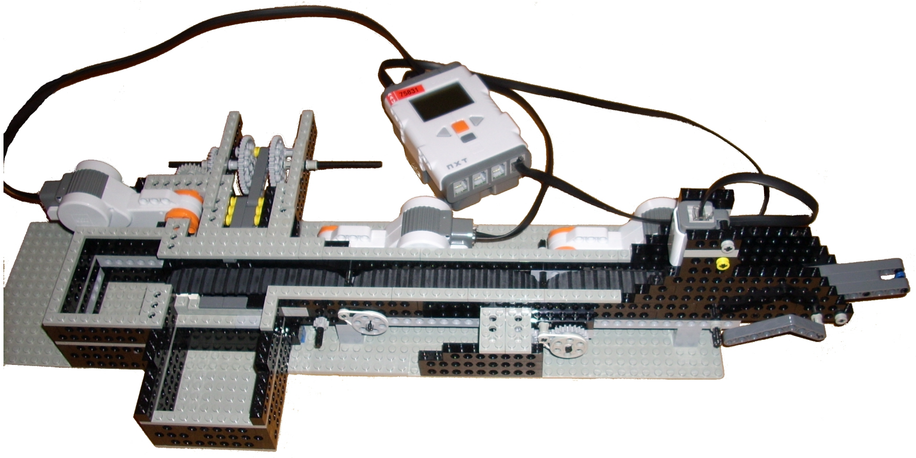
From games to executables!

Implementations of strategies generated from UPPAAL TIGA

by

Kenneth Blanner Holleufer
and
Jesper Brix Rosenkilde



THESIS

for the degree of

MASTER OF SCIENCE

(Master of computer science)

Department of Computer Science
Faculty of Engineering, Science and Medicine
Aalborg University

June 2009

Summary

This master thesis is written by Kenneth Blanner Holleufer and Jesper Brix Rosenkilde during the fall semester of 2009 at the Department of Computer Science at Aalborg University. The subject of the thesis is generating executable strategies synthesised by the tool UPPAAL TIGA.

The first chapter contains an introduction to various approaches to control theory. First a general introduction is given for people unfamiliar with the subject. Then we go in to a bit more detail with input-output modelling. This is the clasical way to see control problems, which is as a system of differential equations. This naturally leads to hybrid systems, which is the combination of the way control people view the world, which is as continious systems, and the way computer scientist se the world, which is as discrete systems. Finally discrete event systems are introduced as the type of sytstem which we can represent with timed automata.

The second chapter gives a quick introduction to timed automata and networks of them. Timed automata is the formalismen used in the UPPAAL-family of model checkers, which we are using. Timed automata can be used to model real-time systems. A thorough overview is given to syntax and sematics of timed automata. Since timed automata are used to model systems with time, which is defined over the positive real numbers, the state space for these models is uncountably infinite. To remedy this region graphs and zones are introduced, these contructions allow the infinite state space to be represented finitely by using symbolic states. A symbolic state consists of a discrete part, the location and variables, and a zone representing the clock valuations.

The third chapter takes the formalism from chapter two, and extends it to enable control synthesis. This is done by introducing the notion of real-time games. These are two player games, where a controller tries to reach or stay within a set of safe states in a timed automata, while the environment tries to prevent this. To facilitat this the transitions of the timed automata are divided into two, the controllable and the uncontrollable. The controller makes its moves according to a given strategy which, for the game to be winable, must ensure that no matter what the oppont does at any time does not lead to any bad states. The question is, how does one produce such a strategy, if it exist? This is answered by giving an algorithm for, and discussing how it is implemented in the tool UPPAAL TIGA. A common problem is strategies which produce Zeno-runs, this where a process is forced to do an infinite number of discrete transitions in a finite amount of time. A construction to avoid these types of strategies, by changing the model, is given. Finally a prunning algorithm which removes unreachable states in strategies generated by UPPAAL TIGA is given. This problem can occur because TIGA uses a backpropagation algorithm,

and therefor all states in a strategy might not be reachable through forward exploration under the strategy.

The fourth chapter is devoted to finding out what effect different modelling tricks have on strategy size, generation time and state space size. The basis for the tests is a model of a brick-sorter. The brick-sorter is build with LEGO, and consists of a convyour belt which moves bricks along. At some point the bricks colour is determined by a sensor, and further down the line the brick passes a piston, which acts according to the bricks colour, by either pushing it of or letting it pass. The first model of the system assumes that everything about the system is known, for example precisely where every brick is at all time. This is then refiend by abstracting away information leading to a model based on partial observability. This model is then changed further into two models, one using symetry reduction and another where cycles are removed. These two models are then combined into a sequential and acyclic model. Finnally this model is made discrete. All of these model are then compared to see which is best in terms of states space size, generation time and strategy size.

In the fith chapter we implement a method for taking a strategy generated by UPPAAL TIGA, and use it to controll the brick-sorter build with LEGO. To do this a compact data structure is needed to hold a strategy. This is accomplished using clock difference diagrams combined with binary decision diagrams. These two things are combind in a way, which makes is possible to reasson about at which times and locations, a specific action is to be taken. The brick-sorter is controlled by a LEGO NXT brick, so we have made a Java program which is able to hold a strategy and evaluate it. Finally to connect TIGA and the brick we have written a parser/codegenerator, which takes the output from TIGA and produceses the code needed to initialise a strategy on the brick. All this enables us to take a model in UPPAAL TIGA, change the number of brick processes, or even the property, and automatically have the brick-sorter execute the modelled behaviour.

Contents

Chapter 1

Introduction

Historically, scientists and engineers have concentrated on studying and harnessing natural phenomena which are well-modelled by the laws of gravity, classical and non-classical mechanics, physical chemistry etc. Because of this the systems deal with quantities such as displacement, velocity, and acceleration of particles and rigid bodies, or the pressure, temperature, and flow rates of fluids and gases. These are "continuous variables" in the sense that they can take on any real value as time itself "continuously" evolve. As a result the study of ordinary and partial differential equations currently provides the main infrastructure for system analysis and control. However in the technological and increasingly computer-dependent world, two things can be noted. First, many of the quantities dealt with are "discrete" typically involving natural numbers. (E.g., how many items are on a conveyor belt, how many telephone calls are active). And second, what drives many of the processes we use and depend on are instantaneous "events" such as a sensor registering that something has passed it, pushing a button, or hitting a keyboard key. This typically make such things as communication networks, manufacturing facilities or execution of a computer program event-driven.

A most interesting scenario is controlling a system through a model. The model describes the system, and some sort of strategy is applied to this, in order exhibit the desired behaviour. This can be done by looking at the problem as a game, where the environment and a controller are opponents, each trying to defeat the other. For the controller to win, its strategy must ensure that the proper behaviour is enforced, no matter what the environment does. The model checking tool UPPAAL TIGA is able to synthesis strategies using exactly this method and timed automata.

As an example, suppose we have a system that consists of a conveyor belt, a colour sensor and a piston. Different coloured bricks travel down the belt, and based on the colour that the sensor reads the brick is either pushed off the belt or allowed to travel to the end of the belt. It is possible to model this system and control it with strategy, which guarantees all black bricks are pushed off the belt or similar. An implementation of the system, based on this strategy, can then be produced. But what if we want to push all the white bricks off the belt the next day. Ordinarily a new implementation is needed for the real-world system, but in the model, only the property would need to be changed for UPPAAL TIGA to output the new strategy. So the question is, is it possible to take these strategies and automatically transfer them to a real life system? Such a function would allow easy reprogramming of complex systems and add a high degree of flexibility.

1.1 Input-Output Modelling Process

We are interested in the control of a system, which should be based on well-defined criteria. Therefore a model of an actual system is necessary. Intuitively, a model is a construction that directly reflects the behaviour of the system itself. Input-Output Modelling is a mathematical process to describe the behaviour of the system. What follows is a short description of the traditional method of modelling a system, namely with differential or difference equations. The rest of this chapter is based on [CL08].

To carry out the modelling process, we define a set of measurable variables associated with a given system. For example, particle positions and velocities, or voltages and currents in an electrical circuit, which are all real numbers. By measuring these variables over a period of time $[t_0, t_f]$ data may be collected. Next a subset of these variables is selected, and it is assumed that they can be varied over time. This defines a set of time functions which are called the input variables,

$$\{u_1(t), \ldots, u_p(t)\}, \quad t_0 \leq t \leq t_f.$$

Then another set of variables which can be directly measured while varying the input variables, are selected. The resulting variables are called the output variables,

$$\{y_1(t), \ldots, y_m(t)\}, \quad t_0 \leq t \leq t_f.$$

The output variables can be thought of as describing the "response" of the "stimulus" provided by the selected input functions. Let $\mathbf{u}(t)$ and $\mathbf{y}(t)$ represent the input and output variables as row vectors.

To complete the model, it is reasonable to postulate that there exists some mathematical relationship between the input and output. Thus, there exist functions mapping from one to the other:

$$y_1(t) = g_1(u_1(t), \ldots, u_p(t)), \ldots, y_m(t) = g_m(u_1(t), \ldots, u_p(t)).$$

This gives the model of a system as:

$$\mathbf{y} = \mathbf{g}(\mathbf{u}).$$

This is a very simple possible modelling process, and is illustrated in Figure 1.1 on the following page.

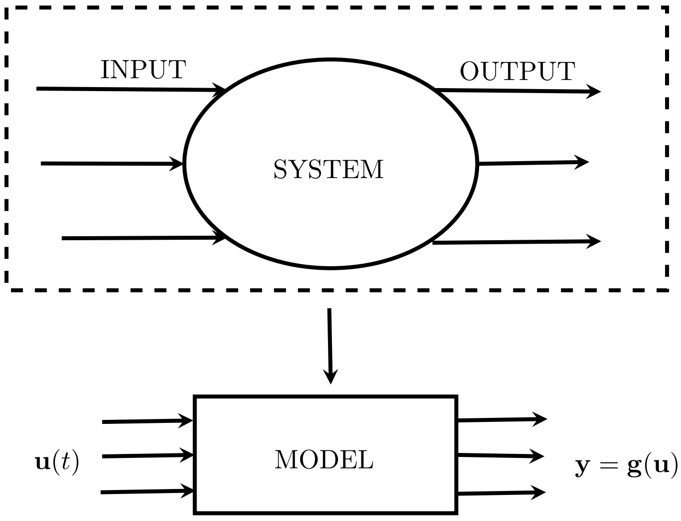


Figure 1.1: Simple modelling process.

1.2 States

The state of a system at a time t should describe its behaviour at that instant in some measurable way.

Definition 1.2.1 (State)
The state of a system at time t_0 is the information required at t_0 such that the output $\mathbf{y}(t)$, for all $t \geq t_0$, is uniquely determined from this information and from $\mathbf{u}(t), t \geq t_0$.

Like the input $\mathbf{u}(t)$ and the output $\mathbf{y}(t)$, the state can be expressed as a vector, $\mathbf{s}(t)$.

Definition 1.2.2 (State space)
The state space of a system, denoted by S, is the of all possible values that the state may take (total number of different states in the system)

The state space can be either continuous or discrete. In continuous-state models the state space S is a continuum consisting of all n-dimensional vectors over real (or sometimes complex) numbers. The behaviour of the system can then be described with a system of differential equations with $\mathbf{u}(t)$, $\mathbf{s}(t)$, $\mathbf{y}(t)$ with t as variable. The model can then be studied by analysing these differential equations.

In discrete models the state space is a discrete set. We usually represent a single state in this set with s. The dynamic behaviour of discrete-state systems is often simpler to visualise. This is because the transition mechanism is normally based on simple logical statements of the form "if something specific happens and the current state is s, then the next state becomes s'."

1.3 Hybrid systems and timed automata

A hybrid system is a modelling structure that consists of discrete states and arbitrary continuous state variables with their own time dynamics. Let $\mathbf{x}$ represent a continuous state vector and let q be a discrete state. Thus the model state is expressed as $(q, \mathbf{x})$ where $q \in Q$ and $\mathbf{x} \in X$. The set of discrete states is Q and the set of continuous states is X. The state $\mathbf{x}$ evolves according to time driven dynamics which are usually described by some differential equation such as $\mathbf{x}' = f(\mathbf{x})$ with a given initial condition $\mathbf{x}_0$. Discrete state transitions can occur when some condition in a state $\mathbf{x}(t)$ is satisfied.

Example 1.3.1
Let us model a simple thermostat. Let $Q = \{OFF, ON\}$ represent the discrete state of the thermostat. The continuous state is the scaler $x \in \mathbb{R}$ and it models the temperature of the room. We want to control the thermostat to maintain a room temperature between 20 and 25 degrees Celsius. Then the condition for switching the thermostat OFF is $x \geq 25$, and the condition for switching to ON is $x \leq 20$. This operation can be modelled though a hybrid automaton, as shown in Figure 1.2. Each discrete state is accompanied by some time-driven dynamics. When $q = ON$, we have $x' = -x + 30$ indicating that the heating system's capacity is such that it can drive the room temperature up to 30 degrees Celsius (when $x = 30$, we have $x' = 0$). When $q = OFF$, we have $x' = -x + 15$ indicating that in the absence of any heating action the room temperature would become 15 degrees. Specifying an initial condition (q_0, x_0) completes the model.

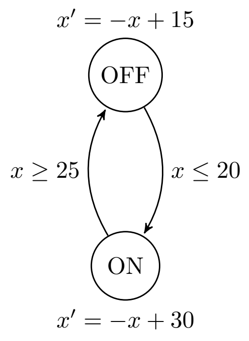


Figure 1.2: A simple thermostat as a hybrid system.

Going a step further away from continuous systems, timed automata consists of finite number of discrete states and number of time driven clocks. If we let Q represent the discrete states, also known as locations, and let $v(\mathbf{x})$ represent the value of the clock vector $\mathbf{x}$. States are then expressed as $(q, v(\mathbf{x}))$ where $q \in Q$. Transitions of the automaton include conditions on the clock values known as guards and resetting the clocks. Transitions between the discrete states can only occur if the condition on the guard it met. The discrete states of the timed automaton can also include conditions on the clocks known as invariants. Staying in a discrete state is only allowed if the condition on the invariant it met.

Example 1.3.2
Since the continuous variables of a hybrid automata depends on time the desired behaviour of the thermostat from example 1.3.1 can be modelled by a timed automata. Again, let $Q = \{OFF, ON\}$ represent the discrete states. Furthermore let Δ be the time is takes to cool the room 5 degrees Celsius when the thermostat is off, and ∇ is the time it takes to heat 5 degrees when it is on. Suppose the room is 20 degrees when the thermostat is activated, then the initial state is $(ON, 0)$. The condition for switching off is then waiting exactly ∇ time units no more and no less. This is expressed by the guard $x = \nabla$ and the invariant $x \leq \nabla$. Similarly, the condition for switching on is waiting Δ time units, expressed by $x = \Delta$ and $x \leq \Delta$. The timed automaton can be seen in Figure 1.3

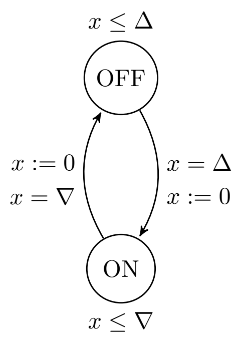


Figure 1.3: A simple thermostat as a timed automata.

1.4 Discrete Event Systems

When the state space of a system can naturally be described by a discrete set, and state transitions only occur at discrete points in time, and the state transitions are then associated with "events", then we can talk about a "discrete event system". An event or an action a, as we will call it through most of this report, should be thought of as occurring instantaneously and causing transitions from one state to another. An event may be identified with a specific action taken (e.g., somebody pressing a button). It may be viewed as spontaneous occurrence dictated by nature (e.g., a conveyor belt breaks down for whatever reason). Or it might be the result of conditions, which are suddenly all met. The set of all events in the system is *Act*.

In the systems we are interested in modelling, an event will at various time instants announce that it is occurring. This means that every event defines a distinct process through which the time instants, when the event occurs, are determined. State transitions are the result of combining these asynchronous and concurrent event processes. These processes need not be independent of each other. This is called an event-driven system.

Definition 1.4.1 (DES)
A Discrete Event System is a discrete-state, event-driven system, that is, its state evolution depends entirely in the occurrence of asynchronous discrete events/actions over discrete time.

Consider the following timed sequence of actions and the time of their occurrence in a model.

$$(a_1, t_1), (a_2, t_2), (a_3, t_3), (a_4, t_4), (a_5, t_5), (a_6, t_6), (a_7, t_7)$$

The first action is a_1 and it occurs at time t_1; the second action is a_2 and it occurs at time t_2, and so forth. Suppose the initial state of this sequence is s_0, and the system is deterministic. Then, from such a sequence of actions, it is possible to recover the state of the system at any point in time, in the sequence.

Consider the set of all timed sequences of actions, that a given system can ever execute. This set is called the timed language model of the system, and is denoted by $\mathcal{L}$. It is a language because the set of actions *Act*, can be thought of as an "alphabet" and the (finite) sequences of actions as "words".

The timing of the sequences is interesting, in contrary to untimed sequences where only actions are present, because it can answer questions such as: "How much time does the system spend in a specific state?" or "How soon can a particular state be reached?" These and related questions are often crucial parts of design specifications.

A language may be thought of as an formal way describing the behaviour of a DES. It specifies all admissible sequences of actions that the DES is capable of "processing" or "generating". A timed automata defines a language and can be manipulated through well-defined operations, so it is possible to construct, and subsequently manipulate and analyse languages. This is done by marking the transitions of the automata with the set *Act* and saying that every possible sequence through the automata generates the language. The language generated by the timed automaton $\mathcal{A}$ is denoted $\mathcal{L}(\mathcal{A})$.

We will not dwell deeper into languages, other than make this connection between discrete event systems, languages and timed automata.

The structure of the report is as follows. In Chapter 2 timed automata will be fully explained and defined. In Chapter 3 we will look into how a timed game automaton can be controlled with a strategy in order to ensure a given set of specifications. Essentially we will find out how to synthesise a controller for a system. In Chapter 4 we will present a model made in UPPAAL TIGA, and several methods to reduce the size of the strategies. In Chapter 5 one of these models will be used to compute strategies in UPPAAL TIGA. Lastly a tool for taking the generated strategies and translating them into code which can run on a LEGO NXT brick, and control a real life system.

Chapter 2

Timed Automata

In discrete event systems the outcome of the actions can depend on their timing, as performing an action “now” or “later” might have completely different consequences. To model such behaviour we can use timed automata.

Timed automata are nondeterministic finite automata equipped with a finite number of real valued clocks whose values grow continuously. Transitions in a timed automaton can be constrained with clock values and are also able to reset clocks. Clock constraints on transitions are referred to as guards, whereas constraints on locations are referred to as invariants. Most of this chapter is based on [LAS07].

2.1 Syntax of timed automata

Let the finite set $C = \{x_1, x_2, \dots, x_n\}$ represent the clock names used in the automaton.

Definition 2.1.1 (Clock constraint)
The set $\mathcal{B}(C)$ of clock constraints over the set of clocks C is defined by the abstract syntax

$$g ::= x \bowtie n | x - y \bowtie n | g_1 \wedge g_2,$$

where $x, y \in C$ is a clock, $n \in \mathbb{N}$ and $\bowtie \in \{\leq, <, =, >\geq\}$.

Each clock in C records the amount elapsed from the last reset. This is expressed as a function $v : C \to \mathbb{R}_{\geq 0}$, called a clock valuation. The value of a clock is denoted by $v(x)$. There are two operations used to manipulate clock valuations, *delay* and *reset*. Let v be a clock valuation. The delay operation $v + d$ gives a clock valuation, where the value of every clock is increased by the positive real number d.

Definition 2.1.2 (Delay)
For each $d \in \mathbb{R}_{\geq 0}$, the valuation $v + d$ is defined by

$$(v + d)(x) = v(x) + d, \forall x \in C.$$

For a subset r of clocks, the reset operation $v[r \leftarrow 0]$ gives a clock valuation, where the values of clocks from r are set to zero and the other clocks remain unchanged.

Definition 2.1.3 (Reset)
For each $r \subseteq C$, the valuation $v[r \leftarrow 0]$ is defined by:

$$v[r \leftarrow 0](x) = \begin{cases} 0, & \text{if } x \in r, \\ v(x), & \text{otherwise.} \end{cases}$$

Because of the notions of clock constraints and clock valuations, it is possible to define when a clock constraint satisfies a given valuation.

Definition 2.1.4 (Constraint satisfaction)
Let $g \in B(C)$ be a clock constraint for a given set of clocks C and let $v : C \rightarrow R_{>0}$ be a clock valuation. Evaluation of clock constraints ($v \models g$) is defined on the structure of g by:

$$\begin{aligned} v \models x \bowtie n &\Leftrightarrow v(x) \bowtie n \\ v \models x - y \bowtie n &\Leftrightarrow v(x) - v(y) \bowtie n \\ v \models g_1 \wedge g_2 &\Leftrightarrow v \models g_1 \text{ and } v \models g_2 \end{aligned}$$

where $x, y \in C$, $n \in \mathbb{N}$, $g_1, g_2 \in B(C)$ and $\bowtie \in \{\leq, <, =, >, \geq\}$.

With all of the above in place it is now possible to define a timed automaton.

Definition 2.1.5 (Timed automaton)
A timed automaton is a 6-tuple $\mathcal{A} = (Act, C, L, l_0, \delta, I)$ where:

- *Act is a finite set of actions.*
- *C is a finite set of clocks.*
- *L is a finite set of locations.*
- *l_0 is the initial location.*
- *$\delta \subseteq L \times B(C) \times Act \times 2^C \times L$ is a finite set of transitions.*
- *$I : L \rightarrow B(C)$ assigns invariants to locations.*

Instead of $(l, g, a, r, l') \in \delta$, transitions are written as $l \xrightarrow{g,a,r} l'$, where l is the source state, g is the guard, a is the action, r is the set of clocks to be reset and l' is the target location.

Example 2.1.1
Here is an example of a timed automaton. Figure 2.1 shows a model of an item on a conveyor belt. In the initial state the item is placed on the belt and moves to a sensor. This takes between 80 and 90 time units. Therefore there is an transition with the guard $x \geq 80$ and action `move` between the `ON` and `SEN` locations. To force the item to the `SEN` location an invariant $x \leq 90$ is placed on the `ON` location. When the transition is taken the clock x is reset. At the sensor three tings can happen: Either the sensor sees the item as red or blue and the item immediately goes into the `RED` or `BLU` locations. The immediate transition occurs because of the invariant $x \leq 0$. Or, if an error occurs in the sensor the item is immediately placed at the start of the belt and x is reset. In the location `RED` the item moves to the end of the belt, which takes between 30 and 40 time units. In the location `BLU` the item will reach a piston along the belt, and be pushed off. This takes between 10 and 20 time units.

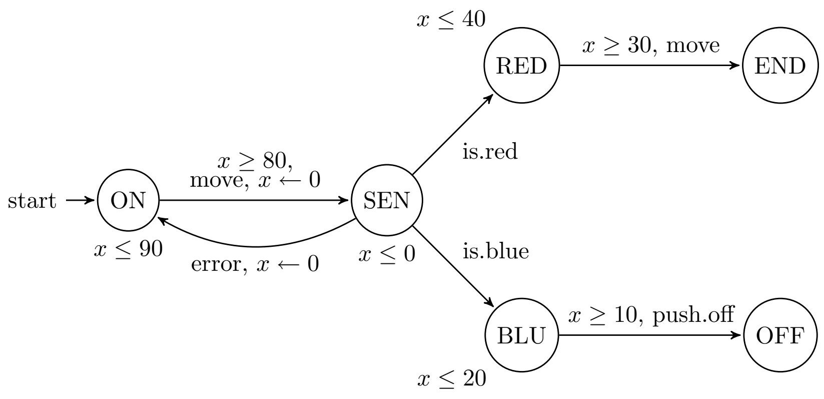


Figure 2.1: A model of an item being sorted.

2.2 Semantics of timed automata

A state of a timed automaton $\mathcal{A}$ is a pair (l, v), where l is the current location the automaton is in, and v is the valuation determined by all the current clock values. The state is legal only if the valuation v satisfies the invariant of l. A transition is enabled, and can thus be taken, if its source location matches the current location l and its guard is satisfied by the current valuation v. If the transition is taken, the target location of the transition then becomes the current, and all the clocks on the transition are reset. It is also possible to delay in the current location by increasing the value of all the clocks by an amount of time d. This does not change the location and is only possible if the invariant of the current location is satisfied by the valuation $v + d$.

Thus a timed transition system generated by a given timed automaton $\mathcal{A}$ is defined as:

Definition 2.2.1 (Timed transition system)
Let $\mathcal{A} = (Act, C, L, l_0, \delta, I)$ be a timed automaton. The timed transition system $T(\mathcal{A})$ generated by $\mathcal{A}$ is defined as $T(\mathcal{A}) = (S, Lab, \{\xrightarrow{a} | a \in Lab\})$ where:

- *$S = \{(l,v) | (l,v) \in L \times (C \to \mathbb{R}_{\geq 0})$ and $v \models I(l)\}$ is the set of states.*
- *$Lab = Act \cup \mathbb{R}_{\geq 0}$ is the set of labels.*
- *The transition relation is defined as:*
  - *$(l,v) \xrightarrow{a} (l',v')$*
    if there is a transition $(l \xrightarrow{g,a,r} l') \in \delta$, s.t. $v \models g \wedge I(l)$, $v' = v[r \leftarrow 0]$ and $v' \models I(l')$.
  - *$(l,v) \xrightarrow{d} (l, v+d)$*
    for all d' where $0 \leq d' \leq d$ and $v + d' \models I(l)$.

$(l,v) \xrightarrow{a} (l',v')$ is called a discrete transition and $(l,v) \xrightarrow{d} (l,v+d)$ is called a timed transition. Let v_0 denote the valuation such that $v_0(x) = 0$ for all $x \in C$. If v_0 satisfies the invariant of the initial location l_0, then (l_0, v_0) is called the initial state of $T(\mathcal{A})$.

$a \in Act$ is said to be enabled in the state (l,v) if there exists another state, (l',v'), such that $(l,v) \xrightarrow{a} (l',v')$. λ (a symbol used to express time elapsing) is enabled in (l,v) if (l',v') exists and $d > 0$ such that $(l,v) \xrightarrow{d} (l',v')$.

Example 2.2.1
Let $\mathcal{A}$ be the timed automaton in Figure 2.2 where there is one transition with action a, guard $x \leq 1$ and reset $x \leftarrow 0$. The single location l_0 has the invariant $x \leq 2$.

Figure 2.3 on the next page represents a small part of the transition system $T(\mathcal{A})$. From the state $(l_0, [x = 1.4])$ it is not possible to perform a transition under the action a, and the state $(l_0, [x = 2])$ has, as the only available transition, a time-elapsing transition with time delay 0.

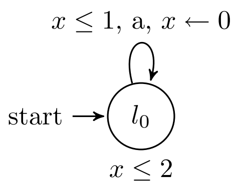


Figure 2.2: A simple model.

$$\begin{tikzcd}
(l_0, [x = 0]) \arrow[loop above, "a"] \arrow[r, "0.7"] & (l_0, [x = 0.7]) \arrow[l, bend right, "a"'] \arrow[r, "0.3"] & (l_0, [x = 1]) \arrow[ll, bend left, "a"] \arrow[r, "0.4"] & (l_0, [x = 1.4]) \arrow[r, "0.6"] & (l_0, [x = 2]) \arrow[loop above, "0"]
\end{tikzcd}$$

Figure 2.3: A timed transition system for Figure 2.2 on the preceding page.

2.3 Networks of timed automata

Many real-world systems consist of a number of independent components running in parallel and communicating whenever necessary. For example, a production line may consist of a number of independent sensors and actuators, for a single-purpose operation, that have to synchronise in order for the whole production task to be completed. This behaviour is modelled with a number of timed automata running in parallel, which are able to synchronise. Such a model is called a network of timed automata.

A transition can be taken separately, or synchronised with another automaton through channels, both results in a new state. Timed Automata are composed into a network, over a common set of clocks and actions, consisting of n timed automata $\mathcal{A}_i$. The following definition gives the behaviour of a network of timed automata.

Definition 2.3.1 (Networks of timed automata)
Let $\mathcal{A}_i = (Act, C, L_i, l_i^0, \delta_i, I_i)$ be a network of n timed automata. $L_i, l_i^0, \rightarrow_i$ and I_i is, for each automata in the network, the same as in ordinary automata. The set of clocks C contains all the clocks in the network and the actions are defined as follows:

$$Act = \{c! | c \in Chan\} \cup \{c? | c \in Chan\} \cup N,$$

where N is a finite set of ordinary actions, and $Chan$ is a finite set of channel names.

The timed transition system is defined as $\langle S, s_0, \rightarrow \rangle$, where $S = L_1 \times \cdots \times L_n \times (C \rightarrow \mathbb{R}_{\geq 0})$ is the set of states, $s_0 = (\bar{l}_0, v_0)$ is the initial state and $\rightarrow \subseteq S \times S$ is the transition relation defined by:

- $(\bar{l}, v) \rightarrow (\bar{l}, v + d)$ *for all* $d \in \mathbb{R}_{\geq 0}$ *such that* $v + d' \models I(\bar{l})$ *for each real number* d' *in the interval* $[0, d]$.
- $(\bar{l}, v) \rightarrow (\bar{l}[l_i'/l_i], v')$ *if there exist* $(l_i \xrightarrow{g,\tau,r} l_i')$ *such that* $v \models g$, $v' = v[r \leftarrow 0]$, $v' \models I(\bar{l}[l_i'/l_i])$ *and* $\tau \in N$.
- $(\bar{l}, v) \rightarrow (\bar{l}[l_j'/l_j, l_i'/l_i], v')$ *if there exist* $(l_i \xrightarrow{g_i,c?,r_i} l_i') \in \delta_i$ *and* $(l_j \xrightarrow{g_j,c!,r_i} l_j') \in \delta_j$ *such that* $v \models g_i \wedge g_j$, $v' = v[r_i \leftarrow 0 \cup r_j \leftarrow 0]$ *and* $v' \models I(\bar{l}[l_j'/l_j, l_i'/l_i])$.

Where $\bar{l} = (l_i, \ldots, l_n)$ is a location vector. $\bar{l}[l'_i/l_i]$ is used to denote the vector where the ith element l_i of $\bar{l}$ is replaced by l'_i and

$$I(\bar{l}) = \bigwedge_{i \in \{1,\ldots,n\}} I_i(l_i)$$

is the invariant function over location vectors.

In Definition 2.3.1 on the preceding page the channel names end with either ! or ?. If an automaton has a channel that ends with !, e.g. $a!$, it means that it can synchronise on the channel a with some other automaton, offering the action $a?$ in exchange. Intuitively, action $a!$ can be thought of standing for an "output on channel a", whereas $a?$ stands for an "input on channel a". The automaton can then communicate using hand-shake synchronisation. Moreover, all channels are implicitly assumed to be restricted at the highest level; hence the synchronisation in networks of timed automata is always forced.

Example 2.3.1
To see how networks and channels can be used, let us revisit the item-sorter model. Suppose that we want to model a piston as a separate entity that, based on the sensor output can decide to push the item of the conveyor belt or do nothing and let the item pass. There are two automata in the network, and three different channels. The two automata are obvious the item model and the piston model and the channels are `is.red`, `is.blue` and `push.off`. The network can be seen in Figure 2.4 on the following page

The piston can only push blue items off the belt, so whenever it synchronises through the red channel it just loops back into the active (`ACT`) location. However if the sensor sends `is.blue!`, the piston will immediately go into the push (`PUS`) location with the action `is.blue?` while resetting its own internal clock y. In the `PUS` location it will use the clock y to send the `push.off!` action to the item when y is in the interval $[10, 20]$.

2.4 Region graphs and Zones

The problem with timed automata is that all, but the simplest ones, will generate a timed transition systems with infinitely many reachable states. This is because the states of a timed automata contain not only locations but also particular clock valuations. Even with only a single clock there will be uncountably many clock valuations, unless all transitions are guarded by $x \leq 0$ and all locations have the invariant $x \leq 0$, as the clock can take any value in a non-empty interval, belonging to $\mathbb{R}_{\geq 0}$.

Definition 2.4.1 (Real decomposition)
Let $d \in \mathbb{R}_{\geq 0}$ be a real number. $\lfloor d \rfloor$ is the integer part of d and $frac(d)$ is the fractional part of d. Any $d \in \mathbb{R}_{\geq 0}$ can then be written as $d = \lfloor d \rfloor + frac(d)$.

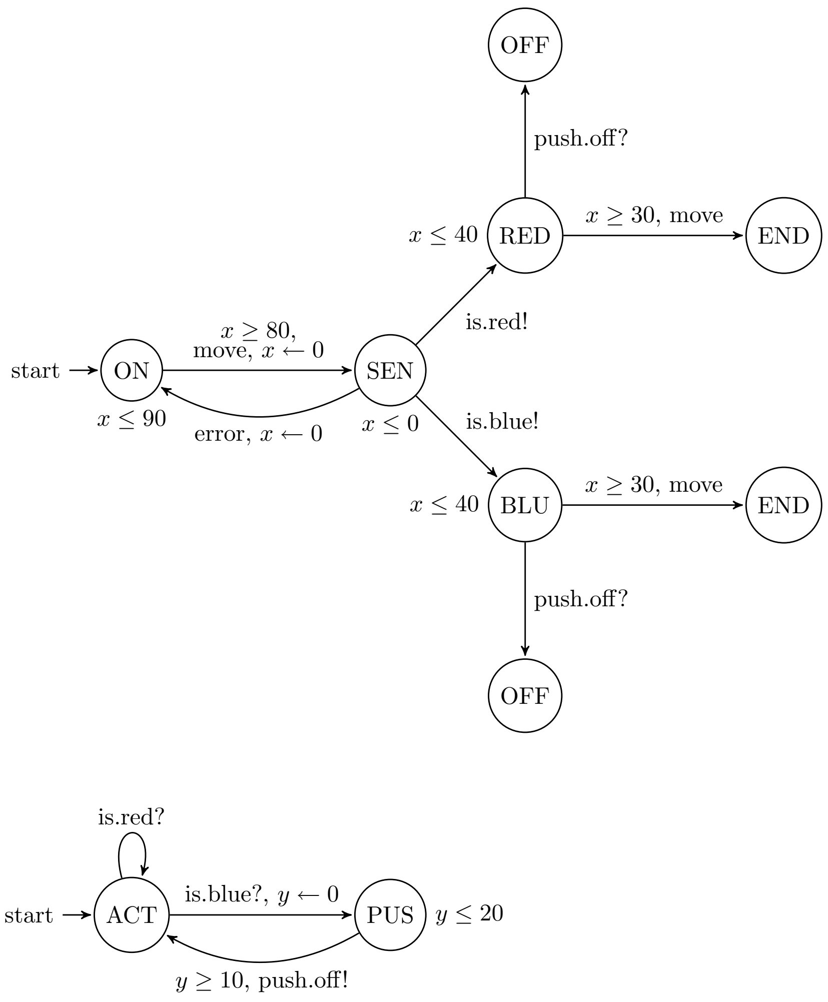


Figure 2.4: Two networked timed automata.

A way to represent infinitely many clock valuations, finitely, is to use region graphs. Region graphs partition the collection of valuations, for a given timed automaton, into finitely many equivalence classes. The partitioning must be done in a way, such that valuations from the same class does not create any difference in the behaviour of the system, with respect to reachability of states. That is, if a set of states, which share their locations, and their associated clock valuations are located in the same region, then these configurations can reach the same regions.

Definition 2.4.2 (Clock valuations equivalence)
Let $\mathcal{A}$ be a timed automaton and let c_x denote the largest constant, which the clock $x \in C$ is ever compared with either in the guards or invariants of $\mathcal{A}$. We say that the two clock valuations v and v' are called equivalent, and write $v \equiv v'$, iff.

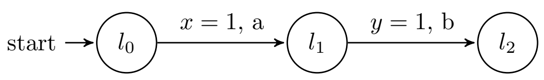


Figure 2.5: A timed automaton with two clocks.

1. *For each $x \in C$, it holds that either both $v(x)$ and $v'(x)$ are greater than c_x or*
$$\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$$

2. *For each $x \in C$ such that $v(x) \leq c_x$ we have*
$$frac(v(x)) = 0 \Leftrightarrow frac(v'(x)) = 0$$

3. *For all $x, y \in C$ such that $v(x) \leq c_x$ and $v(y) \leq c_y$ we have.*
$$frac(v(x)) \leq frac(v(y)) \Leftrightarrow frac(v'(x)) \leq frac(v'(y)).$$

In part 1 of Definition 2.4.2 on the preceding page, the valuations $v(x)$ and $v'(x)$ are compared to the greatest clock constant x is ever compared to. If they both are greater, it means that they can only leave their region if they both are reset. This implies that they have the exact same behaviour and can therefore be put in the same equivalence class. Continuing in part 1, since clocks are only compared to values in $\mathbb{N}$ on guards and invariants, the fractional part of a clock valuation can be ignored. So if two valuations share the same integer part, they can be put in the same equivalence class. As an example suppose, in a location l, there is a transition with the guard $x \leq 1$ and another with $x \leq 2$. From the two states $(l, [x = 1.2])$ and $(l, [x = 1.4])$ it is only possible to take the second transition, so the valuations are in the same equivalence class. From the states $(l, [x = 0.4])$ and $(l, [x = 0.9])$ it is possible to take both transitions, so these two evaluations are together in another equivalence class.

Part 2 of Definition 2.4.2 on the facing page takes care of the special case when the fractional part equals zero. The valuations $[x = 1]$ and $[x = 1.4]$ in the previous example are not equivalent, because the first one could take both transitions, while the second cannot. Therefor two valuations, where one has a zero fractional part, can only be equivalent if the other one also has a zero fractional part.

Part 3 of Definition 2.4.2 on the preceding page deals with the ordering of the fractional parts between two different clocks. For the timed automaton seen in Figure 2.5 imagine the valuations $v_1 = [x = 0.8, y = 0.3]$ and $v_2 = [x = 0.5, y = 0.9]$.

Both requirements from part 1 and 2 of Definition 2.4.2 on the preceding page are met, but the states (l_0, v_1) and (l_0, v_2) can lead to different behaviours. State (l_0, v_1) can delay 0.2 time units, take transition a, delay 0.5 units again, and take b. The state (l_0, v_2) cannot match this, because it would first have to delay 0.5 to take a resulting in y growing to 1.4 and disabling b.

It can be showed that the equivalence $\equiv$ on clock valuations have a number of closure properties, which leads to the following theorem:

Theorem 2.4.1 (Finite regions and untimed bisimilarity)
[LAS07] (Thm 11.3, page 209)
Let $\mathcal{A}$ be a timed automaton. The equivalence relation $\equiv$ partitions the clock valuations of $\mathcal{A}$ into finitely many equivalence classes. Moreover, whenever v and v' are in the same equivalence class ($v \equiv v'$ holds) then, for any location l of $\mathcal{A}$, the states (l, v) and (l, v') are untimed bisimilar.

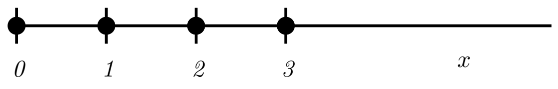
What this theorems says, is that: whenever $(l_1, v_1) \equiv (l_2, v_2)$ and $(l_1, v_1) \xrightarrow{a} (l'_1, v'_1)$ with $a \in Act$, then there exists $(l_2, v_2) \xrightarrow{a} (l'_2, v'_2)$ such that $(l'_1, v'_1) \equiv (l'_2, v'_2)$; if $(l_1, v_1) \equiv (l_2, v_2)$ and $(l_1, v_1) \xrightarrow{d_1} (l'_1, v'_1)$ for some $d_1 > 0$, then there exist $d_2 > 0$ such that $(l_2, v_2) \xrightarrow{d_2} (l'_2, v'_2)$ and $(l'_1, v'_1) \equiv (l'_2, v'_2)$.

Each clock valuation v can be represented by an equivalence class, denoted by $[v]_\equiv$, also called a region. Each region consists of a finite collection of clock constraints that it satisfies. For instance, consider the valuation v over two clocks x, y such that $v(x) = \sqrt{2}$ and $v(y) = 1.3$. If both c_x and c_y are equal to 2, then each valuation v' that is equivalent to v satisfies the constraint $1 < y < x < 2$ and $[1 < y < x < 2]_\equiv$ is used to denote the region $[v]_\equiv$.

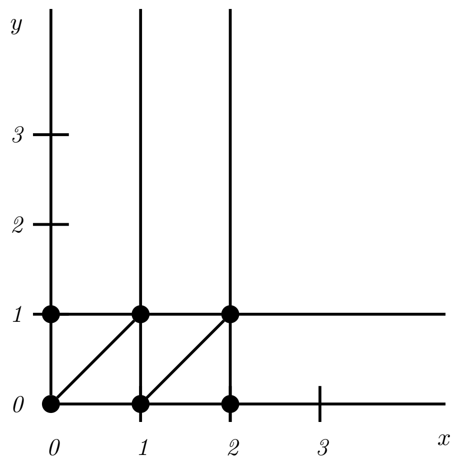
Example 2.4.1
Consider a timed automaton with only one clock x such that $c_x = 3$. In this automaton there are exactly 8 regions, consisting of 4 corner points, 3 closed line segments and 1 open line segment

- $[x = 0]_\equiv, [x = 1]_\equiv, [x = 2]_\equiv, [x = 3]_\equiv$,
- $[0 < x < 1]_\equiv, [1 < x < 2]_\equiv, [2 < x < 3]_\equiv$ *and*
- $[3 < x]_\equiv$.

The regions can be illustrated as follows:



Example 2.4.2
Consider the timed automaton with two clocks x and y such that $c_x = 2$ and $c_y = 1$. These two clocks give exactly 28 regions depicted graphically as follows:



In general each region of a timed automaton $\mathcal{A}$ can be uniquely represented by specifying the following items of information:

- For each clock x, one constraint from the set

  $\{x = n | n \in \{0, 1, \ldots, c_x\}\} \cup \{n < x < n+1 | n \in \{0, 1, \ldots, c_x - 1\}\} \cup \{c_x < x\}$;

- For each pair of clocks x and y, where $n < c_x$ and $m < c_y$. If these clocks satisfy constraints of the form $n < x < n + 1$ and $m < y < m + 1$, there must be an indication, for each valuation v in that region, if $frac(v(x)) < frac(v(y))$ or $frac(v(y)) < frac(v(x))$.

A state of the form (l, v) can be replaced by a symbolic state $(l, [v]_\equiv)$ in the region graph, where $[v]_\equiv$ is the region to where v belongs. When there is a delay or discrete transition between two states, there is also a transition between the corresponding symbolic states.

Definition 2.4.3 (Region graph)
The region graph of a timed automaton $\mathcal{A}$ over a set of clocks C and actions Act is a labelled transition system

$$T_r(\mathcal{A}) = (S, Act \cup \{\varepsilon\}, \{\overset{a}{\Rightarrow} | a \in Act \cup \{\varepsilon\}),$$

where:

- $S = \{(l, [v]_\equiv) | l \in L,\ v : C \to \mathbb{R}_{\geq 0}\}$ *are symbolic states.*
- $\Rightarrow$ *on symbolic states is defined as following:*
  - *For each action $a \in Act$,* $(l, [v]_\equiv) \overset{a}{\Rightarrow} (l', [v']_\equiv) \Leftrightarrow (l, v) \xrightarrow{a} (l', v')$.
  - $(l, [v]_\equiv) \overset{\varepsilon}{\Rightarrow} (l, [v']_\equiv) \Leftrightarrow (l, v) \xrightarrow{d} (l, v')$*, for some $d \in \mathbb{R}_{\geq 0}$.*

Region graphs provide a finite representation of the infinite timed transition systems generated by timed automata, however the state space might still be very large. The state-space explosion is exponential in the number of clocks and in the maximal constants appearing in the guards [AD94]. A more efficient representation of the configuration space is to use zones. Zones are convex unions of regions and give a coarser and more compact representation of the state space.

Definition 2.4.4 (Zone)
A zone Z is the set of clock valuations described by an clock constraint $g_z \in \mathcal{B}(C)$:

$$Z = \{v | v \models g_z\}.$$

A symbolic state is now a pair (l, Z), where l is a state and Z is a zone. Like clock valuations there are three operations used to manipulate zones.

Definition 2.4.5 (Delay, reset and intersection of zones)
Let Z and Z' be zones and r a set of clocks. Then

- $Z^{\uparrow} = \{v + d | v \in Z \wedge d \in \mathbb{R}_{\geq o}\}$.
- $Z[r] = \{v[r] | v \in Z\}$.
- $Z \wedge Z' = \{v | v \in Z \wedge v \in Z'\}$

Zones are closed under the three operations [LPY95], which is to say that whenever Z_1 and Z_2 is described by a clock constraint g_1 and g_2 then there are clock constraints g_1', g_1'' and g_{12} describing $Z_1^{\uparrow}$, $Z_1[r]$ and $Z_1 \wedge Z_2$. The intersection operator can be used to constrain zones. For example, if the transition system is in zone Z and there is a guard g describing Z' on an outgoing transition a. We can intersect Z with Z' to create the zone Z'' where the transition a is enabled.

Symbolic transitions between symbolic states describe sets of corresponding concrete transitions.

Definition 2.4.6 (Symbolic transition relation)
The symbolic transition relation $\rightsquigarrow$ over symbolic states is defined as follows:

- $(l, Z) \rightsquigarrow (l, Z^{\uparrow} \wedge I(l))$.
- $(l, Z) \rightsquigarrow (l', (Z \wedge g)[r] \wedge I(l'))$ *if* $l \xrightarrow{g.a.r} l'$.

In Definition 2.4.6, the first clause corresponds to simultaneously performing delay transitions from all the concrete states corresponding to (l, Z). The resulting target zone consists of all the valuations in the future of Z that satisfy the invariant of location l. The second clause corresponds to simultaneously performing the discrete action corresponding to the transition $l \xrightarrow{g,a,r} l'$. The resulting target zone consists of all the valuations that satisfy the invariant of location l', and it may be obtained by resetting the clocks in r for valuations in the zone Z that meet the guard g.

Example 2.4.3
For example, consider the simple timed automaton in Figure 2.6. The following sequence of symbolic transitions, with zones illustrated in Figure 2.7 on the following page shows how to reach the location l_1:

$$\begin{aligned}(l_0, x = y = 0) &\rightsquigarrow (l_0, x - y = 0)\\ &\rightsquigarrow (l_0, y = 0 \wedge x \leq 2)\\ &\rightsquigarrow (l_0, 0 \leq x - y \wedge x - y \leq 2)\\ &\rightsquigarrow (l_0, y = 0 \wedge x \leq 4)\\ &\rightsquigarrow (l_0, 0 \leq x - y \wedge x - y \leq 4)\\ &\rightsquigarrow (l_1, y \leq 2 \wedge 4 \leq x \wedge x - y \leq 4).\end{aligned}$$

The shaded areas in Figure 2.7 on the next page represent the futures of the zones described by the solid lines. The darker grey area in (a) and (b) describes when the action a is enabled and in (c) when the action c is enabled and the location l_1 can be reached.

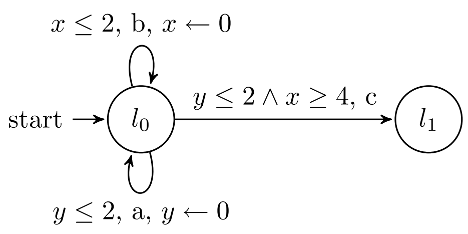

$x \leq 2$, b, $x \leftarrow 0$
start
l_0
$y \leq 2 \wedge x \geq 4$, c
l_1
$y \leq 2$, a, $y \leftarrow 0$

Figure 2.6: A timed automaton.

This chapter has presented time automata and extensions that allow multiple timed automata in a network, together with structures that represent the infinite state space of timed automata in an finite way. What comes next, is how to control a timed automaton such that certain properties are upheld. Because timed automata can model discrete event systems this is equivalent to synthesise a control program for such a system.

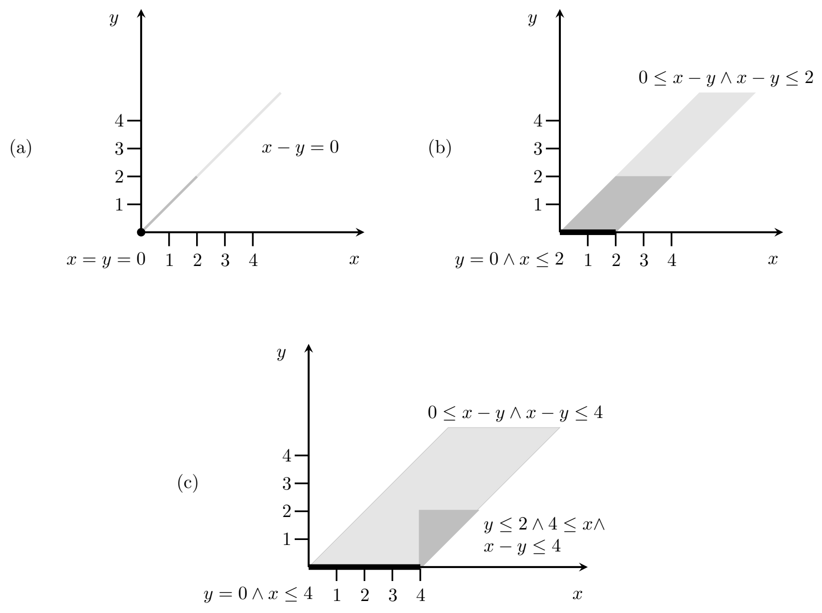


Figure 2.7: Symbolic exploration of the timed automaton in Figure 2.6 on the previous page.

Chapter 3

Games and strategy synthesis

An approach to program synthesis, is to view the interaction between the controller one wants to design and the environment, in which it is supposed to operate, as a two player games. The two players are the controller and the environment respectively, and the objective of the game is simply for one of the players to win, by forcing a state which is bad for the opponent. There is are two different game types, reachability and safety. A reachability game, is when the controller tries to reach a specific set of winning states, and the environment tries to prevent reaching them. In a safety game, the controller tries to stay within a set of safe states, and the environment tries to force the system of these, into the bad states. If the environment is able to do this the controller loses, and thus the system cannot be controlled. Conversely if the controller wins, its strategy will enable it to control the system.

A strategy for a given game is a function of actions and delays that tell the controller what to do in any given state of the game. A strategy is a winning strategy if the controller always wins no matter what the environment does. This chapter is based on [MPS95] and [BC06].

3.1 Real-time Games

There are several formulations of the control problem [BC06]. As we have already written, it can be defined as a game between a controller and a environment. Of special interest is asymmetric games where the environment has precedence over the controller. This means that if the environment decides to do an action or delay it will happen even though the controller wants to do an action at the same point in time. This behaviour reflects reality closely. For example: in a production line, if a piston breaks down just as the controller issues a command to use it, nothing will happen; or if a button on a keyboard is pushed just as a computer loses power, the push will not be registered.

To model this kind of behaviour we take a timed automaton, and give it two sets of actions, the actions the controller is able execute and the actions that occur in the environment.

Definition 3.1.1 (Timed game automaton)
A timed game automaton is a timed automaton $\mathcal{A} = (Act, B(C), L, l_0, \delta, I)$ where the actions in Act are partitioned into two subsets:

- *Act_c is the set of controllable actions.*
- *Act_u is the set of uncontrollable actions.*

Intuitively, the controller will be able to perform the controllable actions, whereas the environment will be able to perform the uncontrollable actions.

Example 3.1.1
Let us go back to the item sorter example. Suppose we want a controller that makes sure that all blue items are pushed off the conveyor belt. The controller does not have any control over when the item is on the belt, when the item arrives at the sensor, what the sensor registers, or when the item arrives at the end of belt. Therefore all these actions are uncontrollable. The controller can only decide to push or not to push the item off, if the item is within the time interval of the piston. This makes the action `push.off` the only controllable action in the model. Let all the uncontrollable edges be represented with dotted lines. The revised model can be seen in Figure 3.1 An obvious strategy for the controller is to wait until the item is in location `BLU` and then push off the item when the clock x is within the interval $[10, 20]$, or if the item ends in location `RED` do nothing and let it reach the end of the belt.

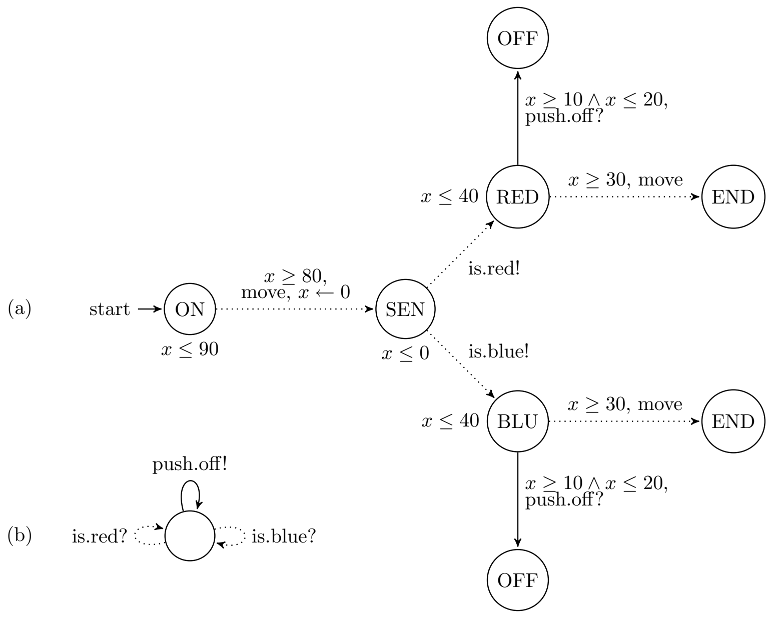


Figure 3.1: **a:** The item-sorter model with controllable and uncontrollable transitions. **b:** The controller for a.

In order to use real-time games to synthesise a strategy for a timed game automaton, we introduce runs and steps in an automaton.

Definition 3.1.2 (Runs and steps)
A run of an automaton $\mathcal{A}$ starting in (l_0, v_0) is a finite or infinite sequence

$$\beta = (l_0, v_0) \xrightarrow{d_0} (l_1, v_1) \xrightarrow{a_0} (l_2, v_2) \xrightarrow{d_1} (l_3, v_3) \xrightarrow{a_1} (l_4, v_4) \xrightarrow{d_2} (l_5, v_5) \xrightarrow{a_2} \dots,$$

of alternating time and discrete transitions, where $l_i = l_{i+1}$ between time transitions. A pair of states $(l_i, v_i), (l_{i+1}, v_{i+1})$ with a time or discrete transition between them is called a step.

Definition 3.1.2 covers all possible runs in an automaton. For two discrete transitions in a row set the time transition between to zero, and for two time transition set the action between them to empty.

A run is non-Zeno, if it has infinitely many delay-transitions and the sum of the corresponding delays diverges. Recall that the symbol λ represents elapsing time. Let the set of all non-Zeno runs that $\mathcal{A}$ can generate be denoted by $\mathcal{L}_Z(\mathcal{A})$.

Strategies that leads to Zeno runs are problematic, as these are unimplementable in reality. Since it is infeasible to do an infinite amount of actions in a finite amount of time. Zeno runs are not a problem with strategies synthesised for reachability games, as the runs in these will always be finite, as the goal states must be reached in a finite number of steps. Safety games however can generate Zeno-runs, as there is nothing that prevents infinite runs in these strategies. This can be avoided by making sure the model forces time to elapse, thus avoiding the whole problem.

Definition 3.1.3 (Real-time Strategy)
A simple real-time strategy is a function $C : L \times (C \to \mathbb{R}_{\geq 0}) \to Act_c \cup \{\lambda\}$. The action must be enabled in the particular state for the strategy to be correct.

According to this function, the strategy commands at any state, (l, v), whether to issue some enabled transition $c \in Act_c$ or to do a delay transition d. The strategy bases its selection of the next command by the last visited automaton state. Therefore the strategy need only observe the current state of the timed game automaton, and only requires a finite amount of memory.

Definition 3.1.4 (Controlled Runs)
Given a simple strategy C, a pair $((l, v), (l', v'))$ of states is a C-step if it is either

- *An u step, where $u \in Act_u$.*
- *A c step such that $C(l, v) = c \in Act_c$.*
- *A d step for some $d \in \mathbb{R}_{\geq 0}$ such that $\forall d', d' \in [0, d)$, $C(l, v + d') = \lambda$.*

A C-run is a run consisting of C-steps, and the set of C-runs of $\mathcal{A}$ is denoted by $\mathcal{L}_C(\mathcal{A})$.

Clearly, every C-run is a run and $\mathcal{L}_C(\mathcal{A}) \subseteq \mathcal{L}_Z(\mathcal{A})$.

Now, as mentioned before we want the strategy ensure a property, which is to say that we want the set of runs which guarantee the property. For every infinite run $\alpha \in \mathcal{L}_Z(\mathcal{A})$, let $Vis(\alpha)$ denote the set of all states appearing in α and let $L \times V$ denote the complete state space $(L \times (C \to \mathbb{R}_{\geq 0}))$ of a timed game automaton.

Definition 3.1.5 (Winning Objective)
Let $\mathcal{A}$ be a timed game automaton. A winning objective for $\mathcal{A}$ is

$$\Omega \in \{(F, \forall\Diamond), (F, \forall\Box), ((F_1, F_2), \forall \boldsymbol{U}), ((F_1, F_2), \forall \boldsymbol{W})\}.$$

where F, F_1, F_2 are subsets of $L \times V$ and referred to as the winning states.The set of runs that are winning according to Ω are defined as follows:

$\mathcal{L}(\mathcal{A}, (F_1, F_2), \forall \boldsymbol{U})$	$\{\alpha \in \mathcal{L}(\mathcal{A}) \mid (Vis(\alpha) \setminus F_2) \subseteq F_1 \wedge Vis(\alpha) \cap F_2 \neq \emptyset\}$	*α always remains in F_1 until it visits F_2*
$\mathcal{L}(\mathcal{A}, (F_1, F_2), \forall \boldsymbol{W})$	$\{\alpha \in \mathcal{L}(\mathcal{A}) \mid (Vis(\alpha) \setminus F_2) \subseteq F_1 \wedge (Vis(\alpha) \cap F_2 \neq \emptyset \vee Vis(\alpha) \cap F_2 = \emptyset)\}$	*α always remains in F_1 until it may visit F_2*
$\mathcal{L}(\mathcal{A}, F, \forall\Box)$	$\{\alpha \in \mathcal{L}(\mathcal{A}) \mid Vis(\alpha) \subseteq F\}$	*α always remains in F*
$\mathcal{L}(\mathcal{A}, F, \forall\Diamond)$	$\{\alpha \in \mathcal{L}(\mathcal{A}) \mid Vis(\alpha) \cap F \neq \emptyset\}$	*α always eventually visits F*

In CTL **U** is denoted as the until operator and the syntax for writing a property is $\forall(F_1 \ \mathbf{U} \ F_2)$, likewise with the weak-until operator **W** which is written $\forall(F_1 \ \mathbf{W} \ F_2)$. $\Diamond$ is the eventually operator and specifies properties with syntax $\forall\Diamond \ F$. Such properties are called reachability-properties. $\Box$ is the always operator written $\forall\Box \ F$ and specifies safety-properties. We will not look further in the until and weak-until operators, but instead focus on the safety and reachability properties. Note that the eventually operator is a special case of the until operator, namely $\forall\Diamond F = \forall((L \times V) \ \mathbf{U} \ F)$. Similarly, the always operator is a special case of the weak-until operator, namely $\forall\Box F = \forall(F \ \mathbf{W} \ \emptyset)$. Let $\mathcal{L}(\mathcal{A}, \Omega)$ denote all the runs which uphold an winning objective. Interested readers may refer to [BK08], for further information on CTL.

Definition 3.1.6 (Real-Time Strategy Synthesis [MPS95])
Given a timed game automaton $\mathcal{A}$ and an winning objective Ω, the problem ***RT-Synth***$(\mathcal{A}, \Omega)$ *is: Construct a real-time strategy C such that $\mathcal{L}_C(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}, \Omega)$.*

3.2 Synthesising the Strategy

In order to tackle the real-time strategy synthesis problem we introduce the following definition.

Definition 3.2.1 (Controllable/uncontrollable discrete predecessors)
Let $\mathcal{A}$ be a timed game automaton. The controllable and uncontrollable discrete predecessors of a set of states $A \subseteq L \times V$ as follows:

- $$\pi_c^D(A) = \left\{ (l,v) \in L \times V \;\middle|\; \begin{array}{l} \exists c \in Act_c,\ c \textit{ is enabled in } (l,v), \\ \textit{and } \forall (l',v') \in L \times V, \\ (l,v) \xrightarrow{c} (l',v') \Rightarrow (l',v') \in A \end{array} \right\}$$
- $$\pi_u^D(A) = \left\{ (l,v) \in L \times V \;\middle|\; \begin{array}{l} \exists u \in Act_u,\ u \textit{ is enabled in } (l,v), \\ \textit{and } \exists (l',v') \in L \times V s.t. \\ (l,v) \xrightarrow{u} (l',v') \textit{ and } (l',v') \in A \end{array} \right\}$$

The set π_c^D is the set of states from which we can enforce a state of A by doing a controllable action. The set π_u^D is the set of states from which the environment can do an uncontrollable action which leads to a configuration in A. Because we are dealing with timed systems we need to define a timed predecessor function $\pi^T(A)$.

A state (l,v) must be in $\pi^T(A)$ if and only if it is possible to let d time units elapse for some $d \in \mathbb{R}_{\geq 0}$ and use a controllable action to reach A and no uncontrollable action played before or at d leads outside A; or A can be reached by just letting time elapse and no uncontrollable action leads outside A. This is a crucial feature of the game, as there are no "turns" and the adversary need not wait for the player's next move.

Definition 3.2.2 (Timed controllable predecessor)
The timed controllable predecessor function is defined as:

$$\pi^T(A) = \left\{ (l,v) \in L \times V \;\middle|\; \begin{array}{l} \exists d\ (l,v) \xrightarrow{d} (l',v'),\ (l',v') \in \pi_c^D(A) \\ \quad \wedge P_{[0,d]}(l,v) \cap \pi_u^D(\overline{A}) = \emptyset; \\ \\ or\ \exists d\ P_{[d,+\infty)}(l,v) \subseteq A \\ \quad \wedge P_{[0,+\infty)}(l,v) \cap \pi_u^D(\overline{A}) = \emptyset \end{array} \right\},$$

where $P_{Int}(l,v) = \{(l, v+d) \in I(l) | d \in Int\}$ with the interval $Int \subseteq \mathbb{R}_{\geq 0}$.

The timed controllable predecessor function is used to calculate the set of winning states W , namely the set of states from which a strategy can enforce good behaviours according to the property Ω. They can be characterised by the following fixed-point expressions:

$$\begin{array}{ll} \forall\Box : & \top W \left(F \cap \pi^T(W)\right) \\ \forall\Diamond : & \bot W (F \cup \pi^T(W)) \end{array}$$

Above $\top$ denotes the greatest fixed point operator and $\perp$ the least fixed point operator. The model is controllable iff $(l_0, v_0) \in W$. The sets W for each of the cases can be calculated by the following iterative algorithms:

```
∀◇-Strategy()
1  $W_0 := \emptyset$
2  for $i = 0, 1, \ldots,$ do
3      $W_{i+1} := F \cup \pi^T(W_i)$
4      if $W_{i+1} = W_i$ then
5          Return
```

```
∀□-Strategy()
1  $W_0 := L \times V$
2  for $i = 0, 1, \ldots,$ do
3      $W_{i+1} := F \cap \pi^T(W_i)$
4      if $W_{i+1} = W_i$ then
5          Return
```

In the $\forall\Diamond$-case, each W_i contains the states from which a visit to F can be enforced after at most i steps and in the $\forall\Box$-case, it consists of the states from which the timed game automaton can be kept in F for at least i steps.

The algorithms shown works, as there is a finite set of regions, and also because regions are closed under the timed predecessor function (Chapter 2 and [MPS95, BC06]), which is to say that whenever W_i is a union of regions then $\pi^T(W_i)$ is a union of regions. This implies that every W_i is an element of a finite set, and thus, by monotonicity, a fixed point is eventually reached within a finite number of iterations.

A strategy is constructed in the following manner: Whenever a state (l, v) is added to W it is due to either waiting $d \in [d_1 = 0, d_2]$ and issuing a $c \in Act_c$ or waiting $d' \in [d_1, d_2 = +\infty)$ will lead to a winning position. Therefore by letting $C(l, v) = \lambda$ when $d_1 > 0$ and $C(l, v) = c$ when $d_1 = 0$ a strategy is constructed. Care should be taken in avoiding to keep adding winning states that loops around and ends in themselves, even though a fix point will eventually be reached it will result in a lot of unnecessary computations.

Example 3.2.1

Here is an example to show how the time controllable predecessor function and one of the fixed point algorithms work. Assume the following timed game automaton in Figure 3.2 on the facing page with $Act_c = \{c\}$, $Act_u = \{u\}$, $x, y \in C$ and $F = \{(l_3, (x, y)) | x \geq 0 \text{ and } y \geq 0\}$. Let the winning objective be $(F, \forall\Diamond)$:

$$\begin{cases} W_1 & = F \\ W_2 & = W_1 \cup \{(l_1, (x, y)) | y \leq 1 \text{ and } x - y > 1\} \\ W_3 & = W_2 \cup \{(l_0, (x, y)) | y \leq 2 \text{ and } y - x < 1\} \\ W_4 & = W_3 \end{cases}$$

The set of states from which the strategy can enforce F is thus W_4, which includes the initial state $(l_0, (0, 0))$.

Through this section it has been argued that:

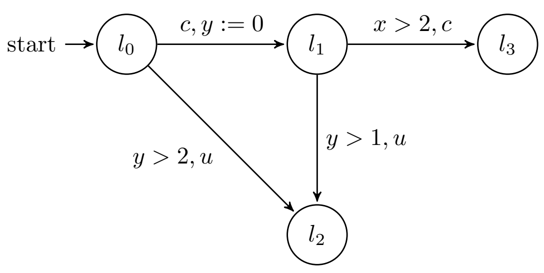


Figure 3.2: A simple timed game automaton.

Theorem 3.2.1 (Strategy Synthesis for Timed Systems [MPS95])
Given a timed game automaton $\mathcal{A}$ and an acceptance condition

$$\Omega \in \{(F, \forall\Diamond), (F, \forall\Box), ((F_1, F_2), \forall \boldsymbol{U}), ((F_1, F_2), \forall \boldsymbol{W})\}.$$

*the problem **RT-Synth**$(\mathcal{A}, \Omega)$ is solvable.*

While we have showed that synthesising a strategy is possible, the methods here do not take in to consideration how to find the good states in the first place.

3.3 UPPAAL TIGA

UPPAAL TIGAis an extension to the model checker UPPAAL, which uses a network of timed game automata to synthesise reachability and safety strategies. A strategy in TIGA is a subset of the controllable transitions which guarantees the specified property holds, no matter which uncontrollable transitions are taken at any time. This section is based on [CDF$^+$05]

Definition 3.3.1 (Timed game automaton)
A timed game automaton (TGA) in UPPAAL TIGA, is a timed game automaton. Where the set of actions, Act, are partitioned into two sets, Act_c and Act_u. Act_c is the set of controllable actions, and Act_u is the set of uncontrollable action.

The following must hold:

- $Act_c \subseteq Act$,
- $Act_u \subseteq Act$,
- $Act_c \cap Act_u = \emptyset$ *and*
- $Act_c \cup Act_u = Act$.

UPPAAL TIGA uses a reachability algorithm that allows it to synthesise strategies. The algorithm uses backwards-propagation to only include states which are safe. A state is safe if it can reach the goal states, specified in the property, using delay and controllable transitions, no matter which delay and uncontrollable transitions the environment takes along the way.

In order to introduce the algorithm we will first introduce some helper functions. The first ones are the $Pred_a$ and the $Post_a$, these functions gives all the predecessors or successors respectively of a symbolic state given a discrete transition with an action $a \in Act$.

Definition 3.3.2 (a-predecessors)
$Pred_a(X) = \{(l,v) | \exists (l',v') \in X, (l,v) \xrightarrow{a} (l',v')\}$ where $X = (l,Z)$ denotes a symbolic state, l and l' are locations and v and v' are clock valuations.

Definition 3.3.3 (a-successors)
$Post_a(X) = \{(l',v') | \exists (l,v) \in X, (l,v) \xrightarrow{a} (l',v')\}$ where $X = (l,Z)$ denotes a symbolic state, l and l' are locations and v and v' are clock valuations.

To find the winning set of states we also need to be able to find all the controllable- and uncontrollable-predecessors of a symbolic states. This is done by the two functions $Pred_c$ and $Pred_u$.

Definition 3.3.4 (Controllable-predecessors)
$Pred_c(X) = \{(l,v) | \exists (l',v') \in X, (l,v) \xrightarrow{c} (l',v'), c \in Act_c\}$ where $X = (l,Z)$ denotes a symbolic state, l and l' are locations and v and v' are clock valuations.

Definition 3.3.5 (Uncontrollable-predecessors)
$Pred_u(X) = \{(l,v) | \exists (l',v') \in X, (l,v) \xrightarrow{u} (l',v'), \alpha \in Act_u\}$ where $X = (l,Z)$ denotes a symbolic state, l and l' are locations and v and v' are clock valuations.

This takes care of the discrete transitions, we also need to take time into account. To do this we introduce two additional functions, $X^{\uparrow}$ and $X^{\downarrow}$. These two functions return the time-successors and -predecessors respectively.

Definition 3.3.6 (Time-successors)
$X^{\uparrow} = \{(l, v+d) | v \models Inv(l), v+d \models Inv(l)\}$ where $X = (l,Z)$ is a symbolic states, l is a location, v is a clock valuation and $d \in \mathbb{R}_{\geq 0}$.

Definition 3.3.7 (Time-predecessors)
$X^{\downarrow} = \{(l, v-d) | (l,v) \in X\}$ where $X = (l,Z)$ is a symbolic state, l is a location, v is a clock valuation and $d \in [0,v] \subset \mathbb{R}$.

The final functions which we need is the safe time-predecessors, $Pred_t(X, Y)$. This function gives all the time predecessors of X, which cannot delay into Y, see Figure 3.3 for a graphical representation of the function.

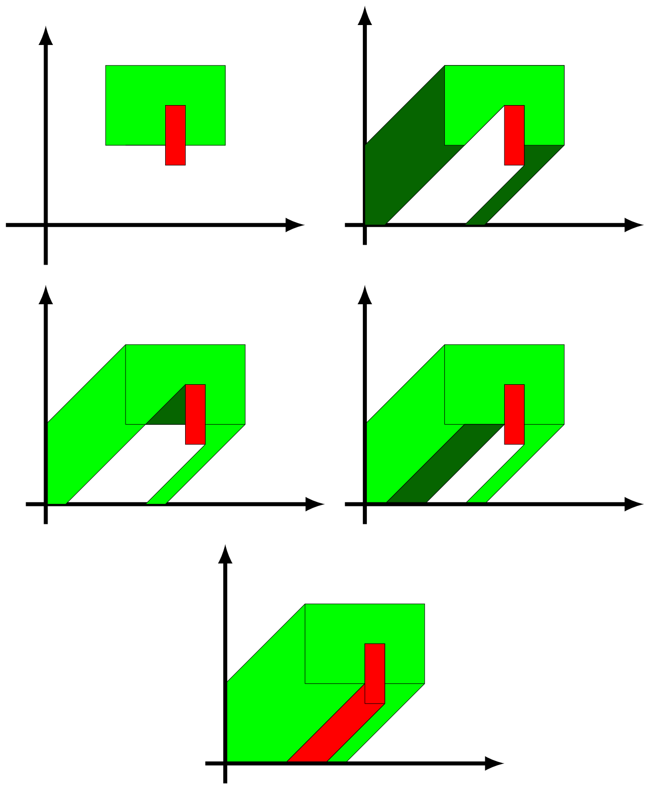


Figure 3.3: **Top left:** The green square represents X, and the red square represents Y. **Top right:** The dark green area represents the addition to X after $X^{\downarrow} \setminus Y^{\downarrow}$. Notice that part of X has disappeared. **Middle left:** The dark green area represents the addition of $(X \cap B^{\downarrow}) \setminus Y$. **Middle right:** The dark green represents the addition of $((X \cap B^{\downarrow}) \setminus Y)^{\downarrow}$. **Bottom:** This is the final result of $Pred_t(X, Y)$, where the green represents the safe time-predecessors and the red represents unsafe predecessors.

Definition 3.3.8 (Safe time-predecessors)
$Pred_t(X,Y) = (X^{\downarrow} \setminus Y^{\downarrow}) \cup \big((X \cap Y^{\downarrow}) \setminus Y\big)^{\downarrow}$ *where X and Y are symbolic states.*

All these functions come together as the algorithm in Figure 3.4 on the following page. The algorithm does a normal forward exploration of the state space, unless a winning state is encountered. Then the algorithm simply puts it back on $Waiting$-list for re-examination, to see if some of its predecessors should also be included in the winning-set. The dependencies-list simply keep track of which visited states precedes a particular state. The winning-list is used to extract the final strategy, if one exists. It is populated from the the set Win^*, which is calculated using the functions mentioned above. The calculation simply takes all the winning states for the current location combined with all the states from the current location which have controllable edges to winning states. This is then used as the first argument for the safe-time predecessor function. The second argument of the function are all the states with uncontrollable edges from the current location, which does not lead to winning states. Using these two arguments in the Safe time-predecessor function gives all the safe states which precedes the current states. This is in turn intersected with the current state S, to give exactly the safe states that can be reached and leads to winning states, see Figure 3.3 on the previous page. [CDF+05]

```
TIGA-REACHABILITY()
$S_0 = (l_0, \mathbf{0})^{\uparrow}$
$Passed \leftarrow \{S_0\}$
$Waiting \leftarrow \{(S_0, a, S) | S = Post_a(S_0)^{\uparrow}\}$
$Win[S_0] \leftarrow \{S_0\} \cap (\{Goal\} \times \mathbb{R}_{\geq 0}^{|C|})$

while $(Waiting \neq \emptyset) \wedge (S_0 \notin Win[S_0])$ do
$(S, a, S') \leftarrow pop(Waiting)$
if $S' \notin Passed$ then
   $Passed \leftarrow Passed \cup \{S'\}$
   $Depend[S'] \leftarrow \{(S, a, S')\}$
   $Win[S'] \leftarrow \{S'\} \cap (\{Goal\} \times \mathbb{R}_{\geq 0}^{|C|})$
   $Waiting \leftarrow Waiting \cup \{(S', a, S'') | S'' = Post_a(S')^{\uparrow}\}$
   if $Win[S'] \neq \emptyset$ then
      $Waiting \leftarrow Waiting \cup \{(S, a, S')\}$

   else $Win^* \leftarrow Pred_t(Win[S] \cup \bigcup_{S \xrightarrow{c \in Act_c} T} Pred_c(Win[T]),$
        $\bigcup_{S \xrightarrow{u \in Act_u} T} Pred_u(T \setminus Win[T])) \cap S$
        if $Win[S] \subset Win^*$ then
           $Waiting \leftarrow Waiting \cup Depend[S]$
           $Win[S] \leftarrow Win^*$
           $Depend[S'] \leftarrow Depend[S'] \cup \{(S, a, S')\}$
```

Figure 3.4: TIGA-reachability.

When the algorithm has finished running, the winning-list contains states which are safe according to the property. If no strategy was found $Win[S_0]$ will be empty. If $Win[S_0]$ is not empty then a strategy can be extracted, by starting in S_0 and only include all the transitions which leads to a state in the winning-list, then do the same for all these states and so on. This will result in a mapping from states to transitions.

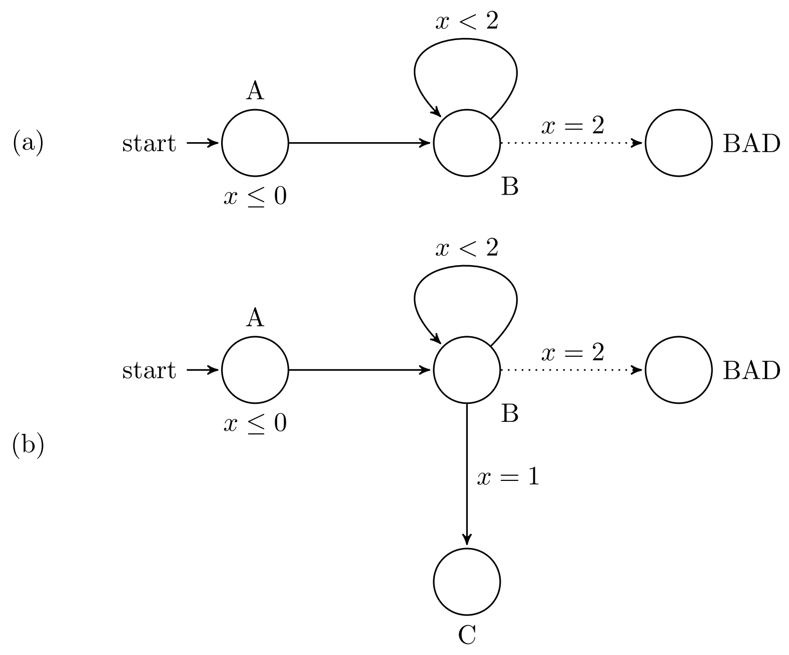


Figure 3.5: **a**: A simple model demonstrating Zenoness. **b:** A model with Zenoness, which can be avoided.

3.4 Zenoness

We want to avoid strategy which leads to Zeno-runs, because they will allow the strategy to make an infinite amount of discrete transitions in an finite amount of time. Earlier in this chapter we simply did not allow Zeno-runs, but how can they be avoided in reality?

To illustrate the problem consider the models in Figure 3.5. The goal here is to always make sure that you never enter the BAD state, in UPPAAL this can be expressed as:

- `control: A[] not BAD`

To avoid the BAD state in Figure 3.5 (a), the strategy is to first take the transition from A to B, and then keep taking the transition to B without delaying forever. This yields a Zeno-run.

The above strategy can of course not be realised in the real world, so a method to avoid it is necessary. The idea is to change the model, to include a transition from B leading to a new location, C, see Figure 3.5 (b). The changed model is then run in parallel with the process seen in Figure 3.6 on the next page, together with a fitting property which forces a constant amount of time to elapse between discrete transitions.

The construction in Figure 3.6 on the following page ensures that there are exactly one time unit between changing the location from D to E and visa versa, and that this change will always happen. The following property ensures the desired behaviour:

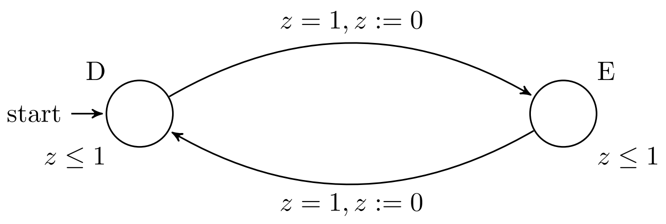


Figure 3.6: A simple model to help remove Zenoness.

- `control: A[] (not BAD and A<> D)`

The `A<> D` part of the property does the whole trick. After exactly one time unit the model in Figure 3.6 is forces to location E, at this point in time the model in Figure 3.5 on the preceding page (b) will be at location B. In order to uphold the property it must move to location C. If it stays at B, then at exactly two time units when the second model moves to D, in according to the property, the uncontrollable transition to BAD will be enabled, and therefore violating the property.

3.5 Pruning the strategies

To ensure that we generate as compact a strategy as possible, we must remove any non-reachable states which are included during the back-propagation.

TIGA already returns the shortest path that leads to or stays within the winning states when taking all the possible uncontrollable transitions from a given state. But consider a situation where both a controllable and uncontrollable transition leads to the winning states. A generated strategy can dictate taking the controllable transition before the uncontrollable becomes enabled, and thus a section of the strategy that takes care of the situation, where the uncontrollable transition has been taken, is not necessary. However, when back-propagating the winning states the TIGA algorithm will in some cases include a strategy for the uncontrollable transition because it leads to the same winning symbolic state.

Consider the model in Figure 3.7 on the next page, where the goal is to reach the END location. From state A, the guards ensures that the controllable transition can only be taken before the uncontrollable transition becomes enabled. But the generated strategy includes a rule for location B2 even though taking the controllable transition from A prevents this situation from ever occurring. The reason is that both B1 and B2 leads to the goal with controllable transitions, but more importantly the time zone generated when taking the B2 path is a a subset of the zone in the B1 path. Thus, when back-propagating from the END location the controllable transition from B2 to END will be added to the strategy.

In order to remove these unwanted parts of the strategies we have developed a relatively simple algorithm that uses the already generated state space and strategy graph in UPPAAL TIGA . The idea is to do a forward exploration from the initial state and only adding the uncontrollable transitions if they

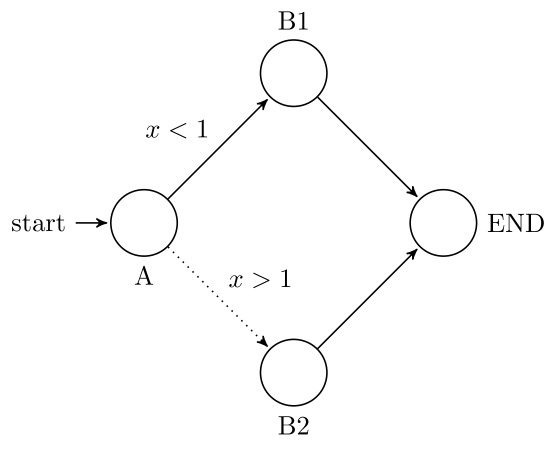


Figure 3.7: A simple model which illustrates the problem

are enabled at the same time or before a winning controllable transition. The algorithm in Figure 3.8 on the following page shows the process. It uses three lists WAITING, PASSED and STRATEGY. The WAITING list contains states that have to be explored, the PASSED list contains states that have been fully explored and the STRATEGY list contains the whole original strategy.

The elements in the STRATEGY list is formed of a discrete part (the locations), a time zone, and the action which leads to the time zone. From the already explored state set the initial state is put on the WAITING list. In the main loop the WAITING is popped. The popped state is compared with the states in the STRATEGY. If a state is found that shares the same discrete part and has an intersecting time zone, then all successors from the popped state are also compared with the states in the STRATEGY. Controllable transitions are explored only if they are represented in the strategy, and the uncontrollable transitions are only explored if they are enabled in the time zone found in the strategy. In line 12, the function $Pred_u$ returns the symbolic state in (l, Z) which can reach (l', Z') by an uncontrollable transition. When all the successors are explored the state is put on the PASSED list.

The last part of the algorithm removes the states in the STRATEGY that are not present in the passed list.

We have implemented the minimizeCurrent algorithm in UPPAAL TIGA for both DBMs and CDDs, the patch is available upon request, if legal access to the UPPAAL-source is possessed.

In this chapter we have looked at how to model systems using two player games. The games are a nice way to look at systems which interact with an unpredictable environment, as the environment can be seen as the opponent of the controller. This ensures that a possible strategy must cover all possibilities, since the environment will always have precedents over the controller when a transition is enabled.

We have also looked into the synthesis of strategies for these games, and how the model checker UPPAAL TIGA implements this feature.

```
minimizeCurrent()
$WAITING \leftarrow \{(l_0, 0)\}$
$PASSED \leftarrow \emptyset$
$STRATEGY \leftarrow \{((l_0, 0), Act_c \cup \lambda), \ldots, ((l_n, v_n), Act_c \cup \lambda)\}$

while $WAITING \neq \emptyset$ do
$(l, Z) \leftarrow$ pop($WAITING$)
for each $(k, Y, \Rightarrow_b)$ in $STRATEGY$ do
if $l = k$ and $Z \cap Y \neq \emptyset$ then
   for each $(l', Z') : (l, Z) \Rightarrow_a (l', Z')$ do
   if $\Rightarrow_b = \Rightarrow_a$ then
      $WAITING \leftarrow WAITING \cup \{(l', Z')\}$
      else if $\Rightarrow_a \neq controllable$ and $Pred_u(l', Z') \cap (k, Y) \neq \emptyset$ then
             $WAITING \leftarrow WAITING \cup \{(l', Z')\}$
$PASSED \leftarrow PASSED \cup \{(l, Z)\}$

for each $(k, Y, \Rightarrow_b)$ in $STRATEGY$ do
if $k \notin PASSED$ then
   $STRATEGY \leftarrow STRATEGY \setminus \{(k, Y, \Rightarrow_b)\}$
```

Figure 3.8: minimizeCurrent.

While looking into strategy synthesis for timed game automata, we ran into a particularly nasty type of strategies. Namely strategies, which leads to Zeno-runs. So we looked into how to avoid such unwanted behaviour. The solution involves changing the model in a way, which avoids the Zeno-runs all together.

Lastly we found an example of a model configuration, which yielded a strategy containing, under the strategy, unreachable states. This possibly gives an unnecessarily large strategy, and to avoid this we developed an algorithm for removing the unreachable states.

In the next chapter, we will look further into decreasing the size of strategies. This will be done for a specific system, in this case the system is a real life brick-sorter, build from LEGO, and controlled by a LEGO NXT. We will model the system in several ways, to determine the model which yields the most compact strategy.

Chapter 4

Model experiments

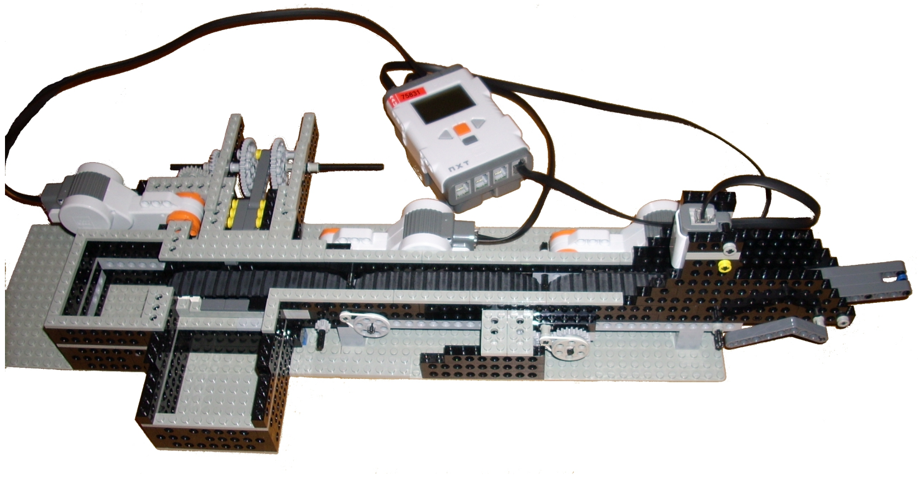


Figure 4.1: Brick-sorter build in LEGO.

In this chapter we will, through experiments, try to identify ways to reduce the generated strategy by applying various modelling tricks. We will start by constructing a model reflecting the real life brick-sorter seen in Figure 4.1. The first model will include all possible information, ex. where a brick is at all times. This full model will be the starting point for the following modelling tricks we will present, in order to reduce the generated strategy.

The modelling is done in UPPAAL TIGA [BCD$^+$06], so the models will be described in the context of this tool. The experiments were run on an Dell PowerEdge 2950, 2x2.5 GHz CPU (Quad Core Intel Xeon), with 32 GB RAM. UPPAAL can only utilise 4GB RAM, and one CPU core.

The metrics for the tests will be how compact the strategy is for a given number of concurrent bricks, as well as the time it takes to generate the strategy. The compactness is measured by looking at the number of symbolic states in the strategy, where a controllable action must be taken, disregarding wait actions. This in turn gives the number of branches needed to implement the strategy in a naive way.

Each model consists of a number of processes, modelling bricks, run in parallel with a controller processes. Each brick process represents an actual brick on the conveyor belt. This means that a strategy generated with three brick processes, can handle a maximum of three simultaneous real bricks. The controller process can receive information from the bricks, as well as act upon the bricks. The controller is naturally the same for all the experiments, so we can compare the strategies, and can be seen in Figure 4.2.

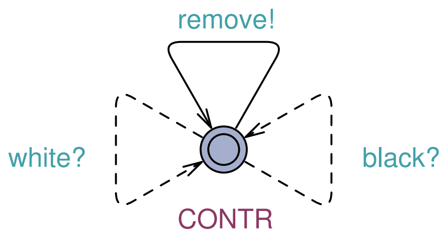


Figure 4.2: **Controller:** the controller process.

The controller is a very simple process, with only one location, and thus the controller is memoryless. It can act upon the bricks by sending a push-command. The corresponding edge in the brick model is controllable, so the controller can decide when a brick is pushed off. There are also two colour events, one for black and one for white, this tells the controller that a brick passed under the sensor, and which colour it reported.

Because the controller is the same for all the experiments, then naturally there will also be some transitions, and locations which are the same for all the models. The transitions are the ones which synchronise with the controller, that is the transitions which fires when a black or a white brick is sensed, and the transition taken, when the controller decides to push a brick. There are also two locations which need to be present in all the brick models. These are the OFF and the END locations, because the property we use to generate the strategies depend on them. The property will also be the same for all the experiments, for us to compare the results. The property can be seen in Figure 4.3, the property includes a colour variable which must also be in all the models. The property quite simply says that when a brick is in the location OFF, then the colour variable must be white, and if it is in the location END the colour must be black.

```
control: A[] forall(i:id_t)
    (Brick(i).OFF imply Brick(i).colour ==  WHITE) and
    (Brick(i).END imply Brick(i).colour ==  BLACK)
```

Figure 4.3: The property, which the generated strategy must comply with.

A number of constants are used in all the models, in order to ease comparison, and changing timing for all the models at once. These constants can be seen in Figure 4.4 on the facing page, and will be explained as they are used.

```
// Global declarations
const int SENSE_TIME_MIN = 8;
const int SENSE_TIME_MAX = 10;

const int PISTON_TIME_MIN = 100;
const int PISTON_TIME_MAX = 120;

const int TOTAL_TIME_MIN = SENSE_TIME_MIN + PISTON_TIME_MIN;
const int TOTAL_TIME_MAX = SENSE_TIME_MAX + PISTON_TIME_MAX;

const int END_TIME = 2;

const int BLACK = 0;
const int WHITE = 1;

const int RES = 1;

const int N = 4;

typedef int[0, N-1] id_t;

chan remove, white, black;
broadcast chan tick;

int turn;

int next() {
    if (turn == N-1) {
        return 0;
    }

    return turn + 1;
}
```

Figure 4.4: Constants and the $next() - function$ used throughout the experiments.

4.1 The full model

The first experiment is done on the full model to establish benchmark for the various modelling tricks. The full modelled, referred to as “Brick_FO_CYC” since it is fully observable, and cyclic, can be seen in Figure 4.6 on the next page and its local declarations, can be seen in Figure 4.5.

```
// Local declarations
clock x;
bool colour;
```

Figure 4.5: The local declarations for the full model.

The full model consists of a number of locations, which together with the time zones constrained by the guards an invariants describes all the possible states of the system. The IDLE location represents the states, where the block is not on the conveyor yet. All the states where the brick is on the conveyor, but not under the sensor yet, are all in the ON location. The SENSOR and SENSED locations cover all the states in which a brick is under the sensor, and until it is reaches the piston. The PISTON location is for the states when the brick is at the piston, and it is possible for the controller to push the brick of the belt. The OFF and END locations have already been covered above.

As there is no guard on the transition from IDLE to ON, the bricks can be put on the belt at any time. The SENSE_TIME_MIN and SENSE_TIME_MAX used the guard and invariant for the ON to SENSOR transition, represents the minimum and maximum time the brick takes to reach the sensor, from the beginning of the conveyor. The PISTON_TIME_MIN and PISTON_TIME_MAX is the minimum and maximum time it takes a brick to reach the piston from the sensor, and is therefor used as guard and invariant for the SENSED to PISTON transition. Finally the END_TIME is the time it takes for a brick to pass the piston, which means is can no longer be pushed by the controller.

The model is cyclic, because of the transitions from END and OFF back to IDLE. This means that when the brick has gone through the system and ended up being either pushed off, or went all the way to the end, and the machine is ready to take sort another brick. Another thing to notice, is that the brick processes are not ordered in any way, so whenever a brick is introduced into the system process representing it in the model, is selected randomly from the processes which are in the IDLE states.

For the testes an incrementing number, starting from one, of processes modelling the full system is run in parallel with the controller process. The system is is described in TIGA as:

```
Brick(const id_t ID) = BRICK_FO_CYC(ID);

system Controller, Brick;
```

The number of concurrent bricks is controlled by the constant N, seen in Figure 4.4 on the previous page, and corresponds directly to the N in the results table. The tests were run with N from one, up until the machine ran out of

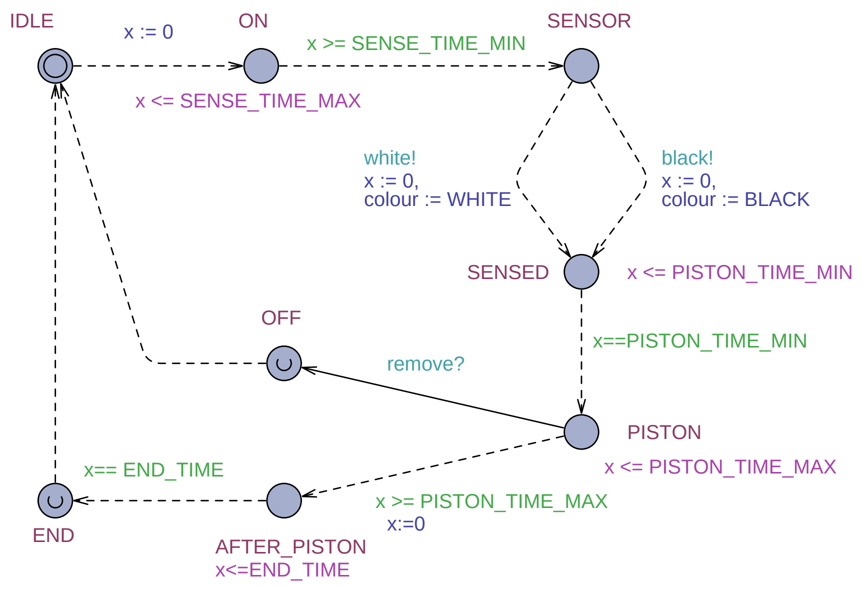


Figure 4.6: **Brick_FO_CYC(const id_t ID):** Full model, with cycles.

memory, and the size of the strategy as well as the generation time has be noted, with and without minimisation algorithm from the previous chapter, in Table 4.1. The result from this experiment will be the measure of comparison for the rest of the experiments.

N	Size, no minimisation	time	Size, with minimisation	time
1	1	0.01s	1	0.01s
2	25	0.02s	25	0.02s
3	470	0.36s	470	0.39s
4	7989	32.87s	7989	34.92s
5	–	–	–	–

Table 4.1: Results for the full model.

4.2 Partial observation

So far we have only looked at controller synthesis in systems with full observability. That is systems where the exact state and time is always known, to the controller. However as the controller-process, in reality, cannot observe the full state of a system, for example due to a limited number of sensors, one might view a set of states as being indistinguishable from the controllers point of view. This introduce the notion of observations, which are the state changes which the controller can actually observe.

If a strategy for a given system depends on full observability, that is where all state information is accessible to the controller, but the controller cannot separate some of the different states, due to limited observability. Then the strategy will not work as the controller might not be able to separate some states requiring different actions to win. A way to avoid this is to change the model to include only observable locations.

Partial observability in timed game automata, was introduced in [BMP03] and [CDL$^+$07]. The first article shows that controller synthesis in the general case is undecidable, however it also shows that limiting the systems resources regains decidability. The second article looks at a particular type of strategy and impose some limitations to avoid the undecidablity result. Common to both methods is that they incorperate partial observability into the model checking algorithm. This is done by defining what is obersvabel and what is not.

In order to better realise a strategy in a real world system and subsequently make the model checking and strategy generation faster, we will introduce a formalism to alter a detailed model and compress it. This is achieved by abstracting away the unobservable states.

In the case of the bricker sorter model, the locations which are observational equivalent can be seen in Figure 4.7 on the next page, which when collapsed, reduced to the model seen in Figure 4.8 on page 46. The blue locations are collapsed into the location called READY, the red locations are collapsed into the location SENSED, the orange locations are collapsed into the location END and lastly the green location is unchanged as OFF. The local declarations for this model can be seen in Figure 4.9 on page 46

This test runs an incrementing number of processes, starting from one, of the reduced brick in parallel with the controller process. The system is is described

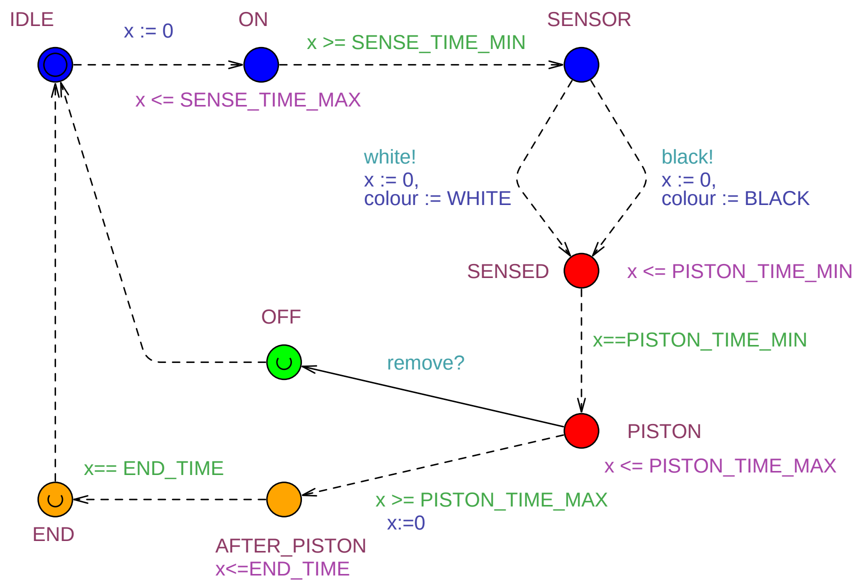


Figure 4.7: The full model with observational equivalent locations in the same colour.

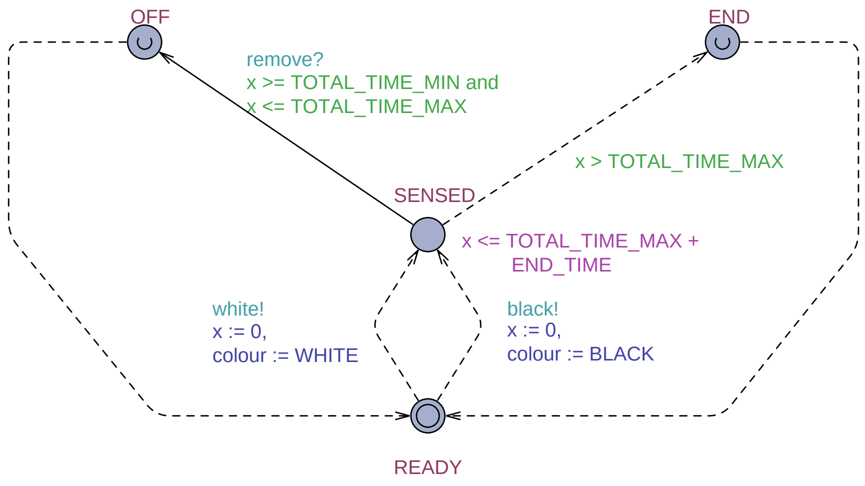


Figure 4.8: **Brick_PO_CYC(const id_t ID):** Reduced model, with cycles.

```
// Local declarations
clock x;
bool colour;
```

Figure 4.9: The local declarations for the reduced model.

in TIGA as:

```
Brick(const id_t ID) = BRICK_PO_CYC(ID);

system Controller, Brick;
```

The results have been noted in Table 4.2.

N	Size, no minimisation	time	Size, with minimisation	time
1	1	0.01s	1	0.01s
2	12	0.01s	12	0.02s
3	108	0.04s	108	0.04s
4	864	0.50s	864	0.63s
5	6480	18.78s	6480	19.84s
6	46656	1471.17s	46656	1489.42s
7	–	–	–	–

Table 4.2: Results for the reduced model.

This modelling technique is based on the theory in the next subsection.

4.2.1 Alternating simulation

In the articles mentioned earlier partial observability is introduced directly into the model checking algorithm. The ideas behind have been integrated into a development version of UPPAAL TIGA, with partial observations. However at the time of this writing, the tool is not ready for production use. Instead we have chosen to look at partial observability as a modelling abstraction. This entails that some transitions in a given model leads to observable changes, and some do not. The states which are connected only by unobservable transitions can then be collapsed into one, in a abstract representation of the original model.

Every controllable action leads to an observation change. The argument behind this is simple, if the controller at some state issues a controllable action the controller must be aware that a state change has happened. Since only the controller can take a controllable transition it will always know when one has occurred.

Every winning state in the abstract model must belong to its own observation. The reason behind this is simple. In order to find a winning strategy we must have a property. The property for the abstract model must be the same as for the concrete model in order to ensure an equivalent strategy and therefore we must be able to declare the same bad and good states.

Lastly sensory input can be modelled as uncontrollable channels that communicates with the controller. Thus uncontrollable transitions which are not internal leads to changes in the observation.

Based on the discussion above we need a set of rules that guarantee that if we have a winning strategy for the abstract model it implies that there exist a strategy for the concrete system.

First we define a set of weak transitions that consists of an ordinary transition and several τ transitions, which are unobservable internal transitions in a model.

Definition 4.2.1 (Weak Transitions)
The weak transitions consists of a transition or a series of transitions, where transitions can be either controllable, delays or τ-transitions, and is defined as:

- $\overset{c}{\Rightarrow}$:
  $\xrightarrow{\tau_1}\xrightarrow{\tau_2}\cdots\xrightarrow{\tau_k}\xrightarrow{c}$ *, where* $k \geq 0$
- $\overset{d}{\Rightarrow}$:
  $\xrightarrow{\tau}\cdots\xrightarrow{d_1}\cdots\xrightarrow{\tau}\cdots\xrightarrow{d_2}\cdots\xrightarrow{\tau}\cdots\xrightarrow{d_k}$*, where there can be any number of* τ *transitions and* $\sum_{i=1}^{k} d_i = d$.

We will formulate an alternating simulation and show that if an abstract model follows the relation, a winning strategy w.r.t. reachability exists for the concrete system. To do this we put some restrictions on the models. The concrete system must not have any τ-divergence. Furthermore we assume determinism for all actions including delay actions. The reason for no τ-divergence can be seen in Figure 4.10. Suppose we have a strategy for (b), but then we can not be sure to also have a strategy for (a), because the environment could force the system to go to B every time it is in A thereby preventing us from reaching D.

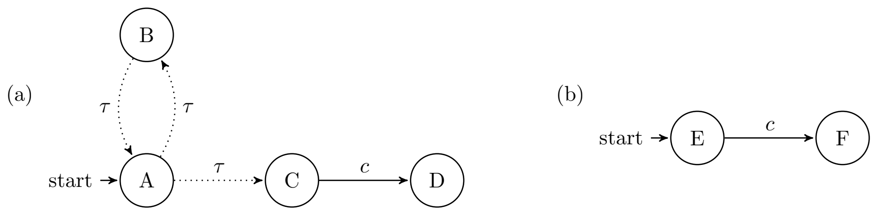


Figure 4.10: **a:** An example of a concrete system. **b:** The abstraction of (a). E is an abstraction of A,B and C. F is an abstraction of D. F and D are the winning states.

We are now ready to define an alternating simulation, which can determine if a model is an abstraction of a concrete system.

Definition 4.2.2 (Alternating simulation)
Let s_c be a state for the concrete system and let s_a be a state for the abstract model. Furthermore let W_c be the winning states for the concrete system and let W_a be the winning states for the abstract. An alternating simulation between states is a relation $s_c\ R\ s_a$ with the following rules:

1. $s_a \in W_a \quad \Rightarrow \quad s_c \in W_c$.
2. a. $s_c \xrightarrow{u} s'_c \quad \Rightarrow \quad s_a \xrightarrow{u} s_a \quad \wedge \quad s'_c\ R\ s'_a$.
   b. $s_c \xrightarrow{\tau} s'_c \quad \Rightarrow \quad s'_c\ R\ s_a$.

c. $s_c \overset{d}{\Rightarrow} s'_c \quad \Rightarrow \quad s_a \xrightarrow{d} s'_a \quad \wedge \quad s'_c \; R \; s'_a.$

3. *a.* $s_a \xrightarrow{c} s'_a \quad \Rightarrow \quad \forall s'_c.\; s_c \overset{c}{\Rightarrow} s'_c \quad \Rightarrow \quad s'_c \; R \; s'_a.$

To illustrate that a state s_2 is an abstraction of state s_1 we write $s_1 \leq s_2$. This leads to the following definition.

Definition 4.2.3 (Abstraction)
Let s_c be a state for the concrete system and set s_a be a state for the abstract model.

- *$s_c \leq s_a$ if there exist an alternating simulation R such that $s_c \; R \; s_a$.*

We assume that there is at most one abstraction for the same concrete state, in other words $\forall s_c, s_a, s'_a$ where $s_c \leq s_a \;\wedge\; s_c \leq s'_a \;\Rightarrow\; s_a = s'_a$.

Definition 4.2.4 (Abstraction function)
Let S_a be states in the abstract model and let S_c be states in the concrete system. An abstraction function $F : S_c \to S_a$ is a function that maps a state to its abstraction, such that whenever $s_c \leq s_a$ then $s_a = F(s_c)$.

From Chapter 3 we know that a strategy is a function $C : S \to Act_c \cup \{\lambda\}$ where for each state an action or wait is chosen. The strategy defines a series of winning runs. We wish to show that if we find a strategy for the abstract model then there exists a strategy for the concrete system. To show this, we define when a strategy in the concrete system corresponds with a strategy in the abstract model.

Definition 4.2.5
Let $F : S_c \to S_a$ be an abstraction function. Whenever $C_a : S_a \to Act_c \cup \{\lambda\}$ is a strategy for the abstract model then $F(C_a)$ is the strategy for the concrete system defined by:

$$F(C_a)(s_c) = C_a(F(s_c)),$$

and $F(C_a)(s_c) = C_c(s_c)$.

The definition above tells us when a strategy in the abstract model corresponds to a strategy in the concrete system. At a certain state s_c, in the concrete system, the action performed by the concrete strategy must be the same as the action performed by the abstract strategy in the state s_a whenever $s_c \leq s_a$. We are now able to formulate the following theorem:

Theorem 4.2.1
Given C_a is a winning strategy w.r.t reachability for the abstract model, then $F(C_a) = C_c$ (defined by $C_c(s_c) = C_a(F(s_c))$) is a winning strategy w.r.t reachability for the concrete system.

Proof. Let

$$\beta_c = s_0 \xrightarrow{d_0} s'_0 \xrightarrow{a_0} s_1 \xrightarrow{d_1} s'_1 \xrightarrow{a_1} s_2 \xrightarrow{d_2} \cdots$$

be a run for the concrete system according to C_c. We claim that

$$\beta_a = F(s_0) \xrightarrow{d_0} F(s'_0) \xrightarrow{\widehat{a_0}} F(s_1) \xrightarrow{d_1} F(s'_1) \xrightarrow{\widehat{a_1}} F(s_2) \xrightarrow{d_2} \cdots$$

is a run in the abstract model according to C_a, where $\widehat{\tau} = \varepsilon$ and $F(s'_i) = F(s_{i+1})$ if $a_i = \tau$. We show that for all i:

1. $F(s_i) \xrightarrow{d_i} F(s'_i)$ according to C_a.
2. $F(s'_i) \xrightarrow{\widehat{a_i}} F(s_{i+1})$ according to C_a.

For (1) we note that $s_i \xrightarrow{d_i} s'_i$ according to C_c. Thus $C_c(t_i) = C_a(F(t_i)) = \lambda$ whenever $s_i \xrightarrow{d'_i} t_i$ with $d'_i < d_i$. From this it follows that (1) is according to C_a.

For (2) we have 3 cases for a_i:

- $\mathbf{a_i} = \tau$, then $s'_i \xrightarrow{\tau} s_{i+1}$ and $F(s'_i) = F(s_{i+1})$ by the alternating simulation (2.b.) and the functionality of $\leq$.
- $\mathbf{a_i} = \mathbf{u}$, then $s'_i \xrightarrow{u} s_{i+1}$. Since $s'_i \leq F(s'_i)$ it follows that $F(s'_i) \xrightarrow{u} t_{i+1}$ with $s_{i+1} \leq t_{i+1}$. By the functionality of $\leq$ we have that $t_{i+1} = F(s_{i+1})$.
- $\mathbf{a_i} = \mathbf{c}$, then $s'_i \xrightarrow{c} s_{i+1}$. Thus $C_c(s'_i) = c = C_a(F(s'_i))$. Then $F(s'_i) \xrightarrow{c} t_{i+1}$. By the alternating simulation (3.a.) it follows that $s_{i+1} \leq t_{i+1}$ and by the functionality of $\leq$ we have $t_{i+1} = F(s_{i+1})$.

Now, since β_a is a run in the abstract model according to C_a, where C_a is a winning strategy w.r.t reachability, it follows that for some n either $F(s_n)$ or $F(s'_n)$ is a goal state ($F(s_n) \in W_a \ \vee \ F(s'_n) \in W_a$). Since $s_n \leq F(s_n)$ and $s'_n \leq F(s'_n)$ it follows that either s_n or s'_n is goal state ($s_n \in W_c \ \vee \ s'_n \in W_c$) of the concrete system. □

4.3 Sequential

Another way to reduce verification time and strategy size is to make sure any identical processes are not explored in all possible combinations, this is called symmetry reduction. This can be done by running the processes in the same sequence every time. Ex. process one always runs first, then the second, then the third, and so on.

This can be implemented in a model by introducing a global variable, in our case called turn, this variable is set to one to begin with. Then each of the processes need a guard on all the edges going out from the initial location, with the expression: $turn == ID$, where ID is the process-id. This ensures that the first process is the only one which can move out of the initial location. To make the processes run in sequence, so the first process represents the first brick, the second process the second brick and so on, the $turn$-variable is set to the next process-id. This ensures that the processes always run in the same sequence. The $turn$-variable, as well as the $next()$-function can be seen in Figure 4.4 on page 41. An implementation of this can be seen in Figure 4.11. The local declarations for this model can be seen in Figure 4.12.

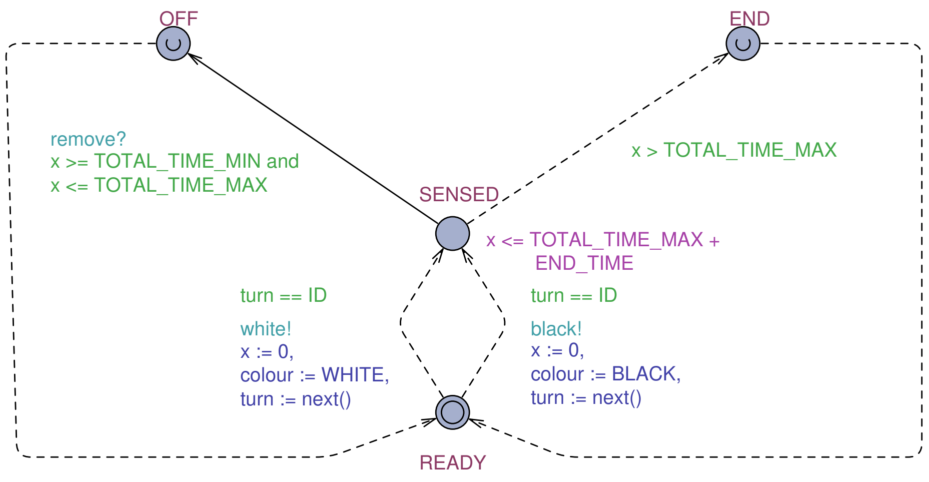


Figure 4.11: **Brick_PO_CYC_SE(const id_t ID):** Reduced model, with cycles, sequential.

```
// Local declarations
clock x;
bool colour;
```

Figure 4.12: The local declarations for the sequential model.

As with the other tests an incrementing number of processes, modelling the sequential behaviour, is run in parallel with the controller process. The system is is described in TIGA as:

```
Brick(const id_t ID) = BRICK_PO_CYC_SE(ID);

system Controller, Brick;
```

The results have been noted in Table 4.3 on the next page.

This method is usable because, if there exists a winning strategy for the symmetry reduced model, then there also exists a winning strategy for the unreduced model.

N	Size, no minimisation	time	Size, with minimisation	time
1	1	0.01s	1	0.01s
2	19	0.01s	17	0.01s
3	195	0.04s	165	0.04s
4	1531	0.28s	1249	0.24s
5	10327	2.24s	8200	2.41s
6	63167	20.65s	49759	27.24s
7	360862	175.29s	283284	186.46s
8	–	–	–	–

Table 4.3: Results for the sequential model.

A winning strategy for the unreduced model can be constructed from the winning strategy for the reduced model. This can be done because the symmetry reduction simply remove runs, which are already covered by another process. The fact that all processes representing the bricks are identical, makes original runs identical except for process-id. This means that if we use the reduced strategy on the unreduced state space, then it will only be able to handle one path through the reduced parts in the state space. However since what was removed from the model by the symmetry reduction, was symmetric, then no matter which path is taken through the reduced part the same actions should be taken. So whenever reducible states are reached in the unreduced model, using the reduced strategy, simply follow the strategy, ignoring the process-id. The fact that the strategy of the ordered system is a subset of the unordered is the reason why this technique might yield a smaller strategy.

4.4 Acyclic

Yet another way to reduce verification time and strategy size, is to remove cycles from the model. This helps by removing the need for the verifier to make multiple runs over the same locations.

This is implemented in a model, simply by removing the edges leading into the initial location. This can be seen in Figure 4.13 on the facing page, with its local declarations in Figure 4.14 on the next page.

As with the other tests, an incrementing number, starting from one, of processes modelling the acyclic behaviour is run in parallel with the controller process. The system is is described in TIGA as:

```
Brick(const id_t ID) = BRICK_PO_ACYC(ID);

system Controller, Brick;
```

The results have been noted in Table 4.4 on the facing page.

This method is usable because, a winning strategy for a cyclic model can be constructed from a winning strategy from a cyclic model with cycles removed.

The construction is quite simple. The problem with a acyclic model, is that once a process has reached one of the winning states, then the process cannot be reused. The solution is simply to reuse the processes. That is when the last process has reached a winning state, simply start using the processes over.

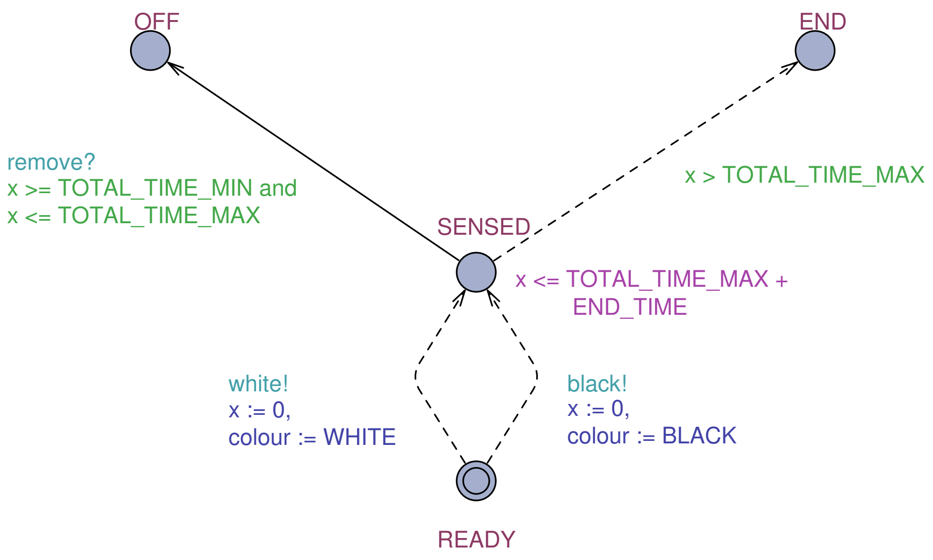


Figure 4.13: **Brick_PO_ACYC(const id_t ID):** Reduced model, without cycles.

```
// Local declarations
clock x;
bool colour;
```

Figure 4.14: The local declarations for the acyclic model.

N	Size, no minimisation	time	Size, with minimisation	time
1	1	0.01s	1	0.01s
2	10	0.01s	10	0.01s
3	75	0.02s	75	0.02s
4	500	0.23s	500	0.25s
5	3125	7.56s	3125	7.96s
6	18750	531.25s	18750	501.43s
7	–	–	–	–

Table 4.4: Results for the acyclic model.

4.5 Acyclic and sequential

The two methods above, the sequential and the acyclic, can be combined. This combines the advantages of both methods. Firstly removal of the cycles, ensures that the verification algorithm only has to search through a three structure, and thus the state space will be greatly reduced. As a further reduction, the sequential nature of the model, remove most of the interleaving, and thus reduces the state space further, as well as reducing the size of the strategy. The implementation of this can be seen in Figure 4.15, and the local declarations in Figure 4.16.

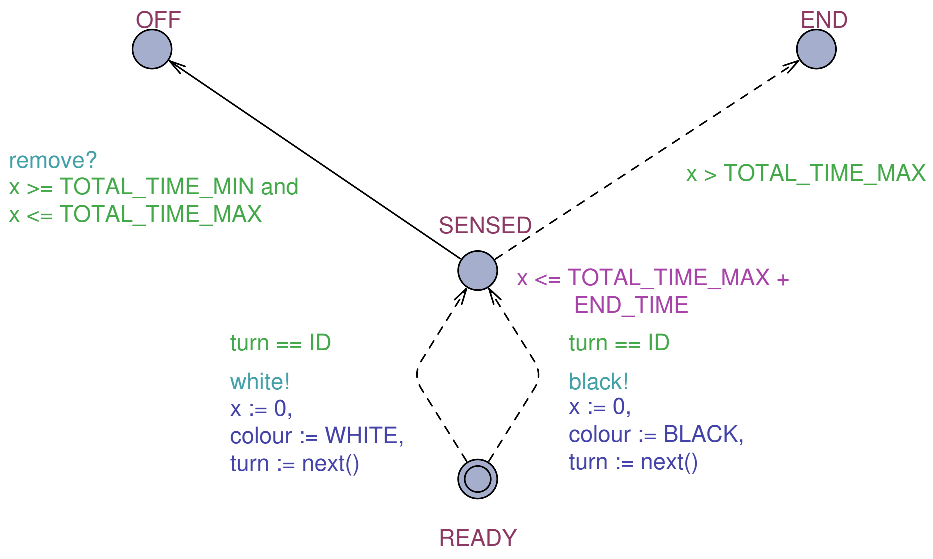


Figure 4.15: **Brick_PO_ACYC_SE(const id_t ID):** Reduced model, without cycles and sequential.

```
// Local declarations
clock x;
bool colour;
```

Figure 4.16: The local declarations for the acyclic and sequential model.

As with the other tests, an incrementing number, starting from one, of processes modelling the sequential and acyclic behaviour is run in parallel with the controller process. The system is is described in TIGA as:

```
Brick(const id_t ID) = BRICK_PO_ACYC_SE(ID);

system Controller, Brick;
```

The results have been noted in Table 4.5 on the next page.

N	Size, no minimisation	time	Size, with minimisation	time
1	1	0.01s	1	0.01s
2	8	0.01s	8	0.02s
3	43	0.01s	43	0.01s
4	194	0.03s	194	0.04s
5	793	0.19s	793	0.20s
6	3041	1.09s	3041	1.25s
7	11164	5.84s	11164	6.08s
8	39696	35.92s	39694	37.14s
9	138235	226.80s	138219	236.12s
10	–	–	–	–

Table 4.5: Results for the acyclic and sequential model.

4.6 Acyclic, sequential and discrete

The final modelling technique we will present is discretization of the model. This is done by adding a discrete clock, in the form of an integer variable. The discrete clock is then used instead of a normal clock, to guard discrete actions. In order for the discrete time to elapse a loop is added to all locations which synchronises with a clock process, see Figure 4.17. The clock process makes sure that the discrete clocks only increment every *RES*, time units, where *RES* is the resolution of the discrete clocks. The resolution of the discrete clocks have a large impact on the number of symbolic states generated, as well as the time spent, during the verification process. The lower the resolution, the more discrete states are generated, and there by defeating the advantage of using time zones.

This type of modelling reflects reality more closely than than our previous models, as a real life implementation of a strategy would run on a discrete controller, in the form of hardware and software. The model only allow the controller to take an action on discrete intervals, where as the environment can take actions at any time.

A discretization of the acyclic and sequential brick sorter model can be seen in Figure 4.18 on the next page, and its local declarations in Figure 4.19 on the following page. The model uses two clocks, x and y, where x is a discrete clock, and y is a normal clock. As discrete time only need to elapse at the location *Sensed* this is the only place the afore mentioned loop is added. The value of *RES* can not be higher than TOTAL_TIME_MAX, as this would allow the environment to disable the discrete action before it is enabled.

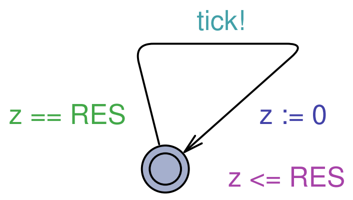


Figure 4.17: **Clock:** The process which ensures the discrete clock are only updateable at fixed times.

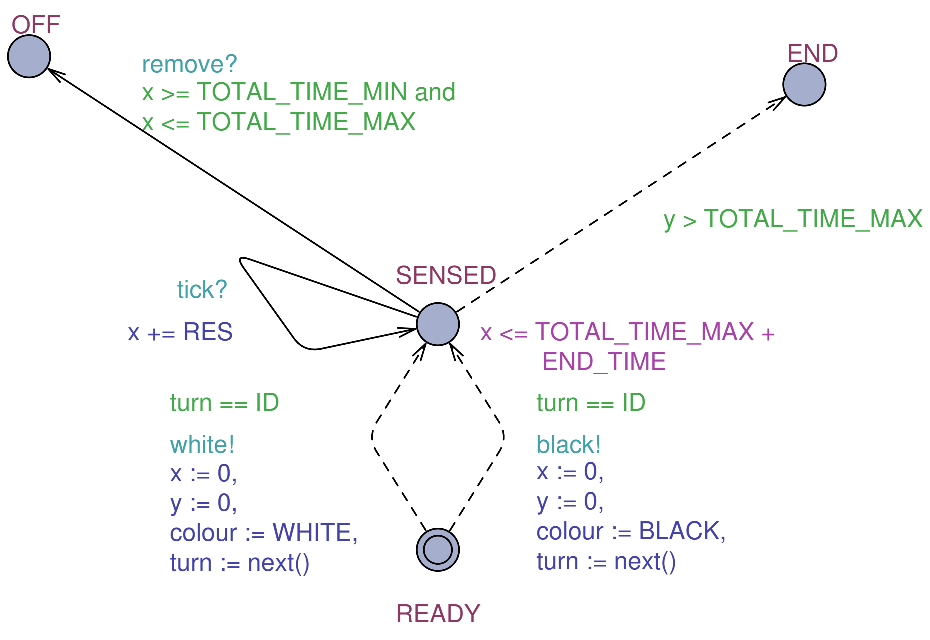


Figure 4.18: **Brick_PO_ACYC_SE_DIS(const id_t ID):** Reduced model, without cycles, sequential and discrete.

```
// Local declarations
int x;
clock y;
bool colour;
```

Figure 4.19: The local declarations for the acyclic, sequential and discrete model.

As with the previous tests, an incrementing number, starting from one, of processes modelling the sequential, acyclic and discrete behaviour is run in parallel with the controller process. A small difference is the addition of the *Clock*-process. The test is also run with different values of *RES*, to demonstrate the impact of the discrete clock resolution. The system is is described in TIGA as:

```
Brick(const id_t ID) = BRICK_PO_ACYC_SE_DIS(ID);

system Controller, Brick, Clock;
```

Several tests were run with resolutions: 1, 10, 65, and 130. The first with resolution one, see result in Table 4.6 on the next page. The state space explosion is hinted by the low number of concurrent processes which can be handled. This results from the enormous subdivision of the discrete states. The second test with resolution ten, Table 4.7 on the facing page, improves a little upon the subdivision problem. The last two testes were run with resolutions: half of TO-

N	Size, no minimisation	time	Size, with minimisation	time
1	312	0.01s	312	0.01s
2	49654	1.50s	49654	1.65s
3	5387673	244.37s	5374834	275.31s
4	–	–	–	–

Table 4.6: Results for the acyclic, sequential and discrete model, with resolution 1.

N	Size, no minimisation	time	Size, with minimisation	time
1	34	0.01s	34	0.01s
2	636	0.02s	626	0.02s
3	8291	0.34s	8258	0.37s
4	88587	5.42s	87271	5.86s
5	812348	68.57s	784237	73.74s
6	–	–	–	–

Table 4.7: Results for the acyclic, sequential and discrete model, with resolution 10.

TAL_TIME_MAX, and TOTAL_TIME_MAX, the results can be seen Table 4.8 and Table 4.9 on the following page respectively.

4.7 Comparison and conclusion

All the test results in have been plotted in Figure 4.20 on page 59, the vertical axis represents the number of symbolic states in the strategy, and the horizontal axis represent the number of brick processes. The discrete test plotted in the figure has a resolution of TOTAL_TIME_MAX, as this was the best of the discrete tests.

From the plot it is clear that the full model is clearly the worst model, the verifier runs out of memory after four brick-processes, and has a larger strategy than all the other models. It is also clear that the observational abstraction helps, as it can handle two processes more than the full model. When the cyclic model is made sequential it is interesting to note that it can handle one process

N	Size, no minimisation	time	Size, with minimisation	time
1	8	0.01s	8	0.01s
2	42	0.01s	42	0.01s
3	185	0.02s	183	0.02s
4	745	0.08s	715	0.09s
5	2850	0.47s	2591	0.49s
6	10552	2.81s	8863	2.90s
7	38171	15.59s	28927	16.67s
8	135591	83.09s	90751	84.92s
9	–	–	–	–

Table 4.8: Results for the acyclic, sequential and discrete model, with resolution half TOTAL_TIME_MAX.

N	Size, no minimisation	time	Size, with minimisation	time
1	6	0.01s	6	0.01s
2	26	0.01s	26	0.01s
3	101	0.01s	99	0.01s
4	375	0.05s	349	0.05s
5	1360	0.27s	1161	0.28s
6	4852	1.55s	3681	1.57s
7	17067	8.41s	11201	8.55s
8	59249	44.04s	32897	44.51s
9	203174	223.47s	93697	225.24s
10	–	–	–	–

Table 4.9: Results for the acyclic, sequential and discrete model, with resolution TOTAL_TIME_MAX

more than the corresponding non-sequential model, but with a larger strategy. This indicates that the sequential model generates fever symbolic states during verification, however it takes more symbolic states to represent the strategy. This might be explained by extra variable introduced to the model, which the strategy must be able to handle. If cycles are removed from the process, as with the acyclic model, then the strategy size is reduced considerably, however number of processes which the verifier can handle remains at six. Curiously when the acyclic model is made sequential, the number of processes go up to nine, and the number of symbolic states needed to represent the strategy goes down as well. This could be explained because the combination of the two tricks, will inherit the smaller state space from the sequential model and the smaller strategy from the acyclic model. Finally the acyclic, sequential and discrete model, in this case, needs the least amount of states to represent the strategy. However as seen in the previous tests, the size of the strategy, in the discrete case, depends on the resolution of the discrete clock, the higher the resolution the smaller the strategy. And since the largest possible resolution, for a given model, is dependent on the model it might not always be advisable to use discretization.

In the next chapter we will introduce a data structure called Hybrid CDDs, which allows for a very compact representation of a strategy. The CDD is an acyclic graph, which represents a decision tree for a given strategy. UPPAAL TIGA is able to output strategies as CDDs, which we will then parse and use to generate code. This code can then control the brick-sorter, according to the strategy proposed by TIGA.

We have chosen to work on the cyclic model with observational abstraction, even though this model does not generate the most compact strategies. It is however, by far the easiest to implement, as the strategy does not need any post processing, which is the case with the later model.

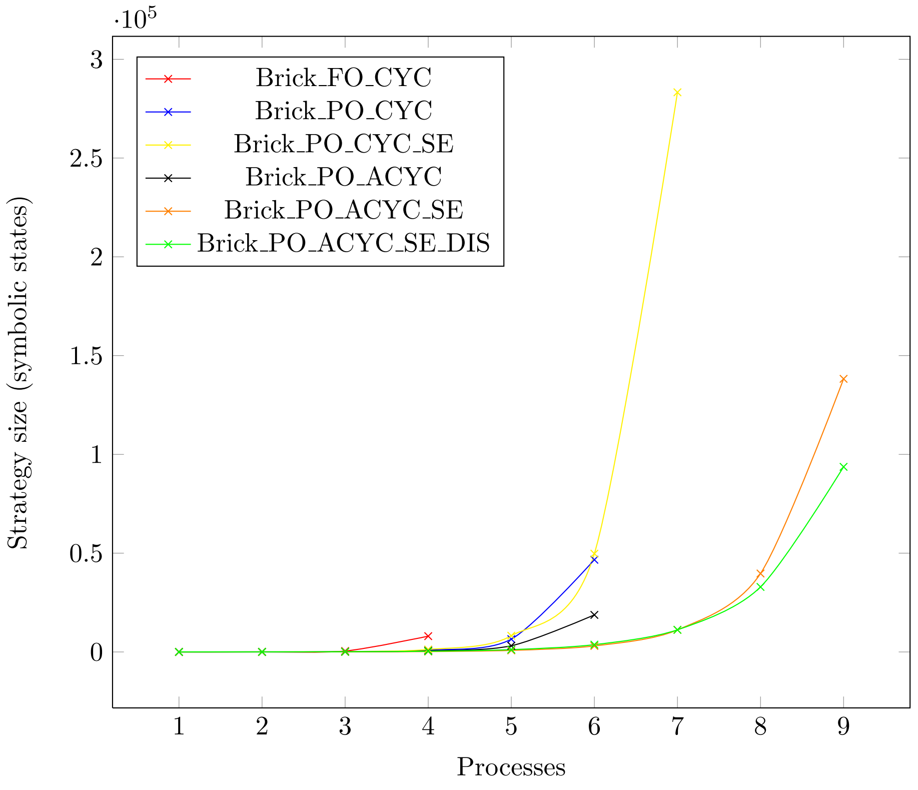


Figure 4.20: Comparison of the test results.

Chapter 5

Implementation

UPPAAL has the feature of outputting the strategies as a hybrid clock difference diagram. The biggest reason for using this data-structure is that looking up actions is linear in the number of clocks in the system. In short, instead of looking for the right set of locations and their associated federations in the complete strategy, you simply read the current clock values, the discrete variables and locations and go through the CDD, which gives the appropriate action.

5.1 Clock difference diagrams

This section gives an introduction to CDDs, and is based on [LWYP98] and [BLP$^+$99]. A CDD is a compact representation of a decision diagram for finite unions of zones. A CDD node, branches with respect to intervals of the reals for a given clock difference. The size and number of intervals is not fixed. However there can only be a finite number of intervals, and they must be compatible with the clock constraints. This means that all intervals have integer bounds, and any bound can either be included or not. A CDD works in the following way: Take a valuation, and follow the unique path along which the constraints given by type and interval are fulfilled by the valuation. Remember that the clocks are denoted $C = \{x_1, \dots, x_n\}$. A type is a pair (i, j) where $1 \leq i < j \leq n$. The types has a linear ordering denoted by $\sqsubseteq$. $(i, j) \sqsubseteq (i', j')$ iff either $j < j'$ or $j = j' \wedge i \leq i'$. In order to relate interval and types to constraints, the following notation is used:

- Given a type (i, j) and an interval Int of the reals, then $Int(i, j)$ denotes the clock constraint having type (i, j) which restricts the value of $x_i - x_j$ to the interval Int.
- Given a clock constraint g and a valuation v, by $g(v)$ the application of g to v is denoted, i.e. the boolean value derived from replacing the clocks in g by the values given in v.

Note that the notation is typically used jointly, i.e. $Int(i, j)(v)$ expresses the fact that v fulfils the constraint given by the interval Int and the type (i, j).

For example, if the type is $(2, 1)$ and $Int = [3, 5)$, then $Int(i, j)$ would be the constraint $3 \leq x_2 - x_1 < 5$. For v, where $v(x_2) = 9$ and $v(x_1) = 5.2$, $Int(i, j)(v)$ would be true, but for v', with $v'(x_2) = 3$ and $v'(x_1) = 4$, $Int(i, j)(v')$ is false.

This leads to the following definition of a CDD:

Definition 5.1.1 (Clock Difference Diagram)
A Clock Difference Diagram (CDD) is a directed acyclic graph consisting of a set of nodes N, such that:

- *N Has exactly two terminal nodes called* `True` *and* `False`*.*
- *All other nodes $n \in N$ are inner nodes written as $((i,j),(Int_1,n_1),\ldots,(Int_q,n_q))$ where $(Int_1,n_1),\ldots,(Int_q,n_q)$ are the successor nodes of n and $n_1,\ldots,n_q \in N$.*

Let $n \xrightarrow{Int} m$ indicate that $(Int,m) \in \{(Int_1,n_1),\ldots,(Int_q,n_q)\}$. For each inner node n, the following must hold:

- *The successors are disjoint: for all $l,m \in \{1,\ldots,q\}$, $l \neq m$ it follows that $Int_l \cap Int_m = \emptyset$.*
- *All successors forms an $\mathbb{R}$-cover: $\bigcup_{m\in\{1,\ldots,q\}} Int_m = \mathbb{R}$.*
- *The CDD is ordered: for all $n \xrightarrow{Int} m$ then $(i,j)_m \sqsubseteq (i,j)_n$.*

Further, the CDD is assumed to be reduced:

- *It has maximal sharing: for all $n,m \in N$, $\{(Int_1,n_1),\ldots,(Int_q,n_q)\} = \{(Int_1,m_1),\ldots,(Int_q,m_q)\}$ implies that $n = m$.*
- *It has no trivial edges: whenever $n \xrightarrow{Int} m$ then $Int \neq \mathbb{R}$.*
- *All intervals are maximal: whenever $n \xrightarrow{Int_1} m$, $n \xrightarrow{Int_2} m$ then $Int_1 = Int_2$ or $Int_1 \cup Int_2$ does not equal any other interval in the graph.*

Let V be the set of clock valuations. The semantics of a CDD is defined as:

Definition 5.1.2
Given a CDD, each node $n \in N$ is assigned a semantics $[\![n]\!] \subseteq V$, recursively defined by:

- $[\![\texttt{True}]\!] = V$,
- $[\![\texttt{False}]\!] = \emptyset$,
- $[\![n]\!] = [\![((i,j),(Int_1,n_1),\ldots,(Int_q,n_q))]\!] = \bigcup_{m\in\{1,\ldots,q\}}\{v \in [\![n_m]\!] | Int_m(i,j)(v) = \texttt{True}\}$.

Figure 5.1 on the following page shows how different Federations can be represented by CDD's. Note that only intervals that leads to true are shown. Every operation needed in a federation is supported by the CDD data-structure such as: union, intersection and letting time pass. [LWYP98] discusses how these

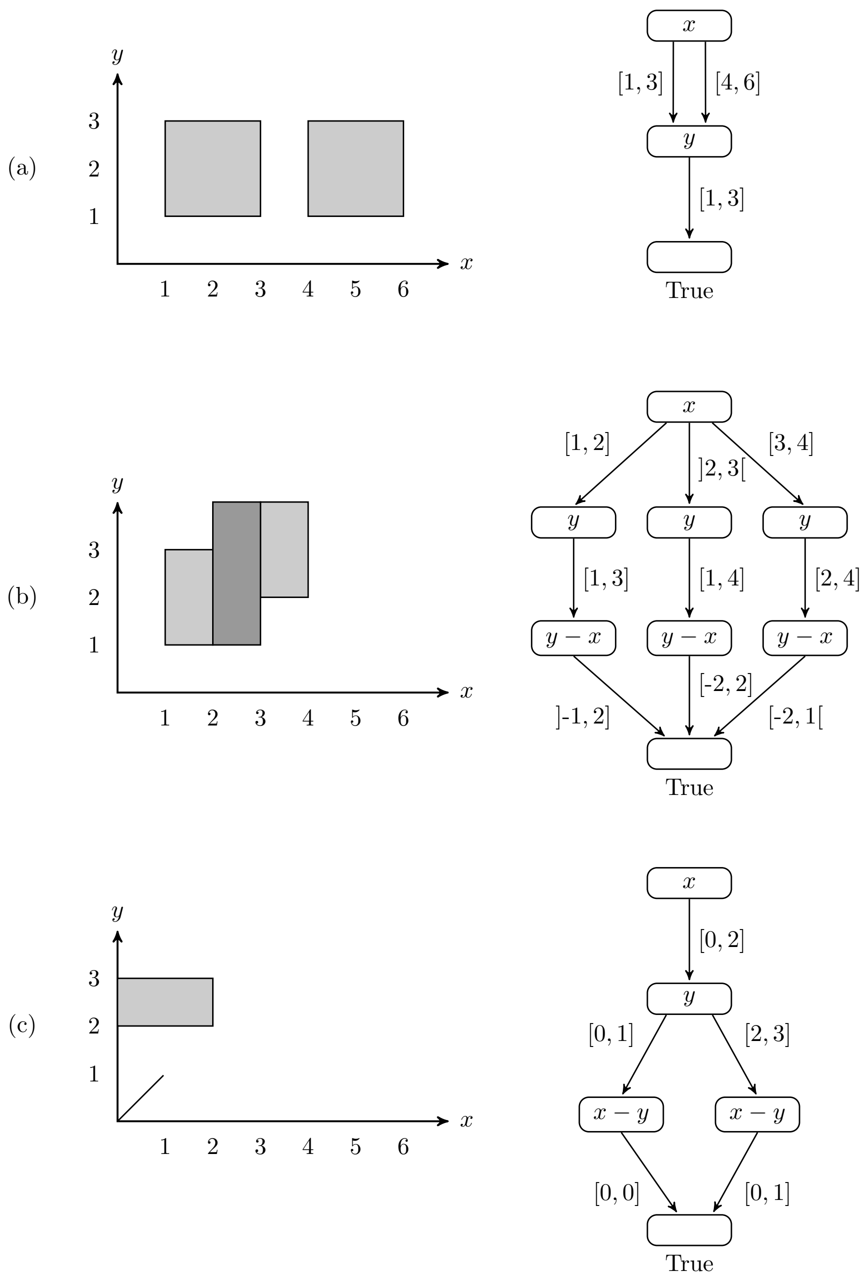


Figure 5.1: Example CDD’s. Intervals not shown lead implicitly to false.

and other operations are accomplished. These operations make representing federations with CDD's a viable option for state-space exploration [BLP+99].

5.2 UPPAAL TIGA hybrid-CDD strategies

TIGA has a feature that allows it to output the strategies as a hybrid-CDD. Hybrid-CDDs consist of a CDD with binary desicion diagrams(BDDs) as subtrees, and actions instead of true and false. The CDD is responsible for evaluating the clocks in the system, and the BDDs evaluate locations and variables of the system. For example, suppose that we want to execute the action `remove` if the clock `x` is between 108 and 130, the brick is in location `sensed` and `colour = black`. The subtree responsible for this querry can be seen in Figure 5.2

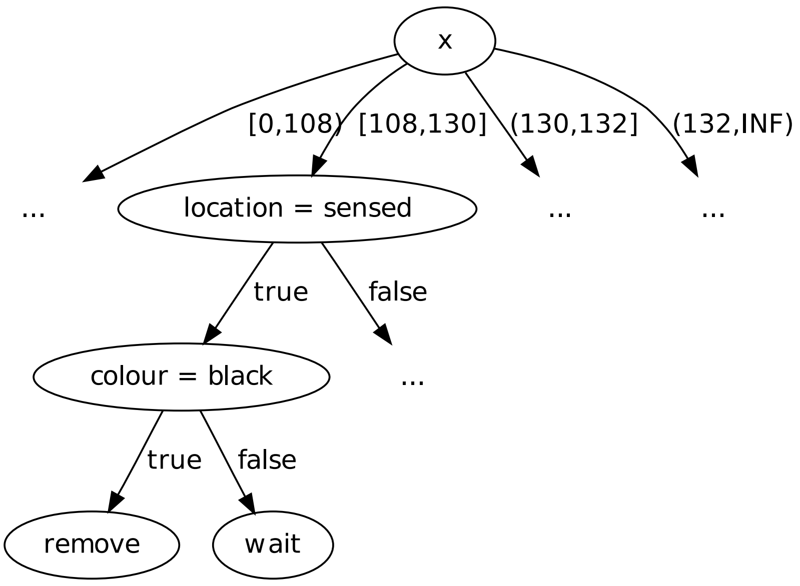


Figure 5.2: A hybrid-CDD subtree.

The BDD part of the tree asks if the current value of a variable or location mathes a given value according to a Boolean expression. If the current value matches the given value the true edge is taken if it does not, the false edge is taken. By traversing the tree with current clock, location and variable values a controller can deside to do an action or wait.

Compare the hybrid-CDD output in Figure 5.3 on the following page, with the default TIGA strategy output bellow:

```
State: ( Controller.CONTR Brick(0).SENSED ) Brick(0).colour=0
While you are in (Brick(0).x<=132), wait.
```

```
State: ( Controller.CONTR Brick(0).OFF ) Brick(0).colour=1
While you are in true, wait.

State: ( Controller.CONTR Brick(0).SENSED ) Brick(0).colour=1
While you are in (Brick(0).x<108), wait.
When you are in (108<=Brick(0).x && Brick(0).x<=130),
take transition Controller.CONTR->Controller.CONTR
{ 1, remove!, 1 } Brick(0).SENSED->Brick(0).OFF
{ x >= TOTAL_TIME_MIN && x <= TOTAL_TIME_MAX, remove?, 1 }

State: ( Controller.CONTR Brick(0).READY ) Brick(0).colour=0
While you are in true, wait.

State: ( Controller.CONTR Brick(0).END ) Brick(0).colour=0
While you are in true, wait.

State: ( Controller.CONTR Brick(0).READY ) Brick(0).colour=1
While you are in true, wait.
```

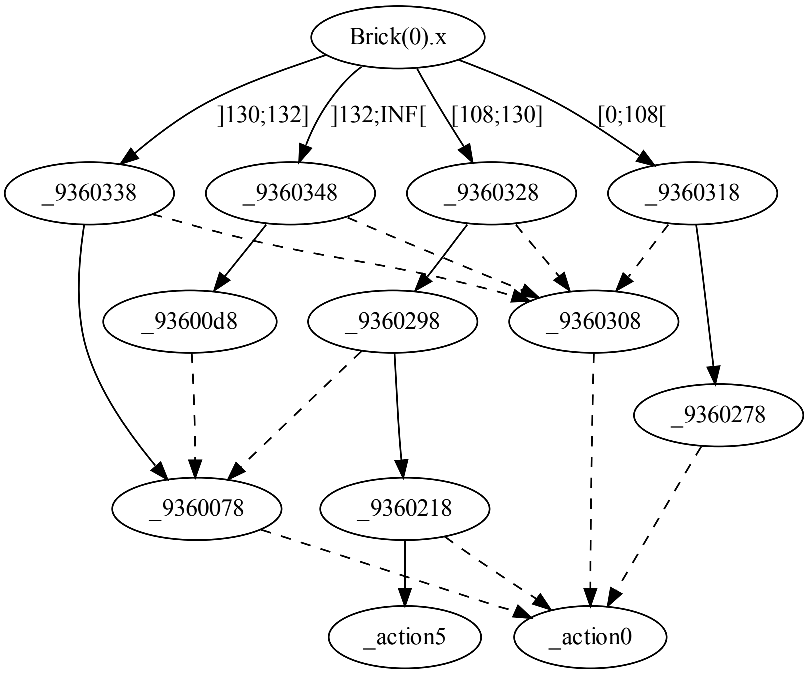


Figure 5.3: A full hybrid-CDD tree as outputted by TIGA. The dotted edges represent false edges, and the solid represents true

The tree in Figure 5.3 on the preceding page uses the followin table to evaluate the locations and variables. `_action0` represents wait and `_action5` represents remove.

```
_9360308: if ((_loc[_Brick(0)] & 2) == 2 && Brick(0).colour == 0)
              goto _error; else goto _action0;
_9360278: if ((_loc[_Brick(0)] & 2) == 0 && Brick(0).colour == 1)
              goto _error; else goto _action0;
_9360318: if ((_loc[_Brick(0)] & 1) == 1)
              goto _9360278; else goto _9360308;
_9360078: if (Brick(0).colour == 1)
              goto _error; else goto _action0;
_9360218: if (Brick(0).colour == 1)
              goto _action5; else goto _action0;
_9360298: if ((_loc[_Brick(0)] & 2) == 2)
              goto _9360218; else goto _9360078;
_9360328: if ((_loc[_Brick(0)] & 1) == 1)
              goto _9360298; else goto _9360308;
_9360338: if ((_loc[_Brick(0)] & 1) == 1)
              goto _9360078; else goto _9360308;
_93600d8: if ((_loc[_Brick(0)] & 2) == 2)
              goto _error; else goto _9360078;
_9360348: if ((_loc[_Brick(0)] & 1) == 1)
              goto _93600d8; else goto _9360308;
```

While the standard output is easier for a human to understand, its is far easier for a computer to use a CDD. This is because the CDD contains very simple expressions, which always evaluate to true or false.

5.3 Hybrid-CDD implementation

We would like to be able to execute a given strategy represented as a Hybrid CDD. To accomplish this, we have decided to implement a Hybrid CDD structure and evaluator in Java, in which we can encode a generated strategy.

In order to execute the code generated by TIGA and the parser we have developed a data structure that allow is to do two things. Firstly it allows us to build a hybrid-CDD tree as outputted by TIGA and secondly it supports traversing the tree in order to determine which or if an action should be executed given the values of the clocks in the system, in which locations the system is and the value of the variables.

This is accomplish with four main classes: the `CDD` class, the `CNode` class, the `DNode` and the `ANode` class. The simplest off these is the `ANode` class which represents actions the controller can take. It consists of a processId, two primary methods besides the constructors, and the assorted get and set methods. The primary methods are the `runCode()` method and the `isInInterval(time)` method. Both methods are common for the tree node classes, but only `runCode()` differs in its execution for each class.

`isInInterval(...)` checks if the value of the clock supplied as argument is within the interval belonging to a node. This is accomplished with a call to `isWithinBound(time)` on the interval. If the node has no interval the method just returns true. A node has a interval if it is a child of a `CNode` and either the constructors or the `setInterval(Left bound, min, max, Right bound)` method have been used to initialise the interval.

```
public boolean isInInterval(int clock){
    if(inter == null){
        return true;
    }
    return inter.isWithinBound(clock);
}
```

The `Interval` class implementation can be seen in Appendix A.8.

The runCode method takes no arguments and simply calls the method which corresponds to the action the node represents. In our case there is only a single action representing activating the piston. The method throws a `Wait` exception because `ANode` like all the other node classes(`DNode` and `CNode`) belongs to the interface class `Node` which contains `runCode()`, `setInterval(...)`, and `isInInterval(...)`.

```
public void runCode() throws Wait{
    Main.remove(processID);
    throw new Wait();
}
```

The `DNode` represents conditional if statements on the locations and variables of the system. Each `DNode` consists of a code object, one or two child nodes each corresponding to either true or false and optionally an interval. A code takes four arguments: The type of variable or location the `DNode` checks, which process in question, the value the variable is compared with and the kind of operator used in the comparison. A call to the method `evaluate()` in the code will return true, if the Boolean expression consisting of current value of the variable, the given value in the `DNode` and the operator returns true, and false otherwise. Implementation of the `NCode` class(code) can be seen in Appendix A.6.

`evaluate()` is used in the `runCode()` method. If `evaluate()` returns true `runCode()` will recursively call `runCode()` of its true child node, if it exist, and similarly the false child if `evaluate()` returns false. If the resulting child does not exist `runCode()` will throw a `Wait` exception. Because we have removed paths that lead only to the delay action in the tree an empty child will represent the delay. The main loop, shown in Appendix B will catch the `Wait` and stop executing the tree and delay a little before trying again.

```
public void runCode() throws Wait{
    if (code.evaluate()){
        if(nTrue != null){
            nTrue.runCode();
        }
        else throw new Wait();
    }
    else if(nFalse != null){
        nFalse.runCode();
    }
    else throw new Wait();
}
```

The `CNode` is used to compare the current value of a given clock with a list of intervals. A `CNode` consists, in addition to its own interval, of a clock and a list of `Node` children. `runCode()` starts by reading the time of the clock, then for each of the node in the child list, it checks if `isInInterval(...)` returns true. If that is the case it recursively calls the `runCode()` on the child node. If there are no child nodes or if `isInInterval(...)` returns false for each child a `Wait` exception is thrown.

```
public void runCode() throws Wait{
    int time;
    time = clock.readClock();
    if (sizeOfNodeChildren()>0){
        ListIterator<Node> it = nodeChildren.listIterator();
        while (it.hasNext()){
            Node child = (Node) it.next();
            if (child.isInInterval(time)){
                child.runCode();
            }
        }
        throw new Wait();
    }
    else throw new Wait();
}
```

The last of the four main classes is the `CDD` class and it is responsible for building the hybrid CDD tree. It consist of a root node and two interesting methods. The most simple one is the `runTree()` which simply calls the `runCode()` method on the root node. The other one is overloaded and is used to build the three. The `addExistingNode(...)` method takes two nodes and adds the second node as the first nodes child. An additional Boolean argument is used if the parent is a `DNode`. The whole CDD implementation can be seen in Appendix A

5.4 Parser and code generation

In order to take the strategy generated by TIGA, and put and encode in into our Hybrid-CDD implementation we must parse the output from TIGA, and generate the Strat.java-file.

First the TIGA verifyta is run with:

```
verifytga -s -q -t 0 -g 1 -w 0 -x model.xml
```

which outputs the strategy as pseudo code and as a Hybrid-CDD. An example of the outputted pseudo code, for the Brick_PO_CYC model with one brick process, which pushes off the white bricks, follows.

```
@DOT
-> tiga_dot_long.dot
-> tiga_dot_long.ps
-> tiga_dot_short.dot
-> tiga_dot_short.ps
```

```
@PROCESSES

#define _Controller 0
#define _Brick(0) 1

@ACTIONS

_error: /* Unknown state. */
_action0: /* Delay action. */
_action1: /* Controller.CONTR->Controller.CONTR { 1, remove!, 1 }
Brick(0).SENSED->Brick(0).OFF
    { x >= 5122 && x <= 6133, remove?, 1 }
*/
_action2: /* Brick(0).READY->Brick(0).SENSED
    { 1, white!, x := 0, colour := WHITE }
Controller.CONTR->Controller.CONTR { 1, white?, 1 }
*/
_action3: /* Brick(0).OFF->Brick(0).READY { 1, tau, 1 }
*/
_action4: /* Brick(0).SENSED->Brick(0).END { x > 6133, tau, 1 }
*/
_action5: /* Brick(0).END->Brick(0).READY { 1, tau, 1 }
*/

@CODE

goto _a1b00c8;
_a1502e8: if ((_loc[_Brick(0)] & 2) == 2 && Brick(0).colour == 0)
    goto _error; else goto _action0;
_a150228: if ((_loc[_Brick(0)] & 2) == 0 && Brick(0).colour == 1)
    goto _error; else goto _action0;
_a1502f8: if ((_loc[_Brick(0)] & 1) == 1) goto _a150228; else
    goto _a1502e8;
_a1b00c8 : if (Brick(0).x>=0 && Brick(0).x<5122)
    goto _a1502f8;
_a150078: if (Brick(0).colour == 1) goto _error; else
    goto _action0;
_a1501d8: if (Brick(0).colour == 1) goto _action1; else
    goto _action0;
_a1501e8: if ((_loc[_Brick(0)] & 2) == 2) goto _a1501d8; else
    goto _a150078;
_a150308: if ((_loc[_Brick(0)] & 1) == 1) goto _a1501e8; else
    goto _a1502e8;
_a1b00c8 : if (Brick(0).x>=5122 && Brick(0).x<=6133)
    goto _a150308;
_a150318: if ((_loc[_Brick(0)] & 1) == 1) goto _a150078; else
    goto _a1502e8;
_a1b00c8 : if (Brick(0).x>6133 && Brick(0).x<=6933)
    goto _a150318;
```

```
_a150148: if ((_loc[_Brick(0)] & 2) == 2) goto _error; else
    goto _a150078;
_a150328: if ((_loc[_Brick(0)] & 1) == 1) goto _a150148; else
    goto _a1502e8;
_a1b00c8 : if (Brick(0).x>6933) goto _a150328;
else goto _error;

@END
```

The interesting thing to note in the above output are the label looking like, _a1502e8 between @CODE and @END. These labels contain expressions which, when evaluated, determines whether a process is in a give location or whether a variable has a give value. These labels decorate the nodes in the CDD outputted by the verifier after running the above verifyta command. The CDD belonging to the above pseudo code can be seen in Figure 5.4.

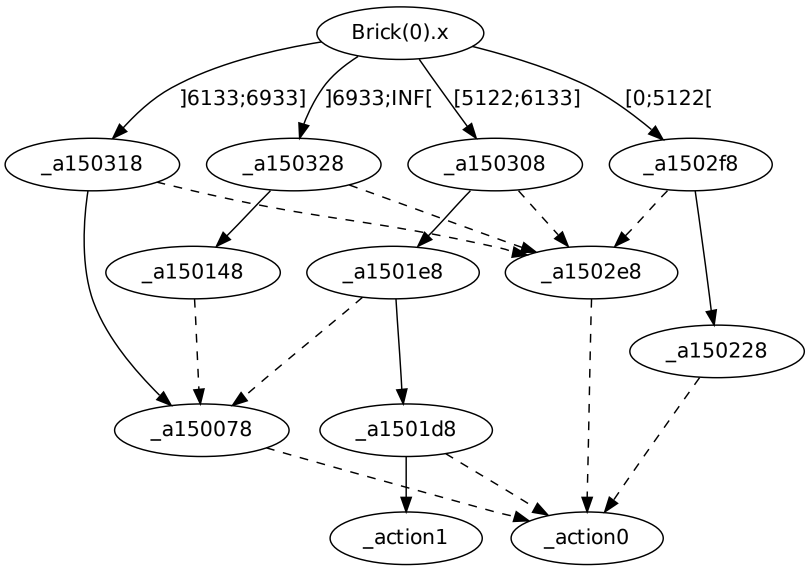


Figure 5.4: CDD for Brick_PO_CYC

The pseudo code and the CDD are run through the following steps to produce runnable for a LEGO NXT. First all the paths leading only to the node labelled _action0 are removed, this node represents the delay-action. This is a simple optimisation, which removes a lot of useless nodes from the CDD. The next step involves reading in the pseudo code corresponding to a given node in the CDD. This pseudo code is then parsed. This process yields a directed acyclic graph, which is decorated with the parsed pseudo code. This graph is then used to generate a JAVA-class which represents the strategy in the form of a Hybrid-CDD. An example of the code generated from the model Brick_PO_CYC, with one brick process and a property which pushes of all the white bricks can be seen in Appendix A.12.

This JAVA-class is then compiled, linked with the needed code, and uploaded to the LEGO NXT. The implementation of a parser/code generator, which supports a subset of the output from TIGA needed to support the Brick_PO_CYC model, can be found in Appendix C.

In this chapter we have presented a way to implement Hybrid CDDs on a LEGO NXT in Java. Even though it does not implement all the functionality to support all the features needed to evaluate every UPPAAL expression, for example there is no support for UPPAAL-C, it still has enough functionality to be used for the Brick_PO_CYC model strategies.

We also presented a small parser/code-generator, which can parse the outputted strategy from TIGA, and generate the Java-code needed to initialise the corresponding Hybrid CDD.

This enables us to take the Brick_PO_CYC model in TIGA, and generate strategies with a changing number brick processes or a different property, and within seconds have the real life brick-sorter working in the intended way.

Chapter 6

Conclusion

Through this report we have gone from defining discrete event systems and modelling formalisms, to using models and modelling tools to automatically generate code for a real life brick-sorter, which we have build with LEGO bricks. The brick-sorter will, depending on the colour of the bricks passing the sensor, either push bricks off the belt or let them travel to the end.

We have defined timed automata in Chapter 2, which is a formalism perfectly suited to model discrete event systems, such as our brick-sorter. The reason timed automata is suited for modelling DESs, is that they can model an event with a transition from one state to another. Each transition can be constrained by guards that only allow the transition to be taken if the values of the clocks satisfied the guards and the invariants, which are constraints on the locations. These constraints allow us to model that events can happen during certain time intervals, and this perfectly reflects real-life systems, which often is dependent on timing. Timed automata use regions and zones to represent infinitely many states in a finite way, which enables full states-pace exploration of the model. While the results presented in the chapter certainly is not something new, they are important to understand in order to get the whole picture from model to implementation.

Chapter 3 dealt with playing a two player game on a timed game automata. The game is used to synthesise a strategy that guarantee certain objectives. The game is played between a controller and an environment. The controller can do two things, it can either decide, at certain states, to take a special transition marked as controllable, if they are enabled, or let time pass. The environment can chose to take transitions marked as uncontrollable or wait at certain states. The transitions taken by the environment take precedence over the ones suggested by the controller. This reflects that unpredictable events do not wait for a process to finish before they happen. The objective of the game can be one of two tings. For a reachability objective, the goal for the controller is to end in a certain set of states, and for the environment to prevent the system from reaching theses states. For a safety objective, the controllers goal is to stay within a set of states, and the goal for the environment is to force the system out of these states.

Given all the winning states found during a state space exploration, we have shown it is possible to generate a strategy from the above rules, and have described how the model checker tool UPPAAL TIGA accomplishes this. The idea is that the synthesised strategies is used as an important part of the implementation of the system, namely which methods, represented as controllable transitions in the model, to call at certain times. As such it is important to make sure that the strategies are as short as possible. To accomplish this we have added a pruning algorithm that in certain situations, at the cost of time used to generate the strategy, will result in smaller strategies.

In Chapter4 we have further explored reducing the size of the strategies by altering the model in a way that guarantee the resulting strategy is still applyable on the brick-sorter. We have developed an alternating simulation, which allow us to abstract multiple locations in the model into a single location. Performed experiments show that the abstract model results in a much smaller explored states-pace and synthesised strategy. The experiments also show that making the model acyclic results in smaller strategies. Making the model sequential reduces the size of the explored states-pace and combining acyclic and sequential models both reduce states-pace and strategy size. Experiments with discrete models show that they can yield the best results with a low enough resolution on the discrete clocks, but upping the resolution quickly results in large states-paces and strategies making utilisation of the model questionable.

The implementation of the abstracted model is described in Chapter 5. UPPAAL TIGA is able to output strategies in a hybrid-CDD representation, which allows faster code execution. Therefore we have implemented data-structure that allows building and executing Hybrid-CDDs in Java. To complement this we have developed a parser that takes the Hybrid-CDD strategy outputted by TIGA and generates code initialise the implemented data structure. The result is that we are able to change which colour of bricks we want to sort off in TIGA and automatically transfer the new program to LEGO NXT. Further developing this technology can result in amazing flexibility of discrete event systems modelled with timed automata.

Chapter 7

Future work

The results and ideas we have introduced in this report is only the begining. Even though we have only worked with a very small toy example, we have hit the roof, in terms of what model checking of real-time systems can handel. It is therefore clear that the ideas from this report can only be put into wide use, if some fundemental problems in model checking are solved. However there is still hope that this will happen, as it is a very active research area, which makes progress every year.

It is also clear from working on this project that chosing the right way to model a system is very important. Even small changes to a model can significantly lower both the memory and time needed to produce a usable strategy. A way to handle this is ofcourse to just rely on the model checkers to become better and better, but since the techniques employed are `PSPACE-Hard`, it might be a good idea to look into alternatives. A viable alternative is, as mentioned earlier, to apply various modelling tricks to improve the resources needed to check a model. However as these tricks require a great deal of knowledge about how the model checker works internally, it would be a good idea to have the model checker apply these tricks automaticaly. This would allow the model checker to handel more complex models, and choose the modelling tricks needed to reduce the verification time, state space usage and the strategy size. Finding ways to reduce the size of the strategy as well as minimising the evaluation time of these is also important. If the strategies are to be implemented on embeded systems, there will be limited resources, in terms of memory, for holding the strategy, and in terms of cpu resoucese, for executing the strategy. Minimising the time it takes to execute the code is extremely important on hard real-time systems, as the system must be able to act on continues input, with out missing important data. For example if a controller for a car airbag takes 100ms to execute the strategy, but it must be able to react wihtin 10ms to save lives.

Another way to make the strategy generation easier, might be to use the fact that large parts of a systems might be modelled discretely. This uses the fact that if the strategy is implemented as hardware or software, then its actions can only occur in discrete intervals. So if a part of the model's only job is to issue and act on discrete events, og change the discrete state of the model, then this part might be seen as a discrete model. There are a lot of very of very efficient techniques for verifying discrete models, and some of these migth be applied to the discrete parts of a model. These techniques include, SAT-solving and partial order reduction.

It seems fair to assume that modelling real-time systems with timed automata and expressing wanted properties in CTL requires specialist knowledge, which normal people does not have. So to make what we have presented in this report accessible to more people, it would be a good idea to put some further abtractions ontop of timed automata and CTL. These abstractions should be target at an end user. For example at software engineer might be more comfortable with UML, a production engineer with flow diagrams or a factory worker might be most comfortabel to tell a system what to do in natural language.

So far all the tings mentioned in this chapter are things which are not specific just to our setup, but general problems which also affects other areas outside our field.

However there are also things which could be improved upon in our project. The Hybrid-CDD implementation we have written, is not general enough to handle every kind of expression UPPAAL TIGA can output. Our implementation can only handle a single action, activating a piston, for the implementaion to be truely usefull, it should be possible to specify what code an action should run, directly in the model. Also the output from TIGA as it is now is not easy to parse, and sometimes the outputtet CDDs are broken. This was not a big problem for us, as our parser is very basic. But if code generation from TIGA is to be used for real work, then a complete parser should be implemented. This could very well be within TIGA itself, which would allow access directly to all the information about a strategy, without going through a text format first.

If all of this comes together, then it would be possible to make a tool, which will enable an engineer to lay out a factory floor with different production machines and define how they are connected. This tool will generate a model which takes the functionality of each machine and represent the functionallity of the entire factory. The next thing would be to define the products the factory are to produce as processes. Then whenever the factory gets an order, the sales people can use these product processes to see if the factory have capacity to produce the desired product numbers, simply be checking if a strategy exits for the wanted order. If the strategy exists then it will simply be a matter of having the factory implement it. As an added bonus it might be possible to optimize the strategy with regards to things like production time, reosource use or overall cost.

Appendix A

hybrid-CDD implementation

These are all the classes used in the hybrid-CDD implementation

A.1 CDD.java

```
public class CDD{
    private Node root;
    static Node node;

    protected void setRoot(Node n){
        root = n;
    }

    public CDD(){
        setRoot(null);
    }

    public CDD(Node n){
        setRoot(n);
    }

    public Node getRoot(){
        return root;
    }

    public void addExistingNode(Node o, Node c){
        CNode p = (CNode)o;
        if ((p != null) && (c != null)){
            p.addChild(c);
        }
    }
    public void addExistingNode(Node o, Node c,
                                boolean type){
        DNode p = (DNode)o;

        if ((p != null) && (c != null)){
            try {
                if(type == false){
                    node = p.getFalse();
                }
                else node = p.getTrue();
            } catch (NoNodeException e){

                if (type == false){
                    p.setFalse(c);
                }
                else p.setTrue(c);
            }
        }
    }

    public void runTree() throws Wait{
        try{
            root.runCode();
        } catch (Wait e){
            throw new Wait();
        }
    }

    public void testTree(){
        root.test();
    }
}
```

A.2 Node.java

```
interface Node{

    public void runCode() throws Wait;
    public void setInterval(Bound left, float min,
                            float max, Bound right);
    public void test();
    public boolean isInInterval(int clock);

}
```

A.3 CNode.java

```
import java.util.*;
```

```
public class CNode implements Node{

    protected ArrayList<Node> nodeChildren
                            = new ArrayList<Node>();

    protected Interval inter;
    protected Clock clock;

    public CNode(){
        inter = null;
        clock = null;
    }

    public CNode(int cl){
        inter = null;
        clock = new Clock(cl);
    }

    public CNode(int cl, float min, Bound left){

        inter = new Interval(min,left);
        clock = new Clock(cl);
    }

    public CNode(int cl, float min, float max,
                 Bound left, Bound right){

        inter = new Interval(min, max, left, right);
        clock = new Clock(cl);
    }

    public void setInterval(Bound left ,float min,
                            float max, Bound right){
        if (inter == null)
            inter = new Interval(min, max, left, right);
        else {
            inter.setInterval(min,max);
            inter.setBounds(left, right);
        }
    }

    public float getIntervalMin(){
        return inter.getMinInterval();
    }

    public float getIntervalMax(){
        return inter.getMaxInterval();
    }
    public void addChild(Node child){
        nodeChildren.add(child);
    }

    public Node getChild(int i) throws NoNodeException{
        if (i > nodeChildren.size()){
            throw new NoNodeException();
        }
        else return (Node)nodeChildren.get(i);
    }

    public int sizeOfNodeChildren(){
        return nodeChildren.size();
    }

    public boolean isInInterval(int clock){
        if (inter == null){
            return true;
        }
        return inter.isWithinBound(clock);
    }

    public void runCode() throws Wait{
        int time;
        time = clock.readClock();
        if (sizeOfNodeChildren()>0){
            ListIterator<Node> it
                        = nodeChildren.listIterator();
            while (it.hasNext()){
                Node child = (Node) it.next();
                if (child.isInInterval(time)){
                    child.runCode();
                }
            }
            throw new Wait();
        }
        else throw new Wait();
    }

    public void test(){
        System.out.println("This is a CNode");
    }

}
```

A.4 DNode.java

```
public class DNode implements Node{

    protected NCode code;
    protected Node nTrue, nFalse;

    protected Interval inter;

    public DNode(){
        code=null;
        nTrue=null;
        nFalse=null;
        inter=null;
    }

    public DNode(NCode d){
        code=d;
        nTrue=null;
        nFalse=null;
        inter=null;
    }

    public DNode(NCode d, float min, Bound left){
        code=d;
        nTrue=null;
        nFalse=null;
        inter=new Interval(min, left);
    }

    public DNode(NCode d, float min, float max,
                  Bound left, Bound right){
        code=d;
        nTrue = null;
        nFalse=null;
        inter=new Interval(min,max,left,right);
    }

    public DNode(int varC, int var, int val, int op){
        code = new NCode(varC, var, val, op);
        nTrue= null;
        nFalse= null;
        inter = null;
    }

    public void setInterval(Bound left, float min,
                            float max, Bound right){
        if (inter == null)
            inter = new Interval(min, max, left, right);
        else {
            inter.setInterval(min,max);
            inter.setBounds(left, right);
        }
    }
```

```
public boolean isInInterval(int clock){
    if (inter == null){
        return true;
    }
    return inter.isWithinBound(clock);
}

public void setTrue(Node node){
    nTrue=node;
}

public void setFalse(Node node){
    nFalse=node;
}

public void setCode(NCode c){
    code=c;
}

public Node getTrue() throws NoNodeException{
    if (nTrue == null){
        throw new NoNodeException();
    }
    else return nTrue;
}

public Node getFalse() throws NoNodeException{
    if (nFalse == null){
        throw new NoNodeException();
    }
    else return nFalse;
}

public NCode getCode(){
    return code;
}

public void runCode() throws Wait{
    if (code.evaluate()){
        if(nTrue != null){
            nTrue.runCode();
        }
        else throw new Wait();
    }
    else if(nFalse != null){
        nFalse.runCode();
    }
    else throw new Wait();
}
```

```
    public void test(){
        System.out.println("This␣is␣a␣DNode");
    }

}
```

A.5 ANode.java

```
public class ANode implements Node{

    protected Interval inter;
    protected int processId;

    public ANode(int id){
        processId = id;
        inter=null;
    }

    public ANode(int id, float min, Bound left){
        processId = id;
        inter=new Interval(min,left);
    }

    public ANode(int id, float min, float max,
                            Bound left, Bound right){
        processId = id;
        inter = new Interval(min, max, left, right);
    }

    public void setInterval(Bound left, float min,
                            float max, Bound right){
        if (inter == null)
            inter = new Interval(min, max, left, right);
        else {
            inter.setInterval(min, max);
            inter.setBounds(left,right);
        }
    }

    public void runCode() throws Wait{
        Main.remove(processId);
        throw new  Wait();
    }

    public boolean isInInterval(int clock){
        if(inter == null){
            return true;
        }
        return inter.isWithinBound(clock);
    }
    public void test(){
        System.out.println("This is a ANode");
    }

}
```

A.6 NCode.java

```
public class NCode{

    protected int variable;

    protected int variableChooser;

    protected int value;

    protected int operator;

    public NCode(){
        variableChooser = 0;
        variable=0;
        value=0;
        operator=0;

    }

    public NCode(int varC, int var, int val, int op){
        variableChooser = varC;
        variable = var;
        value = val;
        operator = op;
    }

    public boolean evaluate(){
        switch (variableChooser){

        case 0:

            switch (operator){

            case 0:
                if ((Strat.location[variable]
                                                & value )>0){
                    return true;
                }
                else return false;

            case 1:
                if (Strat.location[variable] == value){
```

```
                return true;
            }
            else return false;

        default:
            System.out.println("Not␣valid␣code");
        }
        break;
    case 1:
        switch (operator) {
            case 0:
            if (Strat.colour[variable] == value){
                return true;
            }
            else return false;

        case 1:
            if (Strat.colour[variable] <= value){
                return true;
            }
            else return false;

        case 2:
            if (Strat.colour[variable] >= value){
                return true;
            }
            else return false;

        case 3:
            if (Strat.colour[variable] < value){
                return true;
            }
            else return false;

        case 4:
            if (Strat.colour[variable] > value){
                return true;
            }
            else return false;

        default:
            System.out.println("Not␣a␣valid
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣operator␣value");
        }
        break;
    default:
        System.out.println("Not␣a␣valid
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣variable␣type␣value");
    }
    return false;
    }
}
```

A.7 Clock.java

```
public class Clock{

    protected int name;

    public Clock(){
        name = 0;
    }

    public Clock(int n){
        name = n;
    }

    public int readClock(){
        return Main.value(name);
    }

    public void resetClock(){
        Main.reset(name);
    }

    public int getName(){
        return name;
    }

    public void setName(int n){
        name = n;
    }

}
```

A.8 Interval.java

```
public class Interval{

    protected Bound lBound;
    protected Bound rBound;

    protected Float intMin;
    protected Float intMax;

    public Interval(){
        lBound = Bound.closed;
        rBound = Bound.open;
        intMax = new Float(Float.POSITIVE_INFINITY);
        intMin = new Float(0);
    }

    public Interval(float min, Bound left){
        lBound = left;
        rBound = Bound.open;
        intMax = new Float(Float.POSITIVE_INFINITY);
        intMin = new Float(min);
    }

    public Interval(float min, float max, Bound left,
                                          Bound right){
        lBound = left;
        rBound = right;
        intMin = new Float(min);
        intMax = new Float(max);
    }

    public void setInterval(float min, float max){
        intMin = new Float(min);
        intMax = new Float(max);
    }


    public void setBounds(Bound left, Bound right){
        rBound = right;
        lBound = left;
    }

    public float getMinInterval(){
        return intMin.floatValue();
    }

    public float getMaxInterval(){
        return intMax.floatValue();
    }

    public boolean isWithinBound(int clock){

        if (lBound == Bound.closed){
            if (rBound == Bound.closed){
                if (clock >= intMin && clock <= intMax){
                    return true;
                }
                else return false;
            }
            else {
                if (clock >= intMin && clock < intMax){
                    return true;
                }
                else return false;
            }
        }
        else {
            if (rBound == Bound.closed){
                if (clock > intMin && clock <= intMax){
                    return true;
                }
                else return false;
            }
            else {
                if (clock > intMin && clock < intMax){
                    return true;
                }
                else return false;
            }
        }
    }
}
```

A.9 Wait.java

```
class Wait extends Exception{
    Wait(){
        super("Wait");
    }
}
```

A.10 NoNodeException.java

```
class NoNodeException extends Exception{
    NoNodeException(){
        super("No such node");
    }
}
```

A.11 Bound.java

```
enum Bound {closed, open}
```

A.12 Strat.java

Here is an example of a generated strategy, which handles one brick, and white bricks are pushed of the convyour belt.

```
import java.util.Hashtable;

class Strat {
public static CDD tree;
public static void makeStrategy() {
    Hashtable vertices = new Hashtable();

    vertices.put("9dd2ef8", new DNode(0, 1, 2, 0));
    vertices.put("9dd0738", new DNode(1, 1, 1, 0));
    vertices.put("9dd2728", new DNode(0, 1, 2, 0));
    vertices.put("9dd3288", new DNode(0, 1, 2, 0));
    vertices.put("9dd1dd8", new DNode(1, 0, 1, 0));
    vertices.put("9dd3b98", new DNode(0, 0, 1, 0));
    vertices.put("9dd2358", new DNode(1, 0, 1, 0));
    vertices.put("9dd3b78", new DNode(0, 0, 1, 0));
    vertices.put("9e51748", new CNode(1));
    vertices.put("9dd3a38", new DNode(0, 0, 1, 0));
    vertices.put("9dd31e8", new DNode(1, 1, 1, 0));
    vertices.put("9dd3668", new DNode(0, 1, 1, 0));
    vertices.put("9dd3208", new DNode(0, 1, 2, 0));
    vertices.put("9e516e8", new CNode(1));
    vertices.put("9dd3b18", new DNode(0, 0, 1, 0));
    vertices.put("9dd1658", new DNode(1, 1, 1, 0));
    vertices.put("9dd3988", new DNode(0, 0, 2, 0));
    vertices.put("9dd3bb8", new DNode(0, 0, 2, 0));
    vertices.put("9d40068", new ANode(1));
    vertices.put("9dd1858", new DNode(1, 0, 1, 0));
    vertices.put("9dd0748", new DNode(1, 0, 1, 0));
    vertices.put("9dd3608", new DNode(0, 1, 1, 0));
    vertices.put("9e517a8", new CNode(0));
    vertices.put("9dd17f8", new DNode(1, 1, 1, 0));
    vertices.put("9dd36a8", new DNode(0, 0, 2, 0));
    vertices.put("9e51778", new CNode(1));
    vertices.put("9dd3b88", new DNode(0, 0, 2, 0));
    vertices.put("9dd3698", new DNode(0, 1, 1, 0));
    vertices.put("9dd3b68", new DNode(0, 0, 2, 0));
    vertices.put("9c249e8", new ANode(0));
    vertices.put("9dd31f8", new DNode(1, 0, 1, 0));
    vertices.put("9dd3a28", new DNode(0, 0, 1, 0));
    vertices.put("9dd3678", new DNode(0, 0, 2, 0));
    vertices.put("9e51718", new CNode(1));
    vertices.put("9dd0ff8", new DNode(0, 1, 2, 0));
    vertices.put("9dd2b38", new DNode(0, 1, 1, 0));
    vertices.put("9dd2e78", new DNode(0, 1, 2, 0));
    vertices.put("9dd3b08", new DNode(0, 0, 2, 0));
    vertices.put("9dd23e8", new DNode(1, 0, 1, 0));
    vertices.put("9dd3bc8", new DNode(0, 0, 1, 0));
    vertices.put("9dd3638", new DNode(0, 1, 1, 0));
    vertices.put("9dd35f8", new DNode(0, 1, 2, 0));
    vertices.put("9dd3ba8", new DNode(0, 0, 1, 0));
    vertices.put("9dd36f8", new DNode(0, 1, 1, 0));
    vertices.put("9dd3978", new DNode(0, 1, 1, 0));

    tree = new CDD( (Node) vertices.get("9e517a8"));
```

```
tree.addExistingNode((Node)
vertices.get("9e517a8"),
(Node) vertices.get("9e51778"));
((Node) vertices.get("9e51778")).setInterval(Bound.
open, 132.0f, Float.POSITIVE_INFINITY, Bound.open);
tree.addExistingNode((Node) vertices.get("9e517a8"),
(Node) vertices.get("9e51748"));
((Node) vertices.get("9e51748")).setInterval(Bound.
open, 130.0f, 132.0f, Bound.closed);
tree.addExistingNode((Node) vertices.get("9e517a8"),
(Node) vertices.get("9e51718"));
((Node) vertices.get("9e51718")).setInterval(
Bound.closed, 108.0f, 130.0f, Bound.closed);
tree.addExistingNode((Node) vertices.get("9e517a8"),
(Node) vertices.get("9e516e8"));
((Node) vertices.get("9e516e8")).setInterval(
Bound.closed, 0.0f, 108.0f, Bound.open);
tree.addExistingNode((Node) vertices.get("9e51778"),
(Node) vertices.get("9dd3bc8"));
((Node) vertices.get("9dd3bc8")).setInterval(
Bound.closed, 108.0f, 130.0f, Bound.closed);
tree.addExistingNode((Node) vertices.get("9e51748"),
(Node) vertices.get("9dd3ba8"));
((Node) vertices.get("9dd3ba8")).setInterval(
Bound.closed, 108.0f, 130.0f, Bound.closed);
tree.addExistingNode((Node) vertices.get("9e51718"),
(Node) vertices.get("9dd3a38"));
((Node) vertices.get("9dd3a38")).setInterval(
Bound.open, 132.0f, Float.POSITIVE_INFINITY,
Bound.open);
tree.addExistingNode((Node) vertices.get("9e51718"),
(Node) vertices.get("9dd3a28"));
((Node) vertices.get("9dd3a28")).setInterval(
Bound.open, 130.0f, 132.0f, Bound.closed);
tree.addExistingNode((Node) vertices.get("9e51718"),
(Node) vertices.get("9dd3b18"));
((Node) vertices.get("9dd3b18")).setInterval(
Bound.closed, 0.0f, 108.0f, Bound.open);
tree.addExistingNode((Node) vertices.get("9e51718"),
(Node) vertices.get("9dd3b98"));
((Node) vertices.get("9dd3b98")).setInterval(
Bound.closed, 108.0f, 130.0f, Bound.closed);
tree.addExistingNode((Node) vertices.get("9e516e8"),
(Node) vertices.get("9dd3b78"));
((Node) vertices.get("9dd3b78")).setInterval(
Bound.closed, 108.0f, 130.0f, Bound.closed);
tree.addExistingNode((Node) vertices.get("9dd3bc8"),
(Node) vertices.get("9dd3988"), false);
tree.addExistingNode((Node) vertices.get("9dd3bc8"),
(Node) vertices.get("9dd3bb8"), true);
```

```
tree.addExistingNode((Node) vertices.get("9dd3ba8"),
(Node) vertices.get("9dd3988"), false);
tree.addExistingNode((Node) vertices.get("9dd3ba8"),
(Node) vertices.get("9dd36f8"), true);
tree.addExistingNode((Node) vertices.get("9dd3a38"),
(Node) vertices.get("9dd36a8"), true);
tree.addExistingNode((Node) vertices.get("9dd3a28"),
(Node) vertices.get("9dd3678"), true);
tree.addExistingNode((Node) vertices.get("9dd3b18"),
(Node) vertices.get("9dd3b08"), true);
tree.addExistingNode((Node) vertices.get("9dd3b98"),
(Node) vertices.get("9dd3988"), false);
tree.addExistingNode((Node) vertices.get("9dd3b98"),
(Node) vertices.get("9dd3b88"), true);
tree.addExistingNode((Node) vertices.get("9dd3b78"),
(Node) vertices.get("9dd3988"), false);
tree.addExistingNode((Node) vertices.get("9dd3b78"),
(Node) vertices.get("9dd3b68"), true);
tree.addExistingNode((Node) vertices.get("9dd3988"),
(Node) vertices.get("9dd3978"), true);
tree.addExistingNode((Node) vertices.get("9dd3988"),
(Node) vertices.get("9dd2b38"), false);
tree.addExistingNode((Node) vertices.get("9dd3bb8"),
(Node) vertices.get("9dd36f8"), false);
tree.addExistingNode((Node) vertices.get("9dd36f8"),
(Node) vertices.get("9dd2ef8"), true);
tree.addExistingNode((Node) vertices.get("9dd36a8"),
(Node) vertices.get("9dd3698"), true);
tree.addExistingNode((Node) vertices.get("9dd3678"),
(Node) vertices.get("9dd3668"), true);
tree.addExistingNode((Node) vertices.get("9dd3b08"),
(Node) vertices.get("9dd3608"), true);
tree.addExistingNode((Node) vertices.get("9dd3b88"),
(Node) vertices.get("9dd3638"), true);
tree.addExistingNode((Node) vertices.get("9dd3b88"),
(Node) vertices.get("9dd36f8"), false);
tree.addExistingNode((Node) vertices.get("9dd3b68"),
(Node) vertices.get("9dd2b38"), true);
tree.addExistingNode((Node) vertices.get("9dd3b68"),
(Node) vertices.get("9dd36f8"), false);
tree.addExistingNode((Node) vertices.get("9dd3978"),
(Node) vertices.get("9dd0ff8"), true);
tree.addExistingNode((Node) vertices.get("9dd2b38"),
(Node) vertices.get("9dd2728"), true);
tree.addExistingNode((Node) vertices.get("9dd2ef8"),
(Node) vertices.get("9dd23e8"), true);
tree.addExistingNode((Node) vertices.get("9dd3698"),
(Node) vertices.get("9dd2e78"), true);
tree.addExistingNode((Node) vertices.get("9dd3698"),
(Node) vertices.get("9dd35f8"), false);
```

```
tree.addExistingNode((Node) vertices.get("9dd3668"),
(Node) vertices.get("9dd35f8"), false);
tree.addExistingNode((Node) vertices.get("9dd3668"),
(Node) vertices.get("9dd2358"), true);
tree.addExistingNode((Node) vertices.get("9dd3608"),
(Node) vertices.get("9dd35f8"), false);
tree.addExistingNode((Node) vertices.get("9dd3608"),
(Node) vertices.get("9dd3288"), true);
tree.addExistingNode((Node) vertices.get("9dd3638"),
(Node) vertices.get("9dd3208"), true);
tree.addExistingNode((Node) vertices.get("9dd3638"),
(Node) vertices.get("9dd35f8"), false);
tree.addExistingNode((Node) vertices.get("9dd0ff8"),
(Node) vertices.get("9dd0748"), true);
tree.addExistingNode((Node) vertices.get("9dd2728"),
(Node) vertices.get("9dd0738"), true);
tree.addExistingNode((Node) vertices.get("9dd23e8"),
(Node) vertices.get("9dd0738"), false);
tree.addExistingNode((Node) vertices.get("9dd2e78"),
(Node) vertices.get("9dd2358"), false);
tree.addExistingNode((Node) vertices.get("9dd35f8"),
(Node) vertices.get("9dd1dd8"), false);
tree.addExistingNode((Node) vertices.get("9dd35f8"),
(Node) vertices.get("9dd1858"), true);
tree.addExistingNode((Node) vertices.get("9dd2358"),
(Node) vertices.get("9dd1658"), true);
tree.addExistingNode((Node) vertices.get("9dd3288"),
(Node) vertices.get("9dd1dd8"), true);
tree.addExistingNode((Node) vertices.get("9dd3288"),
(Node) vertices.get("9dd2358"), false);
tree.addExistingNode((Node) vertices.get("9dd3208"),
(Node) vertices.get("9dd31f8"), true);
tree.addExistingNode((Node) vertices.get("9dd3208"),
(Node) vertices.get("9dd2358"), false);
tree.addExistingNode((Node) vertices.get("9dd0748"),
(Node) vertices.get("9dd0738"), true);
tree.addExistingNode((Node) vertices.get("9dd0738"),
(Node) vertices.get("9d40068"), true);
tree.addExistingNode((Node) vertices.get("9dd1dd8"),
(Node) vertices.get("9c249e8"), true);
tree.addExistingNode((Node) vertices.get("9dd1858"),
(Node) vertices.get("9dd17f8"), true);
tree.addExistingNode((Node) vertices.get("9dd1658"),
(Node) vertices.get("9c249e8"), false);
tree.addExistingNode((Node) vertices.get("9dd31f8"),
(Node) vertices.get("9dd31e8"), true);
tree.addExistingNode((Node) vertices.get("9dd31f8"),
(Node) vertices.get("9dd0738"), false);
tree.addExistingNode((Node) vertices.get("9dd17f8"),
(Node) vertices.get("9c249e8"), true);
```

```
    tree.addExistingNode((Node) vertices.get("9dd31e8"),
    (Node) vertices.get("9c249e8"), false);
    tree.addExistingNode((Node) vertices.get("9dd31e8"),
    (Node) vertices.get("9d40068"), true);
  }
}
```

Appendix B

Main.java

This is the main class that initiate and controls the sensors, implements the transitions and runs the tree.

```
import lejos.nxt.*;
import lejos.util.*;
import lejos.nxt.addon.*;

public class Main{

    public static DebugMessages dbmsg
                                  = new DebugMessages();

        static TimerListener updateLCD
                                  = new TimerListener() {
            public void timedOut() {
                int i;
                String str;

                LCD.clear();
                for (i = 0; i < Strat.BRICKS; i++) {
                str = "B" + i + "␣" + Strat.location[i]
                              + "␣" + Strat.colour[i]
                              + "␣" + value(i);
                LCD.drawString(str, 0, i);
                }
            }
        };

    static boolean sensor_ready = true;

    static TimerListener sensorToggle
                                   = new TimerListener() {
            public void timedOut() {
            settle.stop();
            sensor_ready = true;
            }
        };

    static Timer settle = new Timer(800, sensorToggle);

    static SensorPortListener sensorL
                            = new SensorPortListener() {
        public void stateChanged(SensorPort aSource,
                            int aOldValue, int aNewValue) {
            int i;

            if (sensor_ready && Math.abs(aNewValue
                                    - aOldValue) < 2) {
                sensor_ready = false;
                settle.start();

                for(i = 0; i < Strat.BRICKS; i++) {
                    if (Strat.location[i] != 3) {

                        if (aNewValue < 750 &&
                                    aNewValue > 700){
                            Strat.location[i] = 3;
                            Strat.colour[i] = 0;
                            reset(i);
                        }
                        if (aNewValue <= 700){
                            Strat.location[i] = 3;
                            Strat.colour[i] = 1;
                            reset(i);
                        }
                        break;
                    }
                }

            }
        }
    };


    protected static void remove(int process){
        Motor.C.rotate(360);
        Strat.location[process] = 2;
    }

    private static void runStrategy(){
        try{

            Strat.tree.runTree();

        } catch (Wait e){
            return;
        }
    }

    public static void main(String[] args)
                                          throws Exception {

        Strat.makeStrategy();

        LightSensor sensor =
                    new LightSensor(SensorPort.S1, true);
        SensorPort.S1.addSensorPortListener(sensorL);

        Timer LCDupdater = new Timer(100, updateLCD);
        LCDupdater.start();

        int i;

        Motor.A.setSpeed(250);
        Motor.B.setSpeed(250);
        Motor.A.forward();
        Motor.B.forward();
        Motor.C.setSpeed(900);

        for (i = 0; i < Strat.BRICKS; i++) {
            reset(i);
            Strat.location[i] = 2;
        }

        while (!Button.ESCAPE.isPressed()) {

            runStrategy();

            for (i = 0; i < Strat.BRICKS; i++) {
                if (Strat.location[i] == 3 &&
                                      value(i) > 6500) {
                    Strat.location[i] = 1;
                    reset(i);
                }
            }
            Thread.sleep(1);
        }


    }

    public static int value(int i) {
        return (int) System.currentTimeMillis()
                                          - Strat.x[i];
    }
    public static void reset(int i) {
        Strat.x[i] = (int) System.currentTimeMillis();
    }

}
```

Appendix C

parser.rb

This is the code for the parser and code generator.

```
require 'rubygems'
require 'rgl/adjacency'
require 'rgl/dot'
require 'rgl/traversal'

$false_edge_list = []
$vertex_label = {}
$edge_label = {}
$actions = {}
$root = ""
$n = 0

def mark_as_false(u, v)
    $false_edge_list.push([u, v])
end

def is_in_false?(u, v)
    return $false_edge_list.include?([u, v])
end

def set_vertex_label(u, label)
    if $vertex_label.has_key?(label)
        if $vertex_label[label].class == String then
            $vertex_label[label] = [$vertex_label[label],
                u]
        else
            $vertex_label[label].push(u)
        end
    else
        $vertex_label[label] = u
    end
end
```

```
def set_edge_label(u, v, label)
    $edge_label[[u, v]] = label
end

def graph_from_dotfile(file)
    g = RGL::DirectedAdjacencyGraph.new

    IO.foreach(file) { |line|
        case line
        #matches a vertex
        when /^\"(.*)\"␣\[label=\"(.*)\"\]\;$/
            g.add_vertex($1)
            set_vertex_label($1, $2)

            v = $1
            action = $2
            if action =~ /\_action[1-9][0-9]*/ then
                $actions[action] = v
            end
        #matches a clock egde
        when /^\"(.*)\"␣\->␣\"(.*)\"␣\[style=(.*),
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣label=\"(.*)\"\]\;$/
            g.add_edge($1, $2)
            set_edge_label($1, $2, $4)
        #matches a discrete edge
        when /^\"(.*)\"␣\->␣\"(.*)\"␣\[style=(.*)\]\;$/
            g.add_edge($1, $2)

            if $3 == "dashed" then
                mark_as_false($1, $2)
            end
        end
    }
    return g
end

def reverse_graph(g)
    f = RGL::DirectedAdjacencyGraph.new

    g.each_edge { |u, v| f.add_edge(v, u) }

    return f
end

def prune_wait_paths(g)
    f = reverse_graph(g)

    visited = {}
    waiting = []
    $actions.each { |action, v|

        f.adjacent_vertices(v).each { |w|
            if not waiting.include?(w) then
                waiting.push(w)
            end
        }

        visited[v] = 1
    }

    until waiting.empty?
        f.adjacent_vertices(waiting[0]).each {  |w|
            if not waiting.include?(w) then
                waiting.push(w)
            end
        }
        visited[waiting[0]] = 1
        waiting.delete_at(0)
    end

    g.each_vertex { |v|
        if not visited.has_key?(v) then
            g.remove_vertex(v)
        end
    }
end

def translate_labels(file)
    tmp = {}
    tmp_actions = {}
    action = ""

    IO.foreach(file) { |line|
        case line
        when /^(\_[0-9a-f]{7})␣?:␣if␣\(?\((.*)\)/
            tmp[$vertex_label[$1]] = $2
        when /^goto␣\_([0-9a-f]{7});$/
            $root = $1
        when /^\#define␣\_Brick/
            $n = $n + 1
        when /^(\_action[0-9]*)/
            action = $1
        when /^Brick\(([0-9])\)\.SENSED.*remove\?/
            tmp_actions[$actions[action]] = $1
        end
    }

    $vertex_label.each { |label, v|
        case label
        when /^_action/ then
            tmp[v] = label
        when /^[^_]/
            if v.class == Array then
                $vertex_label[label].each { |w|
                    tmp[w] = label
                }
            else
                tmp[v] = label
            end
        end
    }
    $vertex_label = tmp
    $actions = tmp_actions
end

def parse_labels()
    $vertex_label.each { |u, label|
        case label
        when /^\_loc\[\_Brick\(([0-9])\)\]␣\&␣([0-9])\)\
␣␣␣␣␣␣␣␣␣␣␣␣==␣[0-9]$/
            $vertex_label[u] = {'vtype' => 'location',
                    'process' => $1,
                'location' => $2, 'operator' => 0}
        when /^\_loc\[\_Brick\(([0-9])\)\]␣==␣([0-9])$/
            $vertex_label[u] = {'vtype' => 'location',
                    'process' => $1,
                'location' => $2, 'operator' => 1}
        when /^Brick\(([0-9])\)\.colour␣==␣([0-9])$/
            $vertex_label[u] = {'vtype' => 'variable',
                    'operator' => 0,
                'value' => $2, 'process' => $1}
        when /^\_action[1-9][0-9]*$/
            $vertex_label[u] = {'vtype' => 'action',
                    'process' => $actions[u]}
        when /^Brick\(([0-9])\)\.x$/
            $vertex_label[u] = {'vtype' => 'clock',
                'process' => $1}
        end
    }

    $edge_label.each { |p, label|
        tmp = {}

        /([\[\]])([0-9]*)\;([0-9]*|INF)([\[\]])/\
            .match(label)

        tmp['leftb'] = case $1
        when '['
            "Bound.closed"
        when ']'
            "Bound.open"
        end

        tmp['rightb'] = case $4
        when ']'
            "Bound.closed"
        when '['
            "Bound.open"
        end

        tmp['max'] = if $3 == "INF" then
                         "Float.POSITIVE_INFINITY"
                     else
                         $3.to_s + ".0f"
                     end

        tmp['min'] = $2.to_s + ".0f"

        $edge_label[p] = tmp
    }
end

def print_dot_graph(g)
    f = RGL::DOT::Digraph.new("G")
    g.each_vertex { |v|
        if $vertex_label.has_key?(v) then
            case $vertex_label[v]['vtype']
            when 'location'
                label = "Location\n" +
                    "Process:␣" +
                        $vertex_label[v]['process'] +
                        "\n" +
                    "Location:␣" +
                        $vertex_label[v]['location'] +
                        "\n" +
                    "Operator:␣" +
                        $vertex_label[v]['operator'].to_s
            when 'clock'
                label = "Clock:␣x\n" +
                    $vertex_label[v]['process']
            when 'variable'
                label = "Variable\n" +
                    "Process:␣" +
                        $vertex_label[v]['process'] +
                    "\ncolour␣==␣" +
                        $vertex_label[v]['value']
            when 'action'
                label = "Action:\npush\nProcess:␣" +
                    $vertex_label[v]['process']
            else
                label = "Action:\nwait"
            end

            f << RGL::DOT::Node.new({'name' => v,
                "label" => label})
            end
        }
    g.each_edge { |u ,v|
        if $false_edge_list.include?([u,v]) then
            f << RGL::DOT::DirectedEdge.new({'from' => u,
                'to' => v, 'style' => 'dashed'})
        else
            if $edge_label.include?([u,v]) then
                label = "$" + ($edge_label[[u,v]]['leftb']
                        == 'Bound.open' ? '(' : '[') +
                    $edge_label[[u,v]]['min'][0...-3] + ";
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣" + ($edge_label[[u,v]]['max'] ==
                            'Float.POSITIVE_INFINITY' ?
                            '\inf' :
                        $edge_label[[u,v]]['max'][0...-3])
                        + ($edge_label[[u,v]]['rightb'] ==
                        'Bound.open' ? ')' : ']') + "$"
                f << RGL::DOT::DirectedEdge.new({'from' =>
                    u, 'to' => v, 'label' => label})
            else
                f << RGL::DOT::DirectedEdge.new({'from' =>
                    u, 'to' => v})
            end
        end
    }

    return f
end

def generate_code(g)
    code = "import␣java.util.Hashtable;\n\n"
    code = code + "class␣Strat␣{\n"
    code = code + "\tpublic␣static␣CDD␣tree;\n"
    code = code + "\tpublic␣static␣int␣BRICKS␣=␣" +
        $n.to_s + ";\n"

    code = code + "\tpublic␣static␣int␣x[]␣=␣" +
        "new␣int[BRICKS];"
    code = code + "\tpublic␣static␣int␣location[]" +
        "␣=␣new␣int[BRICKS];"
    code = code + "\tpublic␣static␣int␣colour[]␣=␣" +
        "new␣int[BRICKS];"

    code = code + "\n\tpublic␣static␣void␣makeStrategy()"
```

```
        + "␣{\n"
    code = code + "\t\tHashtable␣vertices␣=␣" +
        "new␣Hashtable();\n\n"

    g.each_vertex { |v|
        code = code + "\t\tvertices.put(\""
        code = code + v.to_s
        code = code + "\",␣"
        case $vertex_label[v]['vtype']
        when 'clock'
            code = code + "new␣CNode("
            code = code + $vertex_label[v]['process']
        when 'location'
            code = code + "new␣DNode(0,␣"
            code = code + $vertex_label[v]['process']
            code = code + ",␣"
            code = code +
                         $vertex_label[v]['location'].to_s
            code = code + ",␣0"
        when 'variable'
            code = code + "new␣DNode(1,␣"
            code = code + $vertex_label[v]['process']
            code = code + ",␣"
            code = code + $vertex_label[v]['value'].to_s
            code = code + ",␣"
            code = code +
                         $vertex_label[v]['operator'].to_s
        when 'action'
            code = code + "new␣ANode(" +
                $vertex_label[v]['process']
        end
        code = code + "));\n"
    }

    code = code +
        "\n\t\ttree␣=␣new␣CDD(␣(Node)␣vertices.get(\""
    code = code + $root
    code = code + "\"));\n"

    iterator = RGL::BFSIterator.new(g, $root)

    iterator.each { |v|
        g.adjacent_vertices(v).each { |w|
            code = code +
            "\t\ttree.addExistingNode((Node)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣vertices.get(\""
            code = code + v.to_s
            code = code + "\"),␣(Node)␣vertices.get(\""
            code = code + w.to_s
            code = code + "\")"
            case $vertex_label[v]['vtype']
            when 'clock'
                code = code + ");\n"
                code = code + "\t\t((Node)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣vertices.get(\""
                code = code + w.to_s
                code = code + "\")).setInterval("
                code = code + $edge_label[[v,w]]['leftb']
                code = code + ",␣"
                code = code + $edge_label[[v,w]]['min']
                code = code + ",␣"
                code = code + $edge_label[[v,w]]['max']
                code = code + ",␣"
                code = code + $edge_label[[v,w]]['rightb']
                code = code + ");\n"
            else
                if is_in_false?(v, w) then
                    code = code + ",␣false"
                else
                    code = code + ",␣true"
                end
                code = code + ");\n"
            end
        }
    }

    code = code + "\t}\n}"
    return code
end

g = graph_from_dotfile("tiga_dot_short.dot")
prune_wait_paths(g)

translate_labels("tiga_out")

parse_labels()

#puts print_dot_graph(g)
puts generate_code(g)
```

Bibliography

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[BC06] Patricia Bouyer and Fabrice Chevalier. On the control of timed and hybrid systems. *EATCS Bulletin*, 89:79–96, June 2006.

[BCD$^+$06] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. UPPAAL-Tiga: Timed Games for Everyone. In Luca Aceto and Anna Ingolfdottir, editors, *Proceedings of the 18th Nordic Workshop on Programming Theory (NWPT'06), Reykjavik, Iceland.* Reykjavik University, 2006.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* Massachusetts Institute of Technology, 2008.

[BLP$^+$99] Gerd Behrmann, Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 341–353, London, UK, 1999. Springer-Verlag.

[BMP03] Patricia Bouyer, P. Madhusudan, and Antoine Petit. Timed control with partial observability. In *In CAV 03, LNCS 2725*, pages 180–192. Springer, 2003.

[CDF$^+$05] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR'05*, volume 3653 of *LNCS*, pages 66–80. Springer–Verlag, August 2005.

[CDL$^+$07] Franck Cassez, Alexandre David, Kim G. Larsen, Didier Lime, and Jean François Raskin. Timed control with observation based and stuttering invariant strategies. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis*, number to appear in LNCS, page to appear. Springer, 2007.

[CL08] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems.* Springer Science+Business Media, LLC, second edition, 2008.

[LAS07] Kim Guldstrand Larsen Luca Aceto, Anna Ingólfsdóttir and Jiří Srba. *Reactive Systems Modelling, Specification and Verification.* Cambridge University Press, New York, 2007.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and symbolic model-checking of real-time systems. In *In Proc. of the 16th IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, 1995.

[LWYP98] Kim G. Larsen, Carsten Weise, Wang Yi, and Justin Pearson. Clock difference diagrams. Technical Report RS-98-46, December 1998. 18 pp. Presented at . Appears in *Nordic Journal of Computing*, 6(3):271–298, 1999.

[MPS95] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems. pages 229–242. Springer Verlag, 1995.