is proposed. This translation gives rise to some requirements that Uppaal models must satisfy before the proposed translation is possible. In this section these requirements are identified. Note that some of these requirements can be avoided or weakened by a smarter translation. Not enough time was available for this project to construct such a smarter translation. The tool `uppaal2nqc` checks all the requirements defined in the following subsections.

Let an Uppaal model consist of a set $A = \{A_1, ..., A_n\}$ of timed automata. Then the user defines a partition $B = \{B_1, ..., B_m\}$ of this set and a type mapping t. Every block B_i of this partition (hence the *block* compiler directive, see section 5.2) will be translated to one NQC program. In the following subsections CB_i is used to denote the set $\bigcup_{A_k \in B_i} C_k$, all clocks of block B_i. Also IB_i is used to denote the set $\bigcup_{A_k \in B_i} I_k$, all integer variables of block B_i

7.1 Synchronizations

The translation of synchronizations, explained in section 6.6, assumes that automata in A that have actions in common are on the same RCX brick. If this would not be the case, the translation of the synchronization is more difficult. Thus all the blocks of the partition of A must be "closed" with respect to synchronization actions. Let B_i and B_j be blocks of the partition of A, then the following must hold:

$$a \neq \tau \wedge a \in Act_m \wedge a \in Act_n \wedge A_n \in B_i \wedge A_m \in B_j \Rightarrow i = j$$

7.2 Clocks

One RCX has four timers. This means that in one NQC program, a maximum of four clocks can be used:

$$|CB_i| \leq 4$$

Another restriction concerning the clocks, originating from the translation, is that two different blocks cannot share clocks. This is reflected by the following property:

$$CB_i \cap CB_j = \emptyset$$

The RCX has only the capability to *reset* it's clocks to zero. The straightforward translation of assignments, see section 6.6, thus requires that for every clock assignment $x := n$ in the network, n must be equal to 0.

7.3 Integer variables

In all blocks, only one variable may be used to model the IR buffers:

$$\left| \bigcup_{0 \leq k \leq n} \{\, v \,|\, v \in I_k \wedge t_i(v) = \texttt{ir} \,\} \right| \leq 1$$

The following property states that a certain block B_i cannot use more than 32 integer variables. The translation declares $|B_i|$ program counters, $|CB_i|$ clock variables and $|I_g|$ global variables. The fourth term in the following is the number of sensor value attributes used. For each such attribute a global variable is declared. The last term is the number of internal variables that are used in the block. A variable local to the main task is declared for each internal variable. The translation thus requires that the following must hold, where `X` $\in$ `{1,2,3}`:

$$\begin{aligned} 32 \geq \; & |B_i| + |CB_i| + |I_g| \\ & + \\ & |\textstyle\bigcup_{A_k \in B_i} \{\, v \,|\, v \in I_k \wedge t(v) = \texttt{sns\_X\_value} \,\}| \\ & + \\ & |\textstyle\bigcup_{A_k \in B_i} \{\, v \,|\, v \in I_k \wedge t(v) = \texttt{none} \,\}| \end{aligned}$$

The following property means that in a certain block B_i, not more than one variable is used to denote a certain input or output attribute. Thus if a blocks satisfies this property, then the NQC program that results from the translation does not use more than three inputs or three outputs. For all the following must hold:

$$\forall q \in T \backslash \{\texttt{none}, \texttt{ir}\} : \quad \left| \bigcup_{A_k \in B_i} \{\, v \,|\, t(v) = q \,,\, v \in I_k \,\} \right| \leq 1$$

7.4 Assignments and Guards

The translation of assignments possibly involves the API calls of section 2. Because of the strict syntactical forms of these calls - they cannot use variable arguments - it is easy for the translation to assume that only the "proper values", described in section 2, can be assigned to an output variable in the Uppaal model. This can only be achieved by defining these values as Uppaal constants. See for example the local declarations of template `aTrain` in the example Uppaal model in section 4.1. An assignment can be easily translated with one API call in this way. The first argument of the API call is provided by the type mapping and the second argument is the assigned value. This is exactly what has been assumed.

Let $x := a$ be an atomic assignment of an integer variable in some assignment set in some automaton in the network, then the following requirements have been identified:

$t(x) \neq$ `sns_x_value`

$t(x) =$ `sns_x_mode` $\Rightarrow$ $a \in$ { `SENSOR_MODE_RAW, SENSOR_MODE_BOOL, SENSOR_MODE_PERCENT, SENSOR_MODE_ROTATION` }

$t(x) =$ `sns_x_type` $\Rightarrow$ $a \in$ { `SENSOR_TYPE_TOUCH, SENSOR_TYPE_LIGHT, SENSOR_TYPE_TEMPERATURE, SENSOR_TYPE_ROTATION` }

$t(x) =$ `out_x_mode` $\Rightarrow$ $a \in$ { `OUT_ON, OUT_OFF, OUT_FLIP` }

$t(x) =$ `out_x_direction` $\Rightarrow$ $a \in$ { `OUT_FWD, OUT_REV, OUT_TOGGLE` }

Let $a \sim b$, where $\sim \in \{<, \leq, >, \geq, \neq, ==\}$, be a guard of some guard set in some automaton. If this is a clock guard, then b is an integer expression. If it is an integer guard, then both a and b are integer expressions. These integer expressions possibly contain integer variables and constants. For all these integer variables, denoted by n, the following must hold:

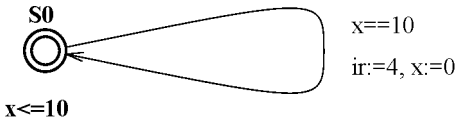
$$t(n) \in \{\texttt{none, ir, sns\_1\_value, sns\_2\_value, sns\_3\_value}\}$$

This means that the Uppaal variables that model the sensor mode and type attributes and the output mode, power and direction attributes, cannot be used in guards. This is due to the fact that these so-called output variables, see section 6.4.1, are not declared in the NQC program and the straightforward translation of guards doesn't support use of these output variables in guards in the Uppaal model.
A possibility to overcome this restriction is to use an extra internal variable in Uppaal that mimics the attribute. This has been done in the automaton `TrainAlarm`: The variable `light` mimics the variable `alarm_light`. The same result can also be achieved by encoding the value of the output variable in the control structure of the automaton.

Note that Uppaal contants that are equivalent to the NQC macro's of section 2.1, *can* be used in guards. The values of these contants in NQC may however differ from their values in the Uppaal model.

8 Relation between the model and the realization

When translating an formal model to an implementation, semantics might be changed. The timed automaton depicted below for example, can never be realized.



Because of the fact that in physical systems it is impossible to time events with infinite precision, all attempts to assign the value 4 to the variable `ir` *exactly* at time 10 would fail.

Let us now take a closer look at the general control structure of a NQC program that results from the proposed translation. It is clear that assignments, the overhead of the control structure and the reading of the input values take time. In Uppaal it is assumed that the assignments of an action transition and the decision which transition to execute can be executed without passing of time; the RCX cannot achieve such performance. This fact has a number of consequences that are explained below.

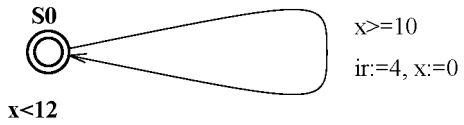
The RCX can only execute a certain, finite, number of action transitions within a finite amount of time. If the Uppaal model of the control program that runs on a certain RCX forces more action transitions within a certain amount of time than the RCX can handle, the control program will deadlock. This is related to the requirement of *nonZenoness* on the timed automata in the model necessary for executability [10].

Uppaal assumes that assignments can be executed without the elapse of time. It should be clear that this can never be realized. In very sensitive systems this might give problems. For example: If in an Uppaal model two motors are shut down at the same time, reflected by two assignments on one edge, then in the realization this will *not* happen at the same time. First one motor is shut down and a fraction of a second later the other motor is shut down.

Then there exists uncertainty about the timing of action transitions. Due to the mentioned overhead in the NQC program, always implicitly a delay transition and possibly one action transition is executed. To estimate the timing uncertainty let us assume that the body of the infinite while-loop executes within 1 RCX clock tick, that is 100 ms, *and* that the timers of the RCX are perfect. The execution returns at a certain time to the start of the loop. Let there the value of a clock be read. That value is for example n. Then it can occur that the

value of this clock is $n+1$ when the automaton following this read is enabled to execute an action transition. This behavior cannot be avoided. Typically, if an Uppaal model is in a certain location and one action transition from that location becomes enabled when a clock has value n, then this action transition occurs in the corresponding RCX timer interval $[n, n+2)$ in the realization.

If the body of the while loop is executed within 1 RCX clock tick and the RCX timers are perfect, a subset of the executions of the following model can be realized.



However if some edges in the model contain more than one assignment, the model is is only approximately realizable, as explained above.

Furthermore, it is not too difficult to construct Uppaal models that satisfy the requirements of section 7, but are not realizable. It is still the job of the user to think about the properties and consequences of the translation and to decide whether or not the realization is acceptable.

9 Validation of the translation

The experimental LEGO setup that Jeroen Kratz has build provided a ready available test case for the translation [6]. An earlier version of the Uppaal model of section 4.1 has been translated to two NQC programs. The first experimental tests of this code on the LEGO construction revealed that the model was not accurate enough. For example, extra assignments to the variables modeling the direction attribute of some actuators had to be added. After some of these minor changes to the initial model, resulting in the model of section 4.1, new experiments showed that the NQC code generated from the new model behaved like expected. These shortcomings of the model could have been foreseen if additional verification properties would have been checked. The Uppaal model and the NQC programs that result from it are available at the project web site [8].

10 Conclusion

This project presents a simple translation from Uppaal models of real-time controllers to NQC programs. The modeling of these controllers in Uppaal provides a way to verify the requirements on these controllers. The user directs the translation by defining a type for each variable used in the model and by assigning each automaton in the model to a controller.

The translation, that has been implemented in the tool `uppaal2nqc`, results in a set of NQC programs that, when all NQC programs are run concurrently, approximately realizes a subset of the executions of the model. An Uppaal model of controllers of an experimental LEGO setup has been translated and the resulting NQC programs have been run in this setup to validate the translation.

The proposed translation is very platform dependent. What the user essentially must do is model the desired behavior of the various RCX bricks in the system. This model is then annotated with compiler directives to facilitate the realization of the model on these RCX bricks. There exist however numerous modeling pitfalls that can only be avoided if the user knows the limitations of the translation very well. The validation of the translation showed that as much verification properties as possible should be checked. If that is done, some problems, like the initialization problems that were encountered during the validation of the translation, might be discovered.

An open issue is that the exact relation between the Uppaal models and the NQC programs resulting from the translation, is not clear. Future research may therefore focus on this relation. The verification of the Uppaal models is thus not very useful, because no exact link between the model and the realization has been established. Iversen et al presented a method for automatic verification of real-time control programs running on the LEGO RCX brick using Uppaal [9]. They constructed the `rcx2uppaal` compiler that uses hardware specific properties of the RCX to construct an Uppaal model of a RCX byte code control program. Future research may focus on the combination of these two compilers. This combinatian might provide a way to model real-time controllers in Uppaal, generate LEGO control programs from these models and finally, verify the LEGO control programs.

Acknowledgements

I thank Frits Vaandrager for the very useful discussions and suggestions and for the reviews of this report and intermediate reports. I also thank Jozef Hooman for several reviews of this report and general comments about conducting research, writing papers and presenting them.

11 References

[1] Thomas Hune, Kim G. Larsen & Paul Pettersson, *Guided Synthesis of Control Programs Using* UPPAAL, In Proceedings of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation, DSVV'2000. This paper can be found at: `http://www.brics.dk/~baris/Papers/index.html`

[2] Kim G. Larsen, Paul Pettersson & Wang Yi, *Uppaal in a Nutshell*, Int. Journal on Software Tools for Technology Transfer 1(1-2), pages 134-152. Springer-Verlag. 1998. This paper can be found at `http://www.uppaal.com/`

[3] Information about the modeling, validation and verification tool Uppaal can be found at `http://www.uppaal.com/`

[4] More information about LEGO Mindstorms can be found at:
`http://mindstorms.lego.com/`

[5] Dave Baum, *NQC Programmer's Guide, Version 2.2 r1.* This document can be found at: `http://www.enteract.com/~dbaum/nqc/index.html`

[6] Jeroen Kratz's report on the LEGO railroad crossing at the KUN:
`http://www.cs.kun.nl/ita/voorlichting/report.ps.gz`

[7] R. Alur, *Timed Automata*, in NATO-ASI Summer School on Verification of Digital and Hybrid Systems, 1998.

[8] The website of this project can be found at:
`http://www.sci.kun.nl/infstud/~martynhe/RL1.html`

[9] Iversen et al., *Model-checking Real-Time Control Programs — Verifying LEGO Mindstorms Systems Using Uppaal*, Proc. of 12th Euromicro Conference on Real-Time Systems, 147-155, IEEE Computer Society Press, June 2000.

[10] M. Abadi and L. Lamport, *An old-fashioned recipe for real time*, in Real-Time: Theory in Practice, REX Workshop, LNCS 600, pages 1-27. Springer Verlag, 1991.

Appendix:

This is the file `Gate.nqc` that results from compilation of the Uppaal model defined in section 3 by the tool `uppaal2nqc`.

```
/*    This file has been generated by uppaal2nqc version 1
 *    Block Gate includes:
 *            Gate
 */


/* The constant(s) of block Gate: */
#define Gate_TR_APP  40
#define Gate_GT_CL   45
#define failure       1
#define ok            2

/* The location(s) of block Gate: */
#define idle                 1
#define wait_for_car         2
#define wait_to_close        3
#define closed               4
#define not_closed           5
#define opening              6
#define open_gate_init       7
#define wait_train_to_leave  8
#define deadlocked           9


/* Declaration of the clock(s): */
int Gate_z;

/* Declaration of input variable(s): */
int sns1v, sns2v, sns3v;

/* The IR variable: */
int ir;

/* Declaration and initialization of the program counter(s): */
int pc_Gate = open_gate_init;
```

```
sub read_input()
{
    /* Read the rcx timer(s): */
    Gate_z = Timer(0);

    /* Read sensor value(s): */
    sns1v = SensorValue(0);
    sns2v = SensorValue(1);
    sns3v = SensorValue(2);

    /* Read the IR value: */
    ir = Message();
}

sub deadlock_automata()
{
    pc_Gate = deadlocked;
    SetOutput(OUT_A,OUT_OFF);
    SetOutput(OUT_B,OUT_OFF);
    SetOutput(OUT_C,OUT_OFF);
}

task main()
{
    /* Initialize output variable(s): */
    SetDirection(OUT_A,OUT_FWD);
    SetPower(OUT_A,2);
    SetOutput(OUT_A,OUT_ON);
    SetPower(OUT_B,7);
    SetOutput(OUT_B,OUT_OFF);
    SetPower(OUT_C,7);
    SetOutput(OUT_C,OUT_OFF);
    SetSensorType(SENSOR_1,SENSOR_TYPE_TOUCH);
    SetSensorType(SENSOR_2,SENSOR_TYPE_LIGHT);
    SetSensorType(SENSOR_3,SENSOR_TYPE_LIGHT);
    SetSensorMode(SENSOR_1,SENSOR_MODE_PERCENT);
    SetSensorMode(SENSOR_2,SENSOR_MODE_PERCENT);
    SetSensorMode(SENSOR_3,SENSOR_MODE_PERCENT);

    /* Reset the timer(s) to zero: */
    ClearTimer(0);
```

```
while(1)
{
    /* Transitions for automaton Gate: */
    read_input();
    if (pc_Gate == idle)
    {
        if (sns2v<=Gate_TR_APP)
        {
            ClearTimer(0);
            SetOutput(OUT_B,OUT_ON);
            SetOutput(OUT_C,OUT_ON);
            pc_Gate = wait_for_car;
        }
        else if (sns3v<=Gate_TR_APP)
        {
            ClearTimer(0);
            SetOutput(OUT_B,OUT_ON);
            SetOutput(OUT_C,OUT_ON);
            pc_Gate = wait_for_car;
        }
    }
    else if (pc_Gate == wait_for_car)
    {
        if (!(Gate_z<52))
            deadlock_automata();
        else if (Gate_z>=50)
        {
            SetDirection(OUT_A,OUT_REV);
            SetOutput(OUT_A,OUT_ON);
            ClearTimer(0);
            pc_Gate = wait_to_close;
        }
    }
    else if (pc_Gate == wait_to_close)
    {
        if (!(Gate_z<27))
            deadlock_automata();
        else if (sns1v>=Gate_GT_CL)
        {
            SetOutput(OUT_A,OUT_OFF);
            pc_Gate = closed;
        }
```

```
    else if (Gate_z>=25 && sns1v<Gate_GT_CL)
    {
        SendMessage(failure);
        SetOutput(OUT_A,OUT_OFF);
        pc_Gate = not_closed;
    }
}
else if (pc_Gate == closed)
{
    if (sns2v<=Gate_TR_APP)
    {
        ClearTimer(0);
        SetDirection(OUT_A,OUT_FWD);
        SetOutput(OUT_A,OUT_ON);
        pc_Gate = opening;
    }
    else if (sns3v<=Gate_TR_APP)
    {
        ClearTimer(0);
        SetDirection(OUT_A,OUT_FWD);
        SetOutput(OUT_A,OUT_ON);
        pc_Gate = opening;
    }
}
else if (pc_Gate == not_closed)
{
    if (sns2v<=Gate_TR_APP)
    {
        ClearTimer(0);
        SendMessage(ok);
        SetDirection(OUT_A,OUT_FWD);
        SetOutput(OUT_A,OUT_ON);
        pc_Gate = opening;
    }
    else if (sns3v<=Gate_TR_APP)
    {
        ClearTimer(0);
        SendMessage(ok);
        SetDirection(OUT_A,OUT_FWD);
        SetOutput(OUT_A,OUT_ON);
        pc_Gate = opening;
    }
}
```

```
        else if (pc_Gate == opening)
        {
            if (!(Gate_z<22))
                deadlock_automata();
            else if (Gate_z>=20)
            {
                ClearTimer(0);
                SetOutput(OUT_A,OUT_OFF);
                SetOutput(OUT_B,OUT_OFF);
                SetOutput(OUT_C,OUT_OFF);
                pc_Gate = wait_train_to_leave;
            }
        }

        else if (pc_Gate == open_gate_init)
        {
            if (!(Gate_z<22))
                deadlock_automata();
            else if (Gate_z>=20)
            {
                SetOutput(OUT_A,OUT_OFF);
                pc_Gate = idle;
            }
        }
        else if (pc_Gate == wait_train_to_leave)
        {
            if (!(Gate_z<202))
                deadlock_automata();
            else if (Gate_z>=200)
            {
                pc_Gate = idle;
            }
        }
    }
}
```

This is the file `Train.nqc` that results from compilation of the Uppaal model defined in section 3 by the tool `uppaal2nqc`.

```
/*    This file has been generated by uppaal2nqc version 1
 *    Block Train includes:
 *            Train
 *            TrainAlarm
 */


/* The constant(s) of block Train: */
#define failure   1
#define ok        2

/* The location(s) of block Train: */
#define idle        1
#define alarm       2
#define on          3
#define deadlocked  4


/* Declaration of the clock(s): */
int TrainAlarm_x;

/* The IR variable: */
int ir;

/* Declaration and initialization of the program counter(s): */
int pc_Train = idle,
    pc_TrainAlarm = idle;


sub read_input()
{
    /* Read the rcx timer(s): */
    TrainAlarm_x = Timer(0);

    /* Read the IR value: */
    ir = Message();
}
```

```
sub deadlock_automata()
{
    pc_Train = deadlocked;
    pc_TrainAlarm = deadlocked;
    SetOutput(OUT_A,OUT_OFF);
    SetOutput(OUT_B,OUT_OFF);
    SetOutput(OUT_C,OUT_OFF);
}

task main()
{
    /* Declaration of local variable(s): */
    int TrainAlarm_light = 0;

    /* Initialize output variable(s): */
    SetPower(OUT_A,7);
    SetPower(OUT_C,7);
    SetDirection(OUT_A,OUT_REV);
    SetDirection(OUT_C,OUT_REV);
    SetOutput(OUT_A,OUT_ON);
    SetOutput(OUT_C,OUT_ON);
    SetOutput(OUT_B,OUT_OFF);

    /* Reset the timer(s) to zero: */
    ClearTimer(0);

    while(1)
    {
        /* Transitions for automaton Train: */
        read_input();
        if (pc_Train == idle)
        {
            if (ir==failure && pc_TrainAlarm == idle)
            {
                SetPower(OUT_A,3);
                SetPower(OUT_C,3);
                SetOutput(OUT_B,OUT_ON);
                ClearTimer(0);
                TrainAlarm_light = 1;
                pc_Train = alarm;
                pc_TrainAlarm = on;
            }
        }
```

```
else if (pc_Train == alarm)
{
    if (ir==ok && pc_TrainAlarm == on)
    {
        SetPower(OUT_A,7);
        SetPower(OUT_C,7);
        SetOutput(OUT_B,OUT_OFF);
        pc_Train = idle;
        pc_TrainAlarm = idle;
    }
}

/* Transitions for automaton TrainAlarm: */
read_input();
if (pc_TrainAlarm == idle)
{
    if (ir==failure && pc_Train == idle)
    {
        SetOutput(OUT_B,OUT_ON);
        ClearTimer(0);
        TrainAlarm_light = 1;
        SetPower(OUT_A,3);
        SetPower(OUT_C,3);
        pc_TrainAlarm = on;
        pc_Train = alarm;
    }
}
else if (pc_TrainAlarm == on)
{
    if (!(TrainAlarm_x<7))
        deadlock_automata();
    else if (ir==ok && pc_Train == alarm)
    {
        SetOutput(OUT_B,OUT_OFF);
        SetPower(OUT_A,7);
        SetPower(OUT_C,7);
        pc_TrainAlarm = idle;
        pc_Train = idle;
    }
```

```
            else if (TrainAlarm_light==1 && TrainAlarm_x>=5)
            {
                TrainAlarm_light = 0;
                ClearTimer(0);
                SetOutput(OUT_B,OUT_OFF);
                pc_TrainAlarm = on;
            }
            else if (TrainAlarm_light==0 && TrainAlarm_x>=5)
            {
                TrainAlarm_light = 1;
                ClearTimer(0);
                SetOutput(OUT_B,OUT_ON);
                pc_TrainAlarm = on;
            }
        }

    }
}
```