CONTROLLER PROGRAM SYNTHESIS
FOR INDUSTRIAL MACHINES

by

Hans-Jörg Peter

A diploma thesis in the

DEPARTMENT 6.2 - COMPUTER SCIENCE

ADVISORS

Prof. Bernd Finkbeiner, PhD
Prof. Dr.-Ing. habil. Hartmut Janocha

SAARLAND UNIVERSITY, GERMANY

November 2005

Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig und unter ausschließlicher Verwendung der angegebenen Hilfsmittel angefertigt habe.

Saarbrücken, im November 2005

This work is dedicated to my grandparents Hannelore and Nikolaus Peter

Abstract

In this thesis, a new synthesis algorithm for industrial controller programs is presented.

Verification and synthesis are the two basic approaches to guarantee that a system is correct. While verification requires the programmer to provide both the specification and the implementation, synthesis automatically transforms the specification into an implementation that is correct by construction.

The presented approach includes a new specification language that is geared towards usability in an industrial set-up. Specifications consist of two parts: a generic description of the machine components that is reused for different programs, and a description of the production goals that is specific to each program. The behaviour of the machine components is described by timed automata, while the production goals are captured by safety and bounded liveness properties.

The advantage of this approach is that the description of the goals, and thus of the behaviour of the overall system, is decoupled from the technical details of the machine components. This results in a high degree of re-usability, adaptivity, and maintainability. The specification of the machine components can be reused for different programs, and a reconfiguration of the machine no longer requires a time-consuming re-implementation.

The synthesis problem is solved by finding a memory-less strategy in a safety game. A winning strategy is transformed into an intermediate controller program, which controls the machine such that the production aims are met. The intermediate program is improved in several optimisation steps before it is cross-compiled for a machine controller.

The approach is illustrated with a prototype implementation that includes a cross-compiler for IEC 1131-3 assembler code. The implementation has been applied in several case studies using the Siemens S7-300 programmable logic controller, which is the current industrial standard.

Acknowledgements

First of all, thanks a lot to Prof. Bernd Finkbeiner, Joachim Fox, and Sven Schewe for giving me suggestions and valuable hints. In particular the discussions with Sven about safe and unsafe programs, as well as with Joachim about whether solving a problem by straight programming is easier than giving a formal specification, revealed some major issues that found their way into the thesis. Thanks to Prof. Hartmut Janocha for supporting me with technical equipment that made a real world application of the developed synthesis tool possible.

Furthermore, thanks to Martin Bauer for the helpful discussions and especially Lea Pfeifer for her patience and support, as well as to my parents Gertrud and Michael Peter.

Contents

Chapter 1

Introduction

1.1 Motivation

When writing a controller program for an industrial machine, the programmer has to be aware of two things:

1. *Safety*: some machine-operations may lead to undesired behaviour or even crashes.
2. *Bounded Liveness*: Within a certain amount of time, the production goals of the machine must be reached.

So, the programmer has to implement a program that regards the safety properties while trying to reach the goals quickly enough. The basic motivation for this thesis raised upon the following consideration: When specifying both safety properties and bounded liveness independently from each other, is it possible to generate the actual program completely automatically?

The following example illustrates this approach: Figure 1.1 shows two concurring robot arms A and B. A loads workpieces to the processing station P in the middle, while B unloads them to an outtake. Both can be controlled independently from each other.

The production goal is given by the bounded liveness criterion: ”At least every 22 seconds, a new workpiece must be processed”. On the other hand, if A and B would move simultaneously to the processing station, they would crash. So, the safety criterion is: ”It is not allowed that A and B are simultaneously at the processing station”. Or, more formally: $\neg(A_{PROCESS} \wedge B_{PROCESS})$. The loading and unloading operation take four seconds each. The processing operation takes six seconds. Since it was specified that 22 seconds as an overall cycle time are enough, a valid program could be:

1. Move A to the processing station in order to load a new workpiece (4s)
2. Process the workpiece (6s)
3. Move A back to its initial position (4s)
4. Move B to the processing station and grab the processed workpiece (4s)

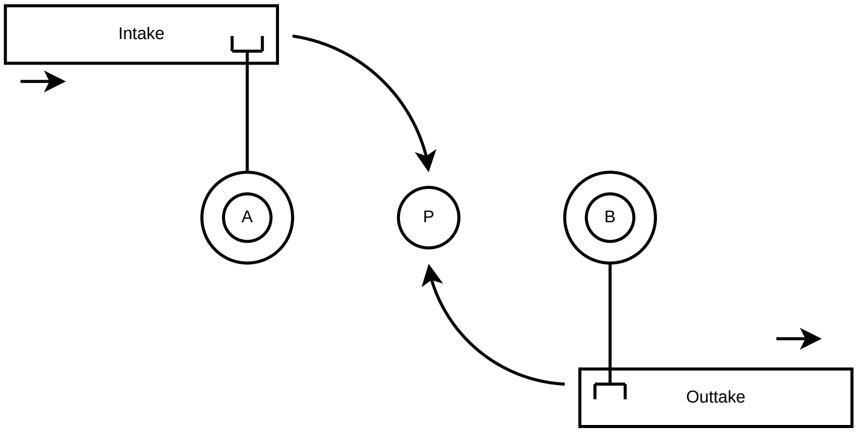


Figure 1.1: Two concurring robot arms

5. Move B to the outtake in order to unload the workpiece (4s)
6. Goto 1

Obviously, this is not the most time-optimal solution. Thus, specifying a stricter upper time bound, say 14 seconds, yields a more compact result:

1. Move A to the processing station in order to load a new workpiece (4s)
2. Do in parallel:

   Process the workpiece (6s)

   Move A back to its initial position (4s)

   Move B to the processing station (4s)
3. Grab the processed workpiece and move B to the outtake in order to unload that workpiece (4s)
4. Goto 1

Now, the synthesis algorithm is forced to exploit parallelism, since this is the only chance of solving the problem within the desired time. However, a time-bound that lies below 14 seconds is actually unrealisable. In this case, the synthesis algorithm indicates this by returning an empty program.

1.2 Correct Programs

Machine programs are a high sensitive field of software development. Small programming failures may result in a great financial disaster or could even be dangerous for human beings. Therefore, machine programs must provide a high degree of correctness under all circumstances.

Computer Aided Verification (CAV) has become a widely used technique in order to proof correctness of complex systems. Especially asynchronous setups lead to state spaces, which can only be exhaustively verified using automated methods. CAV works as follows: a specification is given and a programmer has to create a program, which inhibits that specification. Then, a model-checker (e.g. [16, 6, 7]) checks if the program satisfies the given specification. The disadvantage of this method is that a program must be given *manually*. The model-checker only gives feedback about the correctness of that program: either the specification is satisfied or else a counter-example is provided. The actual error in the program must be found and removed manually.

The approach presented in this thesis in order to get correct programs is the so called Controller Synthesis. Here, the user gives only a specification and the actual program generation works completely *automatically*. From the practical point of view, one can easily see that the synthesis approach needs significantly less user interaction than CAV does, since only the specification must be given manually (see figure 1.2).

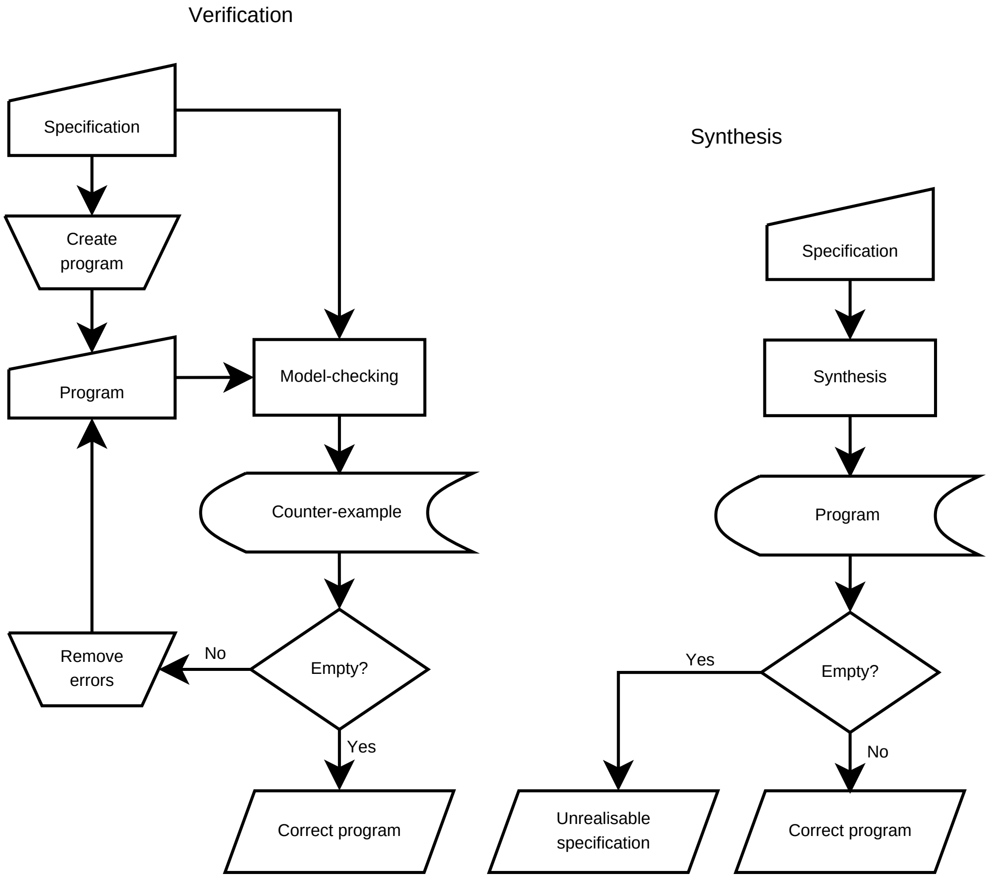


Figure 1.2: Verification vs. Synthesis

The gain in maintainability is also evident. Consider a machine that changes its behaviour (e.g. due to wear or parts exchange). Then, the controller program must be partially rewritten or at least anew verified. This results often in a waste of valuable production time due to the lack of automated methods. Using synthesis, one must only change the specification, which is, practically seen, a lot easier than reprogramming.

Specifying instead of programming means transforming the difficult task of creating a controller program into the rather easy task of describing a machines hardware and its production goals. Typically, when an industrial machine is planned the requirement specifications are defined. This is done by the customer together with the constructing engineer. The engineer decides which components should be used in such a way that the wishes of the customer are satisfied. Depending on those wishes, this task can be very difficult; consider running time requirements such as ”The overall cycle time must not be greater than 15 seconds”. Now, the engineer has to choose the (cheapest) components, which allow the programmer to write a program that inhibits the running time requirement.

Obviously, the complexer the machine, the better must be the engineer. With the classical method, the question whether a specification is realisable or not cannot be answered until the end of the programming phase, actually. Because first then, there exists a verifiable controller program. However, a fundamental change of the specification at this time due to planning failures, can have a major financial impact. On the other hand with the synthesis approach, both customer and engineer can easily check whether their specification can be realised. Thus, synthesis can be seen as a verification of the specification, where the concrete controller program is just a by-product of that analysis.

1.3 Overview

Essentially with this approach, the specification is a formal description of the system that has to be controlled, called plant, along with a definition of the production goals, called plans. Typically, a machine (plant) is constructed by combining multiple asynchronous hardware components with each other. So, the definition of the plant is given as a set of components. Because *real-time* is a major characteristic, one computational model for those components are Timed Automata. On the other hand, industrial machines are reactive. This means that the components may receive orders sent by a controller and respond with plant-messages by their own. The orders are called *controllable* events, while the plant-messages are called *uncontrollable* events. The occurrence of the latter are basically unpredictable for the controller. The computational model that copes with such a non-deterministic behaviour are Two Player Games. Combining both models, one gets Timed Game Automata, which form the computational basis for representing the plant components. Also part of the plant definition are state-assertions represented by first order formulas. The conjunction of those assertions is a safety predicate function. Along with the components, this safety predicate spans a Safety Game.

The production goals (plans) are timed automata that run in parallel with the plant components. They guarantee that within a finite amount of time, certain goals (key events) are achieved. If not, they enter a fail state. The plans can be seen as straightforward programs that describe a rough skeleton of the program that should be generated. Synthesising a valid program that matches the specification means finding a winning strategy for the given safety game.

Figure 1.3 shows the approach as a flowchart: First of all, based on the definition of the plant and the production goals, the timed game automata are constructed. Because the standard timed automata semantics implies implicit temporal behaviour, a preprocessing step that converts each implicit timed automaton into an explicit one is applied first. Then, the spanned safety game is solved by computing the winning control state space. This is done by an on-the-fly algorithm that explores the reachable states forwardly and removes the encountered fail states backwardly. The resulting winning strategy is non-deterministic, since in some states there might be multiple controller requests. A heuristic selects the optimal requests such that in each state there is at most only one possible decision for the controller. Then, that deterministic strategy is compiled into an intermediate controller program. In order to reduce the code-size, several post-optimisations are applied. Here, the dependencies that are defined in the specification are used to remove redundant testing commands. Finally, the concrete assembler program will be compiled from the optimised intermediate program. The language of the synthesised assembler code complies with IEC 1131-3. This gives the opportunity to test the programs on real hardware.

Chapter 2 describes the preliminaries of the computational models described in chapter 3. Those form the basis of the actual synthesis algorithm presented in chapter 5. In chapter 4, the specification language is formally defined and illustrated with some examples. Chapter 6 describes the code generation algorithms. Finally in chapter 7, some real world examples are shown that illustrate approaches for modelling common engineering problem tasks. Also, the developed tool and the generated code are presented.

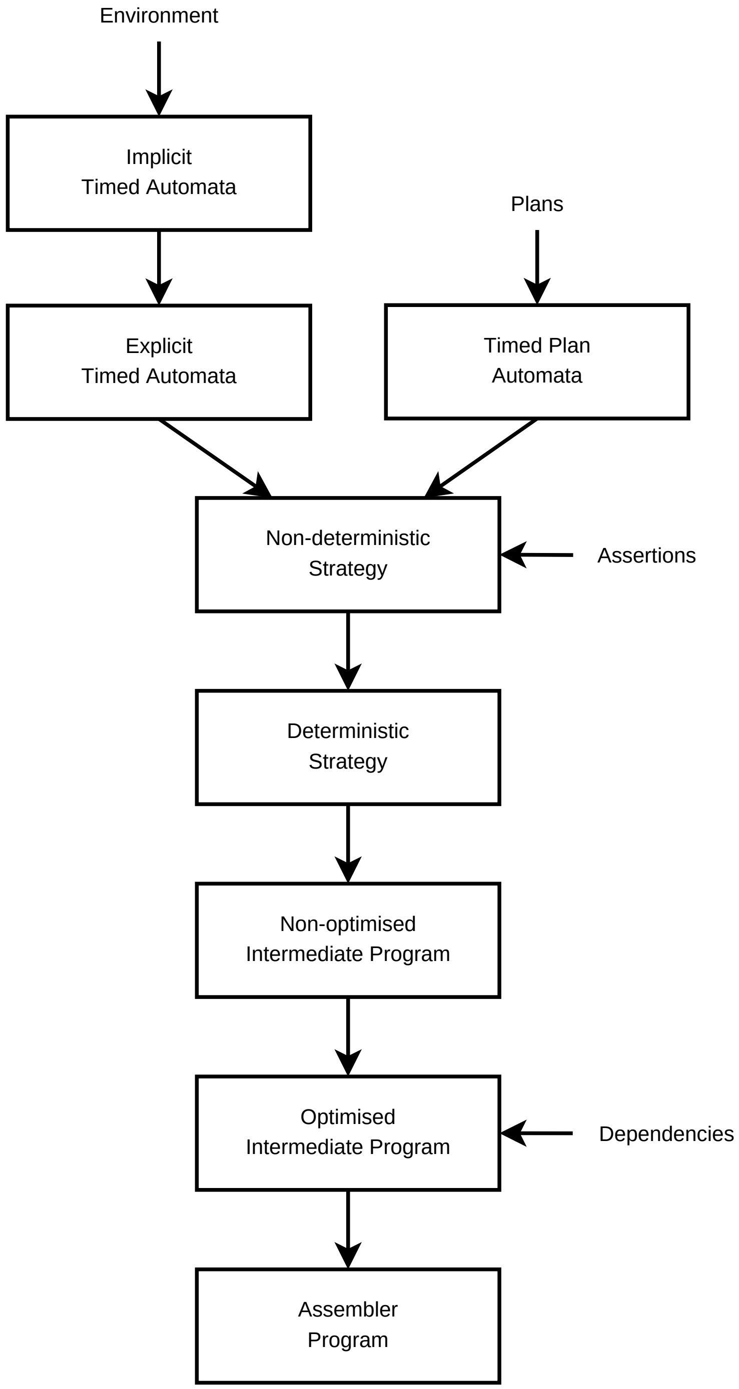


Figure 1.3: Synthesis algorithm overview

1.4 Related Work

There are two major directions in the broad field of program synthesis: *data*-intensive (or deductive) and *control*-intensive (or automata-theoretic) synthesis. With the data-intensive approach, one tries to generate programs based on proofs and, therefore, user interaction is necessarily needed [23, 20, 14]. Control-intensive approaches do not contain any form of data-type. Hence, the field of application is fairly restricted to theoretical problem tasks [13].

Real-time synthesis, which is the aim of this thesis, is based on automata to model the control states of the machine components. On the other hand, an infinite data-type, continuous real-time, is included. So one can say that real-time synthesis is a combination of data- and control-intensive synthesis.

Seen from the point of view of the controller, machine components may show an unpredictable behaviour. Thus, a non-deterministic computational model must be assumed. The standard for modelling real-time systems are *Timed Automata* [4]. Based on that, the authors of [19, 12] introduced the *Timed Controller Synthesis Problem* and formulated a solution as finding a winning strategy for an infinite game. They showed methods such as a fixed-point computation on the space of states and clock configurations.

Similar to them, the synthesis approach in this thesis bases on timed safety games. But in contrast to a pure backward propagation from the fail states on, a forward exploration combined with a backward fail state removal from the initial state on is used to find a winning strategy. Hereby, difference bound matrices [11] that represent clock zones [3] are used to symbolise the continuous part of the state space efficiently. The standard model-checking tool [6] has already shown that this technique works very well even for industrial setups.

In [5], based on timed automata extended by real-time tasks, a synthesis approach for the Lego Mindstorms system was developed. They generate C-code using the TIMES tool for checking reachability and schedulability of such automata. In contrast, for the target systems of this thesis, programmable logic controllers, a scheduling approach is not sufficient to deal with unpredictable plant behaviour.

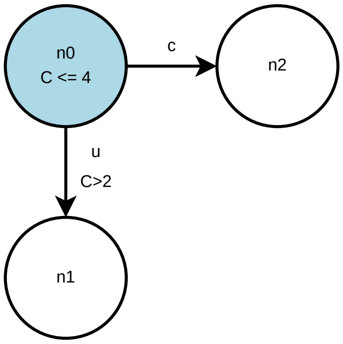


Figure 1.4: Standard simulation graph

In [1], it was shown why the standard simulation graph of a timed automaton cannot be used in a straightforward manner for solving timed games. It was proposed to build a quotient graph of the dense time transition system as a precomputational step. In this quotient graph, some transitions may only change their enabled-state by taking explicit delay-transitions, while in the simulation graph, this state can change while staying in a location and letting time pass. Applying that implicit-to-explicit conversion to the complete product transition system would be very expensive. However, since in this thesis the plant is defined in a modular manner, i.e. that each component is modelled as an independent automaton, the precomputation can be applied to each automaton independently. Because the clocks of the various automata do not overlap, the generation of the individual quotient graphs can be done in a local fashion. Thus, the increase of complexity is negligible, since the running time of this precomputation depends only on the number of the locations, edges, and clocks of the components, which are, by the way, rather tiny compared with the complete product state space.

In figure 1.4, there are two transitions: an uncontrollable transition u from n_0 to n_1 and a controllable c from n_0 to n_2. While c is always enabled, u becomes firstly enabled if two time units pass by. Obviously, the controller has the opportunity to avoid a possible occurrence of u by executing c within the first two time units. Therefore, in this case, u must be regarded as controllable. Figure 1.5 shows the corresponding quotient graph. Here, the nodes are split up according to their explicit decision-intervals. This yields a correct controllable/uncontrollable classification of all transitions.

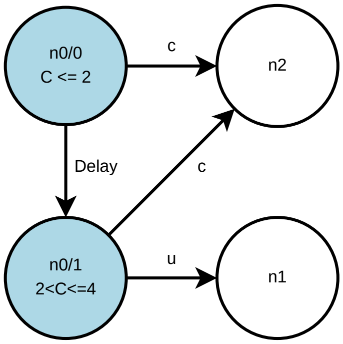


Figure 1.5: Time abstracted quotient graph

Independently to this thesis, a truly on-the-fly algorithm, i.e., without any precomputation, for solving games based on timed game automata was proposed by [8]. It extends the on-the-fly algorithm suggested by [18] for linear-time model-checking of finite-state systems. It is an interleaved combination of forward computation of the simulation graph together with back-propagation of information of winning states. Hereby, the winning status of an individual state can change, which causes a reevaluation of the predecessor states. However, due to an overlap of regions it is possible that some states are evaluated several times, implying the disadvantage that the complexity is not linear in the region graph. In contrast, the game solving algorithm in this thesis is linear in the size of the product graph.

Chapter 2

Preliminaries

In this chapter, the theoretical foundations of the two major computational models of this thesis are introduced. On the one hand, the behaviour of a machine is non-deterministic, i.e. that in a certain state, there can be many uncontrollable reactions that all have to be considered. [19] has shown that infinite games can be used to tackle the problem of a non-deterministic environment. On the other hand, it is needed to introduce an infinite time domain. The concept of Timed Automata by [4] is introduced to deal with continuous real-time. Both models are decidable and represent the theoretical foundations of the actual synthesis algorithm.

2.1 Infinite Games

The essence of each controller program is to handle unpredictable behaviour of the controlled machine components. In contrast to schedule-synthesis, it is not appropriate to create a static rule in which order some requests should be send to the plant such that a flawless cycle is always ensured. Handling such a reactive problem setup is the main topic of this section.

2.1.1 Game Arena

The *game arena* sets up the structure in which the controller plays against the non-deterministic plant. All controllable requests are denoted with a question mark (?) and all uncontrollable reactions are denoted with an exclamation mark (!) [1].

Formally, a game arena (or game structure) is a triple $G = (Q, \Sigma, \Delta)$ where Q is a finite set of states, Σ is a finite message alphabet, and $\Delta \subseteq Q \times \{?, !\} \times \Sigma \times Q$ is a transition relation, which defines the *game moves* for each side. Note that in contrast to the classical theory [19], where there are two types of states, namely controller and plant states, here, the information whether an action belongs to the controller or the plant is modelled within the transitions.

[1] The symbols are chosen that way because the plant is modelled from the point of view of the components

The *successor function* $\delta : 2^Q \rightarrow 2^Q$ is implied by a game arena and returns for a given set of states all successor states that are reachable by taking one move. It is defined as

$$\delta(Q') := \{t \mid \exists s \in Q' : (s, \tau, m, t) \in \Delta\}$$

The function $reach : Q \rightarrow 2^Q$ returns the set of all reachable states from a given origin state. It can be described by a least fixed-point formula:

$$reach(s_0) := \mu R\Big(\{s_0\} \cup \delta(R)\Big)$$

2.1.2 Safety Games

A *safety game* is triple $S = (G, A, I)$ where G is a game arena, $A : Q \rightarrow \{false,\ true\}$ is a safety predicate function, and $I \in Q$ is an initial state. The safety predicate identifies the game states as safe or unsafe (allowed or undesired, respectively). The set of *fail states* is implicitly defined as

$$fail(Q) := \{s \in Q \mid A(s) = false\}$$

Playing a safety game means that the plant must reach a fail state and the controller must prevent that in order to win the game. As already mentioned in the introduction, some goals should be reached while playing the game. Not reaching these goals within a finite amount of time is an undesired behaviour and is therefore classified as a fail state. So just letting the plant do nothing, or remaining in a cycle forever where no goals are achieved, is undesired.

An often quoted example for a safety game is a robber, representing the plant, who wants to escape from a building. A policeman (the controller) must hold him off. The building (the game arena) consists of several corridors that are linked directly or by lockable doors with each other. The robber runs through the corridors in order to find a way that leads outside. In the approach that is shown here, the policeman prevents the robber from fleeing by locking some doors (removing controllable transitions).

2.1.3 Solving a Safety Game

Of course, it is always desired that the controller will win. The main task in solving a safety game is to find a winning strategy for the controller. This strategy is the basis for any synthesised program. If there is no winning strategy for the controller, then there can not be any valid controller program as well. Anyway, a safety game is always *determined* such that one side, either the plant or the controller, is the clear winner (i.e. that there is no draw).

A set of winning states $W \subseteq Q$ describes a sub-graph of the original game arena such that the plant is unable to win within W. In order to determine W, the *attractor set* L of the fail states (i.e., all losing-states) must be computed. This can be done by evaluating the following least fixed-point expression:

$$\mu L\Big(fail(Q)\ \cup\ \{s \mid \exists(s, !, m, t) \in \Delta : t \in L\}\ \cup\ \{s \mid \forall(s, \tau, m, t) \in \Delta : t \in L\}\Big)$$

Algorithm 1 shows the explicit computational steps. Note that states having only uncontrollable outgoing edges are regarded as valid. As we can see later in chapter 6, such states are interpreted as ”wait until any of the uncontrollable events occurs”.

```
Input  : A priori known fail states $fail(Q)$.
Output: Controller losing states $L$.
$L \leftarrow fail(Q)$
repeat
    $L' \leftarrow L$
    $L \leftarrow L' \cup \{s \mid \exists(s,!,m,t) \in \Delta : t \in L\} \cup \{s \mid \forall(s,\tau,m,t) \in \Delta : t \in L\}$
until $L = L'$
return $L$
```

Algorithm 1: Computing the controller losing states

The set of the winning states W is the intersection of all reachable states from the initial state I with the inverse set of the losing states L:

$$W := reach(I) \cap \overline{L}$$

There exists a controller winning strategy iff $I \in W$.

2.2 Timed Automata

One approach to include time in the specification is to assume time to be *discrete*. When time is modeled in this manner, possible clock values are nonnegative integers and events can only occur at integer time values. This type of model is appropriate for *synchronous systems*, where all of the components are synchronised by a single global clock. The duration between successive clock ticks is chosen as the basic unit for measuring time. Since this thesis is assuming unsynchronised components, this approach is inappropriate.

Continuous time, on the other hand, is the natural model for *asynchronous systems*, because the separation of events can be arbitrarily small. This ability is desirable for representing causally independent events in an asynchronous system. Moreover, no assumptions need to be made about the speed of the plant when this model of time is assumed.

In order to model asynchronous systems using discrete time, it is necessary to discretise time by choosing some fixed time quantum so that the delay between any two events will be a multiple of this time quantum. This is difficult to do *a priori*, and may limit the accuracy with which systems can be modeled. Also, the choice of a sufficiently small time quantum to model an asynchronous system accurately may blow up the state space so that an exhaustive exploration is no longer feasible.

The *timed automaton model* of Alur, Courcoubetis, and Dill [4] has become the standard. Most on the research on continuous-time model-checking and synthesis is based on this model. In this chapter, the properties of timed automata are presented and the major techniques how to solve the *reachability* problem for such automata are explained.

The following definitions are taken from [9].

A *timed automaton* is a finite automaton augmented with a finite set of real-valued *clocks*. We assume that transitions are instantaneous. However, time can elapse when the automaton is in a node. When a transition occurs, some of the clocks may be resetted. At any instant, the reading of a clock is equal to the time that has elapsed since the clock was resetted. We assume that time passes at the same rate for all clocks.

A clock constraint, called a *guard*, is associated with each transition. The transition can be taken only if the current values of the clocks satisfy the clock constraint. A clock constraint is also associated with each node of the automaton. This constraint is called the invariant of the node. Time can elapse in the node as long as the invariant of the node is true.

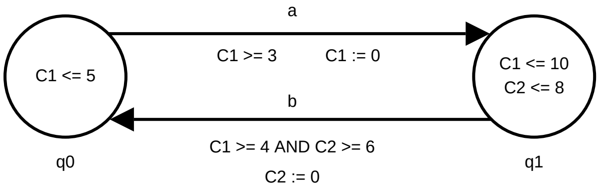


Figure 2.1: An example timed automaton

An example of a timed automaton is shown in Figure 2.1. The automaton consists of two nodes n_0 and n_1, two clocks C_1 and C_2, an a transition from q_0 to q_1, and a b transition from q_1 to q_0. The two clocks C_1 and C_2 start with value 0 and advance synchronously in time. The execution of the automaton starts at node q_0 where it can remain as long as the invariant of q_0 is satisfied, i.e. that $C_1 \leq 5$. Meanwhile, if $C_1 \geq 3$, one can make an a transition to the node q_1, resetting the clock C_1 back to 0. In q1, the execution can remain until either $C_1 > 10$ or $C_2 > 8$. If $C_1 \geq 4$ and $C_2 \geq 6$, a b transition back to q_0 becomes enabled that resets C_2.

The clock constraints are defined as follows: Let X be a set of *clock variables* and $C(X) := (X \rightarrow \mathbb{R}_0^+) \rightarrow \{false,\ true\}$. Then, the function $\varphi \subseteq C(X)$ describes *clock constraints* on X. Let $v : X \rightarrow \mathbb{R}_0^+$ be a *clock assignment*, which maps each clock in X to a nonnegative real value, then $\varphi(v)$ returns $true$ if the clock-values, indicated by v, inhibit the constraints or $false$ if not. φ is the conjunction of a finite number of inequalities $\{c_1(v),\ ...,\ c_n(v)\}$:

$$\varphi(v) \ := \ \bigwedge_{i=1}^{n} c_i(v)$$

Each inequality $c_i(v)$ can have one of the following forms:

- $c_i(v) \ = \ true$

- $c_i(v) \ = \ c_j \prec v(x_i)$
- $c_i(v) \ = \ v(x_i) \prec c_j$

where $\prec$ is either $<$ or $\leq$, $x_i \in X$, and $c_j \in \mathbb{Q}_0^+$. Note that if X contains k clocks, then each clock constraint is a convex subset of k-dimensional Euclidean space. Thus, if two points satisfy a clock constraint, then all of the points lined up in-between do satisfy the clock constraint.

Formally, a timed automaton is a 6-tuple $A = (Q, \Sigma, \Delta, q_0, X, I)$ such that

- Q is a finite set of nodes (also called locations).
- Σ is a finite input alphabet containing all controllable and uncontrollable messages.
- $q_0 \in Q$ is the initial node.
- X is a finite set of continuous clocks.
- $I : Q \rightarrow C(X)$ is a function that associates nodes with clock constraints, called the node invariants.
- $\Delta \subseteq Q \times (\Sigma \times C(X) \times 2^X) \times Q$ is a set of transitions. The 3-tuple (q, e, q') corresponds to a transition from node q to node q' labeled with the event e. The triple $\langle \alpha, \varphi, \lambda \rangle$ denotes an event with message α, a constraint φ that specifies when the transition is enabled, and a set of clocks $\lambda \subseteq X$ that are resetted when the transition is executed.

We will require that time be allowed to progress to infinity, that is, at each node the upper bound imposed on the clocks be either infinity, or smaller than the maximum bound imposed by the invariant and by the transitions outgoing from the node. In other words, it is possible to stay at a node forever, or the invariant will force the automaton to leave the node. If in the latter case no transition is enabled, a *timeout-edge* (T) is implicitly generated that leads the execution to a global timeout node.

2.2.1 Infinite State Transition Graph

A model for a timed automaton A is an *infinite state transition graph* $\mathcal{T}(A) = (S, \Sigma, R, s_0)$. Each state in S is a pair (q, v) where $q \in Q$ is a node, and $v : X \rightarrow \mathbb{R}_0^+$ is a clock assignment, mapping each clock to a nonnegative real value. The initial state s_0 is given by (q_0, v_0) where $\forall x \in X : v_0(x) = 0$.

In order to define the state transition relation for $\mathcal{T}(A)$, we must first introduce some notation. For $\lambda \subseteq X$, we will define the *clock resetting* operator as follows:

$$v[\lambda \leftarrow 0] : x \mapsto \left\{ \begin{array}{ccc} 0 & : & x \in \lambda \\ v(x) & : & x \notin \lambda \end{array} \right.$$

For $d \in \mathbb{R}$, the *time adding* operator is defined as: Let $v' = v + d$, then

$$\forall x \in X : v'(x) = v(x) + d$$

From the brief discussion in the beginning, we know that a timed automaton has two basic types of transitions:

- *Delay transitions* correspond to the elapsing of time while staying at some node. We write $(q, v) \xrightarrow{d} (q, v + d)$, where $d \in \mathbb{R}^+$, provided that for every $0 \leq e \leq d$, the invariant $I(q)$ holds for $v + e$.
- *Message transitions* correspond to the execution of a transition from Δ. We write $(q, v) \xrightarrow{a} (q', v')$, where $a \in \Sigma$, provided that there is a transition $(q, \langle a, \varphi, \lambda \rangle, q')$ such that v satisfies φ and $v' = v[\lambda \leftarrow 0]$.

The transition relation R of $\mathcal{T}(A)$ is obtained by combining the delay and message transitions. We will write $(q, v)\ R\ (q', v')$ or $(q, v) \overset{a}{\Longrightarrow} (q', v')$ if there exists a (q, v'') such that $(q, v) \xrightarrow{d} (q, v'') \xrightarrow{a} (q', v')$ for some $d \in \mathbb{R}$.

Now, our goal is to solve the *reachability problem* for $\mathcal{T}(A)$: Given an initial state s_0, we show how to compute the set of all states $s \in S$ that are reachable from s_0 by transitions in R. This problem is non-trivial because $\mathcal{T}(A)$ has an infinite number of states. In order to accomplish this goal, it is necessary to use a finite representation for the infinite state space of $\mathcal{T}(A)$. Developing such representations is the main topic of the following sections.

2.2.2 Clock Zones

An efficient way to obtain a finite representation for the infinite state space $\mathcal{T}(A)$ is to define *clock zones* [3], which also represent sets of clock assignments. A clock zone is a conjunction of inequalities that compare either a clock value or the difference between two clock values to an integer. We allow inequalities of the following types:

$$x \prec c, \quad c \prec x, \quad x \ - \ y \prec c,$$

where $\prec$ is $<$ or $\leq$.

By introducing a special clock x_0 that is always 0, it is possible to obtain a more uniform notation for clock zones. Since the value of a clock is always nonnegative, we will assume that constraints involving only one clock have the form

$$-c_{0,i} \prec x_i \prec c_{i,0},$$

where $-c_{0,i}$ and $c_{i,0}$ are both nonnegative. Using the special clock x_0, we will replace this constraint by the conjunction of two inequalities

$$x_0 - x_i \prec c_{0,i} \quad \wedge \quad x_i - x_0 \prec c_{i,0}.$$

Thus, the general form of a clock zone is

$$x_0 = 0 \quad \wedge \quad \bigwedge_{0 \leq i \neq j \leq n} x_i - x_j \prec c_{i,j}.$$

The following operations will be used to construct more complicated clock zones from simpler ones [3]. Let φ be a clock zone. If $\lambda \subseteq X$ is a set of clocks, then define $\varphi[\lambda \leftarrow 0]$ to be the set of all clock assignments such that

$$\forall v \in \varphi : \ \forall x \in \lambda : \ v(x) = 0.$$

If $d \in \mathbb{R}^+$, then we define $\varphi + d$ to be the set of all clock assignments $v + d$ where $v \in \varphi$. The set $\varphi - d$ is defined similarly.

Let φ be a clock zone expressed in terms of clocks in X. The conjunction φ will represent a set of assignments to the clocks in X. If X contains k elements, then φ will be a convex subset of k-dimensional Euclidean space. The following lemma shows that the projection of a clock zone onto a lower dimensional subspace is also a clock zone.

Lemma 2.2.1 *If φ is a clock zone with free clock variable x, then $\exists x[\varphi]$ is also a clock zone.*

This lemma turns out to be quite valuable in working with clock zones. A proof is given in [9].

Note that the assignment of values to the clocks in an initial state of timed automaton A is easily expressed as a clock zone since $v(x) = 0$ for every clock $x \in X$. Moreover, every clock constraint used in the invariant of an automaton location or in the guard of a transition is a clock zone. Because of the observation, clock zones can be used as the basis for various state reachability analysis algorithms for timed automata. These algorithms are usually expressed in terms of three operations on clock zones [3].

Intersection

If φ and ψ are two clock zones, then the intersection $\varphi \wedge \psi$ is a clock zone. This is easy to see. Because φ and ψ are clock zones, they can be expressed as conjunctions of clock constraints. Hence, $\varphi \wedge \psi$ is also a conjunction of clock constraints and, therefore, a clock zone.

Clock Reset

If φ is a clock zone and λ is a set of clocks, then $\varphi[\lambda \leftarrow 0]$ is a clock zone. We will show that this is true when λ contains a single clock. In this case, $\varphi[\lambda \leftarrow 0]$ is equivalent to $\exists[\varphi \wedge x = 0]$, and the result follows immediately by Lemma 2.2.1. The result can easily be extended to sets with more than one clock by induction.

Elapsing of Time

Geometrically seen, a clock zone φ is represented by a (bounded) polyhedron. The (unbounded) half-plane that is described by parallel translation of the polyhedron by 45° represents the clock zone that can be reached by time elapsing from an assignment in φ. This region is denoted by $\varphi^{\Uparrow}$.

Formally, if φ is a clock zone, then a clock assignment v will be an element of $\varphi^{\Uparrow}$, if v satisfies the formula $\exists t \geq 0 : (v - t) \in \varphi$ or, equivalently, $\exists t \geq 0 : v \in (\varphi + t)$. This region is a clock zone.

In principle, the three operations on clock zones described above can be used to construct a finite representation of the transition graph $\mathcal{T}(A)$ corresponding to a timed automaton. In the next section it is described how this algorithm can be implemented efficiently by using *difference bound matrices* [3, 11]. In this section states are represented by *zones* [3]. A zone is a pair (s, φ) where s is a location of the timed automaton and φ is a clock zone. Consider a timed automaton A with transition $e \;=\; (s,\ \langle a,\ \psi,\ \lambda\rangle,\ s')$. Assume that the current zone is (s, φ). Thus, s is a location of A, and φ is a clock zone. The clock zone $succ(\varphi, e)$ will denote the set of clock assignments v' such that for some $v \in \varphi$, the state (s', v') can be reached from the state (s, v) by letting time elapse and then executing the transition e. The pair $(s', succ(\varphi, e))$ will represent the set of successors of (s, φ) under the transition e. The clock zone $succ(\varphi, e)$ is obtained by the following steps:

1. Intersect φ with the invariant of location s to find the set of possible clock assignments for the current state.

2. Let time elapse in location s using the operator $\Uparrow$ described above.

3. Take the intersection with the invariant of location s again to find the set of clock assignments that still satisfy the invariant.

4. Take the intersection with the guard ψ of the transition e to find the clock assignments that are permitted by the transition.

5. Set all of the clocks in λ that are reset by the transition to 0.

Combining all of the above steps into one formula, one obtains

$$succ(\varphi, e) \;=\; ((\varphi \wedge I(s))^{\Uparrow} \;\wedge\; I(s) \;\wedge\; \psi)[\lambda \leftarrow 0]$$

Because clock zones are closed under the operations of intersection, elapsing of time, and resetting of clocks, the set $succ(\varphi, e)$ is also a clock zone.

Finally, we describe how to construct a transition system for a timed automaton A. The transition system is called the *zone graph* and is denoted by $Z(A)$. The states of $Z(A)$ are the zones of A. If s is the initial location of A, then $(s, [X \leftarrow 0])$ will be the initial state of $Z(A)$. There will be a transition from the zone (s, φ) in $Z(A)$ to the zone $(s', succ(\varphi, e))$ in $Z(A)$ labeled with the action a for each transition of the form $e \;=\; (s,\ \langle a,\ \psi,\ \lambda\rangle,\ s')$ of the timed automaton A. Because each step in the construction of the zone graph is effective, this gives an algorithm for determining state reachability in the state transition graph $\mathcal{T}(A)$. In the next section we will show how to make this construction more efficient.

Note that the standard reachability graph that is shown here is not appropriate for synthesis, which is the aim of this thesis. In section 5.2.2, this issue is discussed in detail.

2.2.3 Difference Bound Matrices

A clock zone can be represented by a *difference bound matrix* as described by Dill in [11]. This matrix is indexed by the clocks in X together with a special clock x_0 whose value is always 0. This clock plays exactly the same role as the clock x_0 in the previous section. Each entry $\mathcal{D}_{i,j}$ in the matrix $\mathcal{D}$ has the form $(d_{i,j},\ \prec_{i,j})$ and represents the inequality $x_i - x_j \prec d_{i,j}$, where $\prec_{i,j}$ is either $<$ or $\leq$, or $(\infty,\ <)$, if no such bound is known. Because the variable x_0 is always 0, it can be used for expressing constraints that only involve a single variable. Thus, $\mathcal{D}_{j,0} = (d_{j,0},\ \prec)$, means that we have the constraint $x_j \prec d_{j,0}$. Likewise, $\mathcal{D}_{j,0} = (d_{j,0},\ \prec)$, means that we have the constraint $0 - x_j \prec d_{0,j}$ or $-d_{0,j} \prec x_j$. Let $\mathfrak{D}$ donate the functional prototype of a difference bound matrix:

$$\mathfrak{D} := \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{Z} \times \{<, \leq\}$$

The representation of a clock zone by a difference bound matrix is not unique. In fact, a single clock zone can be represented by infinite many matrices. In general, the sum of the upper bounds on the clock differences $x_i - x_j$ and $x_j - x_k$ is an upper bound on the clock difference $x_i - x_k$. This observation can be used to progressively tighten the difference bound matrix. If $x_i - x_j \prec_{i,j} d_{i,j}$ and $x_j - x_k \prec_{j,k} d_{j,k}$, then it is possible to conclude that $x_i - x_k \prec'_{i,k} d'_{i,k}$ where

$$d'_{i,k} = d_{i,j} + d_{j,k}$$

and

$$\prec'_{i,k} = \begin{cases} \leq & : \quad \prec_{i,j} = \leq \ \wedge \ \prec_{j,k} = \leq \\ < & : \quad \text{else} \end{cases}$$

Thus, if $(d'_{i,k}, \prec'_{i,k})$ is a tighter bound than $(d_{i,k}, \prec_{i,k})$, one should replace the latter by the former so that $\mathcal{D}_{i,k} := (d'_{i,k}, \prec_{i,k})$. This operation is called *tightening* the difference bound matrix. We can repeatedly apply tightening to a difference bound matrix until further application of this operation does not change the matrix. The resulting matrix is a *canonical* representation for the clock zone under consideration. Note that a canonical difference bound matrix will satisfy the inequality $d_{i,k} \prec_{i,k} d_{i,j} + d_{j,k}$ for all possible values of the indices i, j, and k.

Finding the canonical form of a difference bound matrix can be automated by using the Floyd-Warshall algorithm [10], which has cubic complexity. The algorithm guarantees that all the possible combinations of indices are systematically checked to determine if further tightening is possible. We determine if a tighter bound can be obtained for $\mathcal{D}_{i,k}$ by checking if the inequality $d_{i,k} \prec_{i,k} d_{i,j} + d_{j,k}$ holds for all possible values of j. If the inequality does not hold for some value of j, then we replace $\mathcal{D}_{i,k}$ by $(d'_{i,k}, \prec'_{i,k})$ as described in the preceding paragraph. Algorithm 2 gives a description in mathematical pseudocode.

After the difference bound matrix has been converted to canonical form, we can determine if the corresponding clock zone is non-empty by examining the entries on the main diagonal of the matrix. If the clock zone described by the matrix is nonempty, all of the entries along the main diagonal will have the form $(0, \leq)$. If the clock zone is empty or unsatisfiable, there will be at least one negative entry on the main diagonal. Note that an entry on the main diagonal of the form $(0, <)$ indicates also an infeasible clock zone.

```
Input  : Non-canonical matrix $\mathcal{D}$ having $n$ clocks plus the zero-clock $x_0$.
Output: Canonical, tightened matrix $\mathcal{D}'$.
$\mathcal{D}' \leftarrow \mathcal{D}$
for $j \in \{0, ..., n\}$ do
    for $i \in \{0, ..., n\}$ do
        for $k \in \{0, ..., n\}$ do
            if $d'_{i,k} \geq d_{i,j} + d_{j,k}$ then
                $d'_{i,k} \leftarrow d_{i,j} + d_{j,k}$
                $\prec'_{i,k} \leftarrow \begin{cases} \leq & : \quad \prec_{i,j}=\leq \; \wedge \; \prec_{j,k}=\leq \\ < & : \quad \text{else} \end{cases}$
```

Algorithm 2: Tightening a difference bound matrix

Now, three operations on difference bound matrices are described. These operations correspond to the three operations defined on clock zones in the previous section.

- Intersection. We define $\mathcal{D} = \mathcal{D}^1 \wedge \mathcal{D}^2$. Let $\mathcal{D}^1_{i,j} = (c_1, \prec_1)$ and $\mathcal{D}^2_{i,j} = (c_2, \prec_2)$. Then $\mathcal{D}_{i,j} = (min(c_1, c_2), \prec)$, where $\prec$ is defined as follows:
    - If $c_1 < c_2$, then $\prec = \prec_1$.
    - If $c_2 < c_1$, then $\prec = \prec_2$.
    - If $c_1 = c_2$ and $\prec_1 = \prec_2$, then $\prec = \prec_1$.
    - If $c_1 = c_2$ and $\prec_1 \neq \prec_2$, then $\prec = <$.
- Clock reset. Define $\mathcal{D}' = \mathcal{D}[\lambda \leftarrow 0]$, where $\lambda \subseteq X$ as follows:
    - If $x_i, x_j \in \lambda$, then $\mathcal{D}'_{i,j} = (0, \leq)$.
    - If $x_i \in \lambda \; \wedge \; x_j \notin \lambda$, then $\mathcal{D}'_{i,j} = \mathcal{D}_{0,j}$.
    - If $x_i \notin \lambda \; \wedge \; x_j \in \lambda$, then $\mathcal{D}'_{i,j} = \mathcal{D}_{i,0}$.
    - If $x_i, x_j \notin \lambda$, then $\mathcal{D}'_{i,j} = \mathcal{D}_{i,j}$.
- Elapsing of time. Define $\mathcal{D}' = \mathcal{D}^{\Uparrow}$ as follows:
    - $\mathcal{D}'_{i,0} = (\infty, <)$ for any $i \neq 0$.
    - $\mathcal{D}'_{i,j} = \mathcal{D}_{i,j}$ if $i = 0$ or $j \neq 0$.

In each case the resulting matrix may fail to be in canonical form. Thus, as a final step, we must reduce the matrix to canonical form. All three of the operations can be implemented efficiently. Moreover, the implementation of these operations is relatively straightforward to program.

2.2.4 Complexity Considerations

The construction of a region graph [9, 2] is exponential in the number of clocks and also in the magnitude of the clocks, since for each interval between zero and the maximum integer constant, a corresponding state must exist. The size of the zone graph, on the other hand, also depends on the number of clocks but not on the magnitude of the clocks. Hereby, the number of zones only depends on the number of distinguishable clock differences. When modelling industrial machines, it is typical that the number of clocks is proportional to the number of components. Since complex machines may have many different components, it is crucial for the whole synthesis process to minimise the unavoidable temporal blowup. Of course, one cannot reduce the number of clocks, but one can choose the optimal clock value discretisation method, which are, in the oppinion of the author, clock zones.

However, the clock zones have one disadvantage: in loops, where unused clocks are not being resetted, they count to infinity, i.e. that the difference to those clocks that are actually resetted, do gradually increase. The result is that for each loop cycle, a new clock zone is generated, since this new zone does not lie within the old ones. One way to avoid such a diverging behaviour is to fix the inequality in the clock zone for all unused clocks C_u in the corresponding states to $0 \leq C_u - C < \infty$ for any other clock C. Alternatively, as a suggestion of a possible future work, one could construct additionally a region graph on top of the zone graph.

Chapter 3

Computational Models

In order to cope with continuous real time, on the one hand, and non-determinism, on the other hand, the two basic concepts of timed automata and infinite games are combined in one major computational model, timed game automata.

3.1 Timed Game Automata

While the nodes of an automaton represent local states of a system, the transitions symbolise occurring events. The combination of infinite games with timed automata is done by adding a message-type information to each transition. Formally, the finite input alphabet of a timed game automaton Σ_G is a tuple containing a message-label and a message-type:

$$\Sigma_G \subseteq \Sigma \times \{?,\ !,\ \$,\ D,\ T\}$$

The five message-types are defined as follows:

- **Controllable** messages, denoted by $?$, represent external *requests to force* the automaton to go into a certain state.
- **Uncontrollable** messages, denoted by !, are *spontaneous reactions* of the plant (e.g. reactions to a prior controller request).
- Transitions having **synchronised** messages, denoted by $\$$, can only be taken if the same message α occurs at the *same time* elsewhere. Synchronised transitions can be used to model internal communications among several automata.
- **Delay-** and **timeout-transitions**, denoted by D and T, are generated automatically by transforming an *implicit* timed automaton to an *explicit* one (see section 5.2.2).

One possibility to synchronise timed automata can either be done *implicitly* or *explicitly*. The classical implicit way is that if automaton A contains an uncontrollable message $!m$ and automaton B contains a controllable message $?m$, then the combined automaton is synchronised via a $?m$ transition. Later, it will be shown how an explicit synchronisation is used to synchronise the production goals with the events of the plant. Also, when modelling industrial machines, the experience was made that the explicit synchronisation method is more convenient than the implicit one.

3.2 Plan Automata

Plan automata are timed automata, that have an additional dedicated location $failed$. They are intended to run in parallel to the state space exploration in order to ensure that as long as no plan automaton enter $failed$ the represented plan is still maintained. A plan automaton has only synchronised transitions, and therefore, it produces no messages by its own. This means that no messages can be received from the controller or sent to the plant. Every synchronised transition represents a certain goal. Each message that is sent within the plant is passed to each plan automaton. If the current plan location has an outgoing transition containing the passed message from the plant, then that goal has been reached. Since $failed$ must always be a sink, a plan automaton can never exit this node.
Formally, a timed plan automaton $P = (Q, \Sigma, \Delta, q_0, failed, X, I)$ is a timed automaton with the following properties:

- $failed \in Q$ is a dedicated fail location.
- $\nexists(x, e, y) \in \Delta : \; x = failed$.
- $\forall(x, \langle\alpha, \tau, \varphi, \lambda\rangle, y) \in \Delta : \; \tau = \$$.

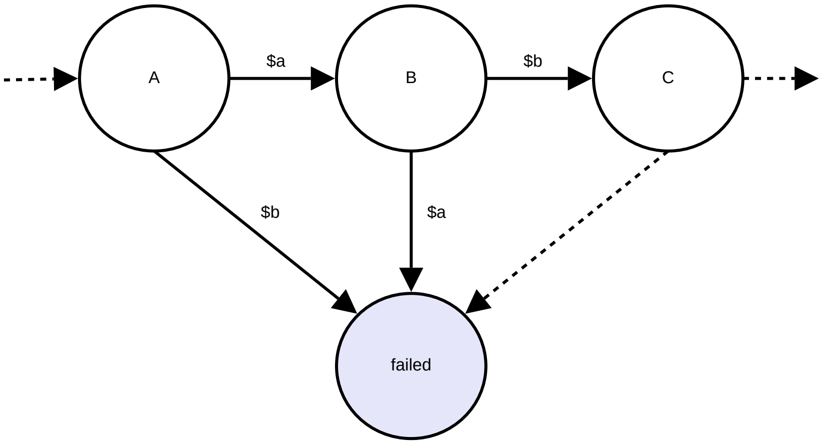


Figure 3.1: Part of a plan automaton

Figure 3.1 shows a part of a plan automaton. It represents the plan: reach event ”a” first then ”b”; the vice versa case is explicitly forbidden.

3.3 Parallel Composition

Paralleling two or more automata means combining their states. An interleaving or asynchronous semantics for this operation is assumed.
Let $A_1 = (Q_1, \Sigma_1, \Delta_1, q_0^1, X_1, I_1)$ and $A_2 = (Q_2, \Sigma_2, \Delta_2, q_0^2, X_2, I_2)$ be two timed automata with the following properties:

- A_1 and A_2 have disjoint sets of clocks: $X_1 \cap X_2 = \emptyset$.
- For each controllable/uncontrollable transition in A_1, A_2 must not have a controllable/uncontrollable transition with the same message:
  Let
  $$\Sigma_i(T) := \{\alpha \mid (x, \langle \alpha, \tau, \varphi, \lambda \rangle, y) \in \Delta_i \ \wedge \ \tau \in T\}$$
  be the set of all messages of a given set of transition-types T in the relation-set Δ_i, then
  $$\Sigma_1(\{?,\ !\}) \ \cap \ \Sigma_2(\{?,\ !\}) = \emptyset$$

. The parallel composition of A_1 and A_2 is the timed automaton

$$A_1 \parallel A_2 := (Q_1 \times Q_2,\ \Sigma_1 \cup \Sigma_2,\ \Delta,\ q_0^1 \times q_0^2,\ X_1 \cup X_2,\ I)$$

where $I(q_1, q_2) = I_1(q_1) \ \wedge \ I_2(q_2)$ and the edge relation Δ is computed by algorithm 3.

```
Input  : Two timed automata $A_1$ and $A_2$.
Output: Combined edge relation $\Delta$.
$\Delta \leftarrow \emptyset$
forall $(p, q) \in Q_1 \times Q_2$ do
    forall $(p,\ \langle \alpha_1,\ \tau_1,\ \varphi_1,\ \lambda_1 \rangle,\ p') \in \Delta_1$ do
        if $\alpha_1 \in \Sigma_2(\{\$\})$ then
            forall $(q,\ \langle \alpha_1,\ \$,\ \varphi_2,\ \lambda_2 \rangle,\ q') \in \Delta_2$ do
                $\Delta \leftarrow \Delta \cup \{((p,q),\ \langle \alpha_1,\ \tau_1,\ \varphi_1 \wedge \varphi_2,\ \lambda_1 \cup \lambda_2 \rangle,\ (p', q'))\}$
        else
            $\Delta \leftarrow \Delta \cup \{((p,q),\ \langle \alpha_1,\ \tau_1,\ \varphi_1,\ \lambda_1 \rangle,\ (p', q))\}$
    forall $(q,\ \langle \alpha_2,\ \tau_2,\ \varphi_2,\ \lambda_2 \rangle,\ q') \in \Delta_2$ do
        if $\alpha_2 \in \Sigma_1(\{\$\})$ then
            forall $(p,\ \langle \alpha_2,\ \$,\ \varphi_1,\ \lambda_1 \rangle,\ p') \in \Delta_1$ do
                $\Delta \leftarrow \Delta \cup \{((p,q),\ \langle \alpha_2,\ \tau_2,\ \varphi_1 \wedge \varphi_2,\ \lambda_1 \cup \lambda_2 \rangle,\ (p', q'))\}$
        else
            $\Delta \leftarrow \Delta \cup \{((p,q),\ \langle \alpha_2,\ \tau_2,\ \varphi_2,\ \lambda_2 \rangle,\ (p, q'))\}$
return $\Delta$
```

Algorithm 3: Computing the combined edge relation

Thus, the nodes of the parallel composition are pairs of nodes from the component automata, and the invariant of such a node is the conjunction of the invariants of the component nodes. There will be a transition in the parallel composition for each pair of synchronised transitions from the individual timed automata with the same label. The source node of the transition will be the composite node obtained from the source nodes of the individual transitions. The target node will be the composite node obtained from the target nodes of the individual transitions. The guard will be the conjunction of the guards for the individual transitions, and the set of clocks that are resetted will be the union of the sets that are resetted by the individual transitions. If the action of a transition is only an action of one of the two processes, then there will be a transition in the parallel composition for each node of the other timed automaton. The source and target nodes of these transitions will be obtained from the source and target nodes of the original transition and the node of the other automaton. All of the other components of the transition will remain the same.

Figure 3.2 shows a simple untimed composition. An untimed composition with a synchronised transition is shown in figure 3.3. Looking at this example, one can see that some states are never reached. Therefore, these states do not need to be included in the total combined state space. So it concludes that when computing the parallel composition of multiple automata, it is smarter to do this in conjunction with the reachability computation in order to avoid non-reachable states at all.

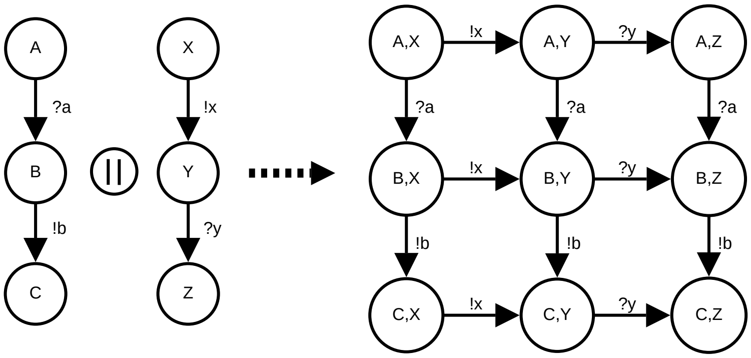


Figure 3.2: Simple untimed parallel composition

3.4 Combined State Space

A *combined configuration (state)* is a pair $(q^*, z) \subseteq (Q_1 \times Q_2 \times ... \times Q_n) \times \mathfrak{D}$. Each Q_i is a finite node set of a corresponding timed automaton $A_i = (Q_i, \Sigma_i, \Delta_i, root_i, X_i, I_i)$, for any $n \in \mathbb{N}$. z represents a difference bound matrix that stands for a specific clock zone. q_i^* refers to the i-th component of the node-tuple q^*. The set $\mathcal{A} = \{A_1, A_2, ..., A_n\}$ is called the *basis* of (q^*, z).

A *combined state space* is a quadruple (S, Σ, Δ, s_0) with basis $\mathcal{A}$. Let $\forall A_i \in \mathcal{A}: \ A_i = (Q_i, \Sigma_i, \Delta_i, root_i, X_i, I_i)$, then

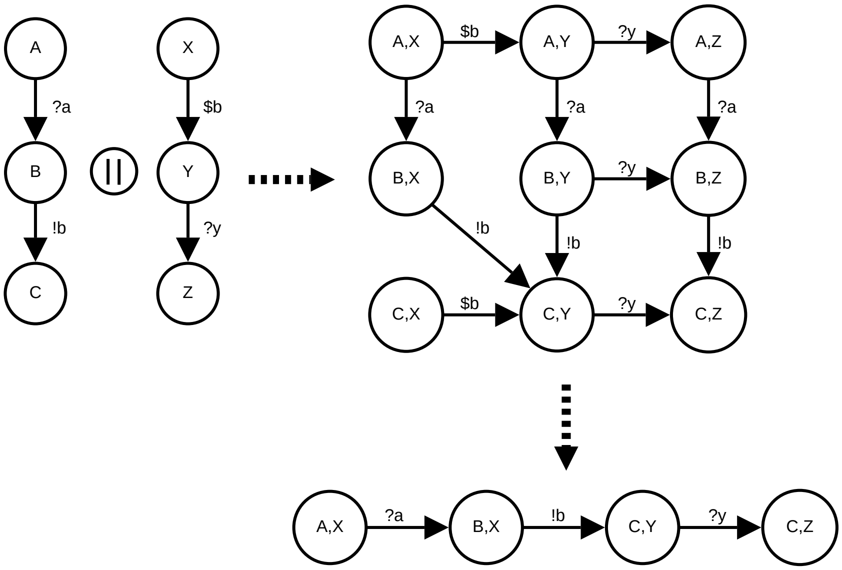


Figure 3.3: Untimed synchronised parallel composition

- $S \subseteq (Q_1 \times Q_2 \times ... \times Q_n) \times \mathfrak{D}$ is a finite set of discrete combined states.
- $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$ is the combined set of the messages of the basis.
- $s_0 \in S$ is the initial combined state.
- $\Delta \subseteq S \times (\Sigma \cup C(X) \times \{!,\ ?,\ \%,\ D,\ T\} \times 2^X) \times S$ is a set of transitions. The triple (s, e, s') corresponds to a transition from state s to state s' labeled with the event e. The triple $\langle \alpha, \tau, \lambda \rangle$ denotes an event with message or either delay-condition α, type τ, and a set of clocks λ that are resetted when the transition is executed.

When computing a state space by combining multiple timed automata, all synchronised transitions are resolved such that the resulting state space has only uncontrollable (!), controllable (?), or guarded (%) transitions. There are two types of ”time advancing” transitions: delay (D) and timeout (T). Timeout transitions are delay transitions that lead to undesired states.

Chapter 4

Specification Language

In this chapter, the developed specification language is introduced. In order to define the formal semantics, for each abstract syntactical construct, the denotational function $\mathcal{D}$ defines how the mathematical representation is extracted from the abstract syntax. Later, these mathematical objects are used as the input of the state space algorithm in chapter 5.

The specification is structured into the parts: plant definition and production goals.

$$specification \quad ::= \quad plant \quad goals$$

4.1 Plant Definition

All components that are part of the controllable plant are described in this section.

$$\begin{aligned} plant \quad &::= \quad components \quad assertions \quad dependencies \\ components \quad &::= \quad \textbf{plant}\ '\{'\ componentlist\ '\}' \\ componentlist \quad &::= \quad component\ ';'\ \{component\ ';'\ \} \\ component \quad &::= \quad automaton \mid hardware \mid operator \end{aligned}$$

The plant can contain multiple components. A component can either be an explicitly defined timed automaton, a hardware component, or an operator component.

4.1.1 Automaton Component

Declaring a component as an automaton, one can define arbitrary user defined timed game automata that comply with the theoretical definition from section 3.1.

$$
\begin{array}{rcl}
automaton & ::= & \textbf{automaton}\ \mathit{label}\ '\{'\ \mathit{automatonbody}\ '\}' \\
automatonbody & ::= & [\ clocksdef\ ]\ nodesdef\ transitionsdef \\
clocksdef & ::= & \textbf{clocks}\ \mathit{clock}\ \{','\ \mathit{clock}\ \}\ ';' \\
nodesdef & ::= & \textbf{nodes}\ nodedef\ \{','\ nodedef\ \}\ ';' \\
nodedef & ::= & \mathit{node}\ [\ \mathit{constraintlist}\ ] \\
constraintlist & ::= & '\{'\ \mathit{constraint}\ \{\ \textbf{and}\ \mathit{constraint}\ \}\ '\}' \\
constraint & ::= & \mathit{clock}\ \textbf{in}\ interval \\
interval & ::= & (\ '('\ |\ '['\ )\ \mathit{constant}\ ','\ \mathit{constant}\ (\ ')'\ |\ ']'\ ) \\
transitionsdef & ::= & transition\ \{\ ';'\ transition\ \}\ ';' \\
transition & ::= & \mathit{node}\ '\rightarrow'\ \mathit{node}\ transitionargs \\
transitionargs & ::= & \mathit{event}\ [\ \mathit{constraintlist}\ ] \\
 & & [\ \textbf{reset}\ \mathit{clocklist}\ ] \\
 & & [\ \textbf{instant}\ ] \\
event & ::= & \mathit{eventtype}\ \mathit{message} \\
eventtype & ::= & (\ '?'\ |\ '!'\ |\ '\$'\ ) \\
clocklist & ::= & '\{'\ \mathit{clock}\ \{','\ \mathit{clock}\ \}\ '\}'
\end{array}
$$

According to [4], each node may have a list of clock constraints, representing the invariant of the node. Analogously, the transitions may also have clock constraints, representing the guards of the edges. Since a constraint for a single clock is always convex, it can be specified as an interval. If no invariant or guard is present, then the constraints function will always return $true$. An interval has a lower- and an upper bound, represented by two integers. The bounds can be either declared as open (”(” or ”)”) or closed (”[” or ”]”). Alternatively, the upper bound can also be infinity (”inf)”). A transition must have a message and may reset some clocks. The keyword *instant* indicates that a transition should be taken prioritised, before any other. A message can either be controllable (”?”), uncontrollable (”!”), or synchronised (”$”).

Example Code

The following example code models the automaton from figure 2.1.

```
plant {
  ...
  automaton A {
    clocks c1, c2;
    nodes q0{c1 in [0,5]}, q1{c1 in [0,10], c2 in [0,8]};
    q0 -> q1 ?a {c1 in [3,inf)} reset{c1};
    q1 -> q0 !b {c1 in [4,inf) and c2 in [6,inf)} reset{c2};
  }
  ...
}
```

4.1.2 Hardware Component

A *hardware* component represents a syntactical frontend to the automaton component. Which means that each hardware unit can be expressed by an equivalent automaton unit. It is integrated for convenience only, since it represents an often occurring pattern in industrial component design.

A hardware component can move to certain defined resting states. This happens when the controller has sent a corresponding request (e.g. ”Goto state X”, where X is one of the defined resting states). Then, the component will firstly enter an intermediate state, representing the movement phase from the origin resting state to the destination state. Also, the corresponding clock is resetted that measures the time that passes during that movement. After the clock-value enters the given time-interval for the movement, the component reaches the destination state and sends a corresponding uncontrollable event (e.g. ”State X has been reached”) that the controller can receive.
The grammar is defined as follows:

$$\begin{array}{rcl}
hardware & ::= & \textbf{hardware}\ \mathit{label}\ '\{'\ hardwarebody\ '\}' \\
hardwarebody & ::= & statesdef\ movesdef \\
statesdef & ::= & \textbf{states}\ \mathit{state}\ \{','\ \mathit{state}\ \}\ ';' \\
movesdef & ::= & \mathit{move}\ \{';'\ \mathit{move}\ \}\ ';' \\
move & ::= & \mathit{state}\ '\rightarrow'\ \mathit{state}\ \textbf{takes}\ \mathit{interval}
\end{array}$$

Example code

The component modeled by the following example code is visualised in figure 4.1. Figure 4.2 shows the resulting automaton.

```
plant {
  ...
  hardware A {
    states X, Y, Z;
    X -> Y takes [2,4];
    Y <- X takes [2,3];
    Y -> Z takes [3,4];
    Z <- Y takes [1,2];
  }
  ...
}
```

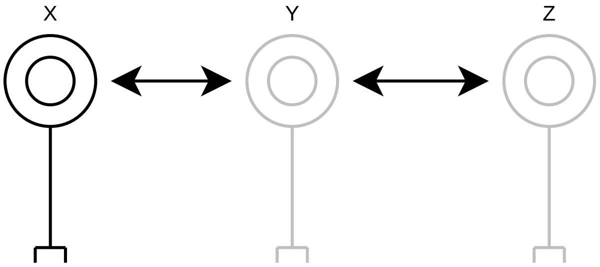


Figure 4.1: A hardware component with three resting states

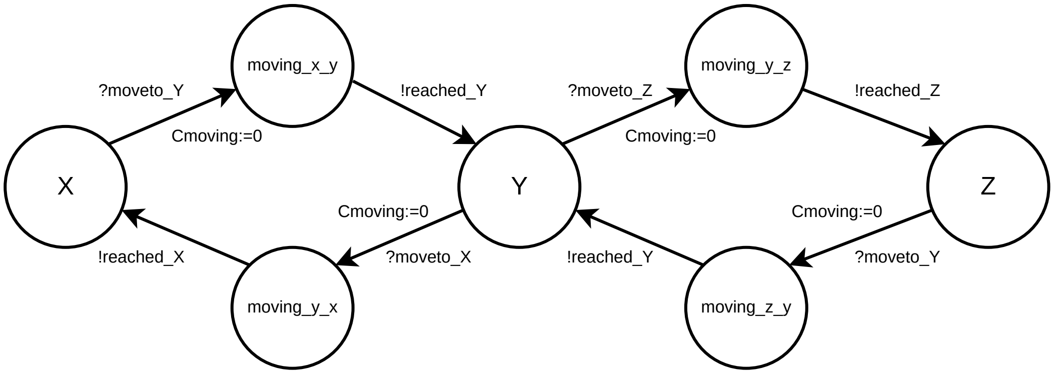


Figure 4.2: Figure 4.1 represented as an automaton

4.1.3 Operator Component

Like the hardware unit, the *operator* unit is also a syntactical frontend to the timed automaton unit. It can be used to model components that wait for a certain external operator input and then perform immediately a reaction. The canonical example for such a unit is a press/release-button. A precise elaboration of this problem is given in section 7.2.1.
The definition of the grammar is similar to the hardware unit:

$$\begin{aligned}
operator &::= \textbf{operator}\ label\ '\{'\ operatorbody\ '\}' \\
operatorbody &::= statesdef\ movesdef \\
statesdef &::= \textbf{states}\ state\ \{','\ state\ \}\ ';' \\
movesdef &::= move\ \{';'\ move\ \}\ ';' \\
move &::= state\ '\rightarrow'\ state
\end{aligned}$$

Example code

```
plant {
  ...
  operator button {
    states released, pressed;
    released <-> pressed;
  }
  ...
}
```

4.1.4 Assertions

The assertions describe the safety predicate, which was introduced in 2.1.2. It is represented as a conjunction of propositions of combined node tuples. They can be used to explicitly define undesired states or to set up global state invariants that must always hold.
The grammar is defined as follows:

$$
\begin{aligned}
assertions &::= \textbf{assertions}\ '\{'\ nodeconditionlist\ '\}' \\
nodeconditionlist &::= nodeconditiondef\ \{\ nodeconditiondef\ \} \\
nodeconditiondef &::= never \mid always \mid onlyif\ ';' \\
never &::= \textbf{never}\ nodecondition \\
always &::= \textbf{always}\ nodecondition \\
onlyif &::= nodecondition\ \textbf{onlyif}\ nodecondition \\
nodecondition &::= or \\
or &::= and\ [\ \textbf{or}\ or\ ] \\
and &::= literal\ [\ \textbf{and}\ and\ ] \\
literal &::= [\ \textbf{not}\ ]\ terminal \\
terminal &::= (\ '('\ or\ ')' \mid state \mid constant\ ) \\
state &::= <unit>\ '.'\ <node>\ [\ '*'\ ] \\
constant &::= (\ true \mid false\ )
\end{aligned}
$$

One can see that not has the highest operator priority, then and, and at least or. With the token $state$, one can refer a single state or a set of states of a certain unit, which was defined in the plant. Multiple states can be referenced by adding a star ($*$) at the end. Doing so, a substring match is performed instead of an equality check. For example, $X.moving*$ references all moving states of a unit X. While $X.moving\_a\_b$ references only one state.

Example code

In the following example code, two components are modeled, controlling a robot arm. The component `vertical` represents a valve that moves the robot arm `down` and `up`, while `horizontal` moves the arm from `left` to `right`. Now, consider that some obstacle is in the physical moving range of the robot arm, such that it must not move horizontally as long as it is not in the upper vertical position. This set-up can be modelled with this plant definition:

```
plant {
  hardware vertical {
    states down, up;
    down <-> up takes [1,2];
  }
  hardware horizontal {
    states left, right;
    left, right takes [1,2];
  }
}
```

and these assertions:

```
assertions {
  horizontal.moving* onlyif vertical.up;
}
```

Note that this definition is equivalent to:

```
assertions {
  horizontal.moving* and not vertical.up;
}
```

This example is visualised in figure 4.3. Here, the hatched states indicate the fail states in which the robot arm would crash with the obstacle.

4.1.5 Dependencies

Dependency information about the events that the plant produces can later be used to optimise the total code size of the intermediate controller program. Using the following grammar, one can define that some uncontrollable (in-) messages depend on one or more controllable (out-) messages.

$$\begin{array}{rcl}
dependencies & ::= & \mathbf{dependencies}\,'\{'\ dependencylist\,'\}' \\
dependencylist & ::= & dependency\ \{\ dependency\ \} \\
dependency & ::= & inmessage\ \mathbf{depends\ on}\ outmessagelist\,';' \\
outmessagelist & ::= & (\ \mathbf{time}\ |\ outmessage\ )\ ','\ outmessage
\end{array}$$

The terminal $\mathbf{time}$ can be used to define that an in-message depends on letting time pass. Let $!m$ be a message that only depends on time. If $!m$ is queried once and the result is $false$, then every succeeding query will be also $false$ as long as no time passes by.

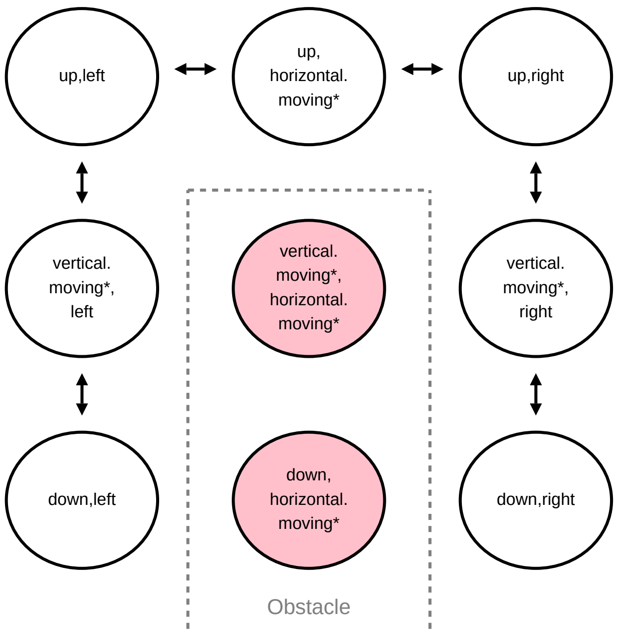


Figure 4.3: Two dimensional moving robot arm with obstacle

Example Code

Consider a hardware component that needs at least x time units to move between two resting states A and B. Now, when x time units have passed, the controller can start querying for the message `!reached_B`. A priori, it is clear that the outcome of a query of `!reached_B` changes only if time passes by. This can be formulated as follows:

```
dependencies {
  reached_B dependson time;
}
```

4.2 Production Goals

In order to give a complete specification of the program that should be synthesised, the production goals must be also defined. They consists of the so called state guards and a sequence of plan automata.

$$goals \quad ::= \quad guards \quad plans$$

4.2.1 State Guards

Guards can be used to ensure that certain plant states are only entered when a corresponding assertion holds. This assertion is interpreted as an uncontrollable message. Starting from a certain state A, when an adjacent state B is to be guarded, an intermediate state is created and inserted between A and B. The grammar is defined as follows:

$$\begin{aligned} guards &::= \mathbf{guards}\,'\{'\, guardlist\,'\}' \\ guardlist &::= guard\ \{\ guard\ \} \\ guard &::= nodecondition\ \mathbf{guardedby}\ message\,';' \end{aligned}$$

Note that $nodecondition$ refers to the token, which was already declared in section 4.1.4.

Example Code

A gripper may grasp a workpiece only when a corresponding sensor signalises that a workpiece is actually available. This can be modeled as follows:

```
plant {
  hardware gripper {
    states open, closed;
    open <-> closed takes [1,1];
  }
}
...
guards {
  gripper.moving_open_closed guardedby gear_is_present;
}
```

This example is visualised in figure 4.4. Here, a new guard node is inserted between the two nodes $gripper.open$ and $gripper.moving\_open\_closed$.

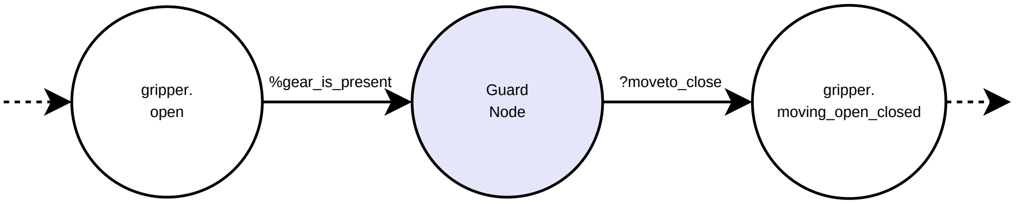


Figure 4.4: A gripper that may only close if the guard holds

4.2.2 Plans

In this part of the input specification, one can specify one or more plans. A plan describes which goals should be reached within a certain amount of time. A goal can be any message that was produced by the plant or sent by the controller. The grammar looks as follows:

$$
\begin{array}{rcl}
plans & ::= & \textbf{plans}\,'\{'\,planlist\,'\}' \\
planlist & ::= & plan\,\{\,plan\,\} \\
plan & ::= & \textbf{plan}\,'\{'\,[\,clockdeflist\,]\,plancommandlist\,'\}'\,';' \\
clockdeflist & ::= & \textbf{clocks}\,clockdef\,\{','\,\,clockdef\,\}\,';' \\
clockdef & ::= & clock\,constant \\
plancommandlist & ::= & [\,label\,'\,:'\,]\,resetcommand\,|\,waitforcommand\,\,';' \\
resetcommand & ::= & \textbf{reset}\,\,clock \\
waitforcommand & ::= & \textbf{waitfor}\,\,waitforlist \\
waitforlist & ::= & message\,[\,'->'\,\,label\,]
\end{array}
$$

The top level is a $planlist$. All plans must be maintained independently from each other, i.e. that a conjunctive semantics is assumed. Each plan consists of a definition of timeout-clocks and a list of plan commands. A plan command can optionally have a label. There are two types of plan-commands: the *reset-command* resets a timeout-clock back to 0 and the *waitfor-command* halts the execution of the plan until the given message occurs. Unless no target label is specified, the execution will continue with the following command. If the end of the plan is reached, an implicit jump to the begin is made. At any time, if a timeout-clock exceeds its given upper time bound, the plan will enter a fail state.

Example Code

Consider a gear checking machine that consists (among other things) of an *inlet* that loads a new gear into the machine, a *classifier* that measure some properties of the loaded gear, and an *outlet* that discharges the gear again. The transporting system of the machine is quite complex and has some perils due to obstacles in the moving range and limitations of the used hardware. The machine should reach the following goals:

1. gear is `loaded`,
2. gear is `classified`, and
3. gear is `unloaded`.

While waiting for a certain goal, the other goals must be avoided (e.g. while waiting for `loaded`, `classified` or `unloaded` must not occur). This can be modeled by the following code:

```
plans {
  plan {
    clocks timeout 15;

    reset timeout;
    waitfor loaded, classified -> failed, unloaded -> failed;
    waitfor classified, loaded -> failed, unloaded -> failed;
    waitfor unloaded, loaded -> failed, classified -> failed;
  }
}
```

With this plan, the synthesis algorithm will generate a program that controls the machine components in such a way that all the goals are reached (i) in the correct order and (ii) quickly enough.

4.3 Semantics

4.3.1 Automaton Component

The denotational semantics describes how the abstract syntax of an automaton is translated into the formal representation $(Q,\ \Sigma,\ \Delta,\ q_0,\ X,\ I)$.

$$\begin{array}{l}
\mathcal{D}[automatonbody(clocksdef,\ nodesdef,\ transitionsdef)]\ :=\\
\quad \textbf{let}\\
\qquad X\ :=\ \mathcal{D}_X[clocksdef]\\
\qquad (Q,\ q_0,\ I)\ :=\ \mathcal{D}_Q[nodesdef]\\
\qquad (\Sigma,\ \Delta)\ :=\ \mathcal{D}_\Delta[transitionsdef]\\
\quad \textbf{in}\\
\qquad (Q,\ \Sigma,\ \Delta,\ q_0,\ X,\ I)
\end{array}$$

$$\mathcal{D}_X[clocksdef(x_1,\ ...,\ x_n)]\ :=\ \{x_1,\ ...,\ x_n\}$$

$$\mathcal{D}_Q[nodesdef((q_1,\ I_1),\ ...,\ (q_n,\ I_n))]\ :=\ (\{q_1,\ ...,\ q_n\},\ q_1,\ \bigcup_{i=1}^{n} \{q_i \mapsto \mathcal{D}_C[c_i]\})$$

$$\begin{array}{l}
\mathcal{D}_\Delta[transitionsdef((q_1,\ q_1',\ \tau_1,\ \alpha_1,\ \varphi_1,\ \lambda_1),\ ...,\ (q_n,\ q_n',\ \tau_n,\ \alpha_n,\ \varphi_n,\ \lambda_n))]\ :=\\
\quad (\{\alpha_1,\ ...,\ \alpha_n\},\ \{(q_i,\ \langle\alpha_i,\ \tau_i,\ \mathcal{D}_C[\varphi_i],\ \lambda_i\rangle,\ q_i') \mid 1 \le i \le n\})
\end{array}$$

$$\mathcal{D}_C[\bot]\ :=\ \lambda v.true$$

$$\mathcal{D}_C[constraintlist(c_1,\ ...,\ c_n)]\ :=\ \lambda v. \bigwedge_{i=1}^{n} \mathcal{D}_C[c_i](v)$$

$$\begin{array}{l}
\mathcal{D}_C[constraint(clock,\ lb,\ ub)]\ :=\\
\quad \lambda v.\mathcal{D}_C[lowerbound(clock,\ lb)](v) \wedge\\
\quad \lambda v.\mathcal{D}_C[upperbound(clock,\ ub)](v)
\end{array}$$

$$\begin{array}{l}
\mathcal{D}_B[lowerbound(clock,\ type,\ const)]\ :=\\
\quad \textbf{if}\ type =' [' \ \textbf{then}\ \lambda v.const \le v(clock)\ \textbf{else}\ \lambda v.const < v(clock)
\end{array}$$

$$\begin{array}{l}
\mathcal{D}_B[upperbound(clock,\ type,\ const)]\ :=\\
\quad \textbf{if}\ type =']' \ \textbf{then}\ \lambda v.v(clock) \le const\ \textbf{else}\ \lambda v.v(clock) < const
\end{array}$$

$\mathcal{D}_X$ returns the set of clocks X. $\mathcal{D}_Q$ returns the set of nodes Q, the root node q_0, which is the first declared node, and the invariant-function I that maps each node q_i to its corresponding clock constraints c_i. If no clock constraints are present, $I(q_i)$ returns always $true$. The function $\mathcal{D}_C$ generates for a given clock constraint a corresponding constraint function. The function $\mathcal{D}_B$ generates an inequality function for a given interval bound.

4.3.2 Hardware Component

The denotational semantics shows how a hardware-unit is transformed into a timed automaton:

```
$\mathcal{D}[hardwarebody(statesdef,\ movesdef)]\ :=$
    $\mathbf{let}$
        $(Q_R,\ q_0)\ :=\ \mathcal{D}_R[statesdef]$
        $(Q_M,\ \Sigma,\ \Delta,\ I)\ :=\ \mathcal{D}_M[movesdef]$
    $\mathbf{in}$
        $(Q_R\ \cup\ Q_M,\ \Sigma,\ \Delta,\ q_0,\ \{C_{moving}\},\ I)$
```

$$\mathcal{D}_R[statesdef(s_1,\ ...,\ s_n)]\ :=\ (\{s_1,\ ...,\ s_n\},\ s_1)$$

```
$\mathcal{D}_M[movesdef((s_1,\ s'_1,\ lb_1,\ ub_1),\ ...,\ (s_n,\ s'_n,\ lb_n,\ ub_n))]\ :=$
    $\mathbf{let}$
        $Q\ :=\ \{moving(s_i, s'_i) \mid 1 \leq i \leq n\}$
        $\Sigma\ :=\ \{moveto(s'_i),\ reached(s'_i) \mid 1 \leq i \leq n\}$
        $\Delta_?\ :=\ \{(s_i,\ \langle moveto(s'_i),\ ?,\ \lambda v.true,\ \{C_{moving}\}\rangle,\ moving(s_i, s'_i)) \mid 1 \leq i \leq n\}$
        $\Delta_!\ :=\ \{(moving(s_i, s'_i),\ \langle reached(s'_i),\ !,\ \mathcal{D}_C[lowerbound(C_{moving},\ '[',\ lb_i)],\ \emptyset\rangle,\ s'_i) \mid 1 \leq i \leq n\}$
        $I\ :=\ \{moving(s_i, s'_i) \mapsto \mathcal{D}_C[upperbound(C_{moving},\ ']',\ ub_i)] \mid 1 \leq i \leq n\}$
    $\mathbf{in}$
        $(Q,\ \Sigma,\ \Delta_?\ \cup\ \Delta_!,\ I)$
```

For each declared resting state in the hardware description, a corresponding node in the target automaton is inserted. The clock C_{moving} is used to measure the time that passes during a movement between two resting states. A declaration of a movement between two resting states A and B implies a generation of

1. an intermediate moving state $moving(A, B)$ (`moving_A_B`),

2. a controllable transition $?moveto(B)$ (`?moveto_B`) from A to $moving(A, B)$, resetting the clock C_{moving}, and

3. an uncontrollable transition $!reached(B)$ (`!reached_B`) from $moving(A, B)$ to B.

Let T_{min} and T_{max} denote the lower- and upper bound of the time that takes the movement from A to B. The invariant of $moving(A, B)$ is set to $\lambda v.v(C_{moving}) \leq T_{max}$, implying that the execution may remain in $moving(A, B)$ for at most T_{max} time units. The guard of the $!reached(B)$ transition is set to $\lambda v.T_{min} \leq v(C_{moving})$, implying that this transition may only be taken after that T_{min} time units since $?moveto(B)$ have passed. So, the destination state B can only be reached if the invariant $v(C_{moving}) \leq T_{max}$ and the guard

$T_{min} \leq v(C_{moving})$ hold true:

$$\begin{aligned} & v(C_{moving}) \leq T_{max} \ \wedge \ T_{min} \leq v(C_{moving}) \\ \Leftrightarrow \quad & v(C_{moving}) \in [0, T_{max}] \ \wedge \ v(C_{moving}) \in [T_{min}, \infty) \\ \Leftrightarrow \quad & v(C_{moving}) \in [0, T_{max}] \ \cap \ [T_{min}, \infty) \\ \Leftrightarrow \quad & v(C_{moving}) \in [T_{min}, T_{max}] \end{aligned}$$

since $T_{min} \leq T_{max}$, by definition.

4.3.3 Operator Component

```
$\mathcal{D}[operatorbody(statesdef,\ movesdef)]\ :=$
  **let**
    $(Q_R,\ q_0)\ :=\ \mathcal{D}_R[statesdef]$
    $(Q_M,\ \Sigma,\ \Delta,\ I)\ :=\ \mathcal{D}_M[movesdef]$
  **in**
    $(Q_R\ \cup\ Q_M,\ \Sigma,\ \Delta,\ q_0,\ \{C_{moving}\},\ I)$

$\mathcal{D}_R[statesdef(s_1,\ ...,\ s_n)]\ :=\ (\{s_1,\ ...,\ s_n\},\ s_1)$

$\mathcal{D}_M[movesdef((s_1,\ s'_1,\ lb_1,\ ub_1),\ ...,\ (s_n,\ s'_n,\ lb_n,\ ub_n))]\ :=$
  **let**
    $Q\ :=\ \{moving(s_i, s'_i) \mid 1 \leq i \leq n\}$
    $\Sigma\ :=\ \{occurred(s'_i),\ reached(s'_i) \mid 1 \leq i \leq n\}$
    $\Delta\ :=$
        $\{(s_i,\ \langle occurred(s'_i),\ !,\ \lambda v.true,\ \{C_{moving}\}\rangle,\ moving(s_i, s'_i)) \mid 1 \leq i \leq n\} \cup$
        $\{(moving(s_i, s'_i),\ \langle reached(s'_i),\ !,\ \lambda v.0 < v(C_{moving}) < 1,\ \emptyset\rangle,\ s'_i) \mid 1 \leq i \leq n\}$
    $I\ :=\ \{moving(s_i, s'_i) \mapsto \lambda v.v(C_{moving}) < 1 \mid 1 \leq i \leq n\}$
  **in**
    $(Q,\ \Sigma,\ \Delta,\ I)$
```

4.3.4 Assertions

$A : Q^n \rightarrow \{false,\ true\}$ represents the safety predicate function to identify explicit fail states. The denotational semantics is formally defined as:

$$\begin{aligned}
\mathcal{D}[nodeconditionlist(c_1, c_2, ..., c_n)] &:= \lambda q. \bigwedge_{i=1}^{n} \mathcal{D}_A[c_i](q) \\
\mathcal{D}_A[never(x)] &:= \lambda q.\neg\mathcal{D}_A[x](q) \\
\mathcal{D}_A[always(x)] &:= \lambda q.\mathcal{D}_A[x](q) \\
\mathcal{D}_A[onlyif(x, y)] &:= \lambda q.\neg\mathcal{D}_A[x](q) \vee \mathcal{D}_A[y](q) \\
\mathcal{D}_A[or(x, y)] &:= \lambda q.\mathcal{D}_A[x](q) \vee \mathcal{D}_A[y](q) \\
\mathcal{D}_A[and(x, y)] &:= \lambda q.\mathcal{D}_A[x](q) \wedge \mathcal{D}_A[y](q) \\
\mathcal{D}_A[not(x)] &:= \lambda q.\neg\mathcal{D}_A[x](q) \\
\mathcal{D}_A[state(unit, node, substr)] &:= \lambda q.q \sim (unit, node, (substr = *)) \\
\mathcal{D}_A[constant(c)] &:= \lambda q.c
\end{aligned}$$

Where the matching operator $\sim$ is defined as follows:

$$(q_1, q_2, ..., q_n) \sim (unit, node, substr) :=$$
$$\begin{cases} true & : \quad substr \wedge \exists q_i :\ label(A_i) \subseteq unit \wedge label(q_i) \subseteq node \\ true & : \quad \neg substr \wedge \exists q_i :\ label(A_i) = unit \wedge label(q_i) = node \\ false & : \quad \text{else} \end{cases}$$

The statement $s_1 \subseteq s_2$ stands for a test, whether the string s_1 is contained in s_2 as a substring.

4.3.5 Dependencies

Let $D \subseteq \Sigma \times \Sigma$ be a dependency relation, then we say

$$\begin{aligned}
e_1 \text{ and } e_2 \text{ are } \textbf{dependent} \text{ w.r.t. D} \quad &\Leftrightarrow \quad (e_1,\ e_2) \in D \ \vee\ (e_2,\ e_1) \in D \\
e_1 \text{ and } e_2 \text{ are } \textbf{independent} \text{ w.r.t. D} \quad &\Leftrightarrow \quad e_1 \text{ and } e_2 \text{ are } \textbf{not dependent} \text{ w.r.t. D}
\end{aligned}$$

The denotational semantics of the dependency grammar is defined as:

$$\begin{aligned}
\mathcal{D}[dependencylist(d_0, d_1, ..., d_n)] &:= \bigcup_{i=0}^{n} \mathcal{D}[d_i] \\
\mathcal{D}[dependency(message, events)] &:= \{(message, e) \mid e \in events\}
\end{aligned}$$

4.3.6 State Guards

$\mathcal{G} = \{G_0, G_1, ...G_n\}$ denotes a finite set of guard functions where each $G \in \mathcal{G}$ is a function $G : Q^n \rightarrow \Sigma \cup \{\bot\}$ that represents a guard of a combined node tuple. The denotational semantics of the guard grammar is defined as:

$$\begin{aligned}
\mathcal{D}[guardlist(g_0, g_1, ..., g_n)] &:= \{\mathcal{D}_G[g_0], \mathcal{D}_G[g_1], ..., \mathcal{D}_G[g_n]\} \\
\mathcal{D}_G[guard(node, message)] &:= \lambda q.\textbf{if } \mathcal{D}_A[node](q) \textbf{ then } message \textbf{ else } \bot
\end{aligned}$$

4.3.7 Plans

The final mathematical representation of the plans is a finite set of vectors $\mathcal{P} = \{P_1,\ ...,\ P_n\}$, where each item P_i represents a single plan automaton.
The denotational semantics is defined as follows:

$$\mathcal{D}[plans(P_1,\ ...,\ P_n)]\ :=\ (\{\mathcal{D}_P[P_1],\ ...,\ \mathcal{D}_P[P_n]\})$$

$$\begin{aligned}&\mathcal{D}_P[plan(clocksdef,\ plancommandlist)]\ := \\ &\quad \mathcal{D}_C[clocksdef,\ plancommandlist]\end{aligned}$$

Where $\mathcal{D}_C[plancommandlist]$ is computed by algorithm 4.

```
Input  : Set of timeout-clocks $T = \{(X_1, T_1),\ ...,\ (X_n, T_n)\}$. List of plan
         commands, represented by a vector $C = ((l_1, C_1),\ ...,\ (l_m, C_m))$. Each
         command $(l_i, C_i)$ is a pair consisting of a label $l_i$ and the command itself
         $C_i$.
Output : Timed plan automaton $(Q,\ \Sigma,\ \Delta,\ q_0,\ failed,\ X,\ I)$ that describes the
         plan.

$(\Sigma,\ \Delta)\ \leftarrow\ (\emptyset,\ \emptyset)$
$X\ \leftarrow\ \{X_i \mid i \in \{1, ..., n\}\}$
$Q\ \leftarrow\ \{failed\} \cup \{l_i \mid i \in \{1, ..., m\}\}$
$I\ \leftarrow\ \lambda v.true$

for $i \in \{1,\ ...,\ m\}$ do
    if $C_i = reset(R)$ then
        $\Delta\ \leftarrow\ \Delta \cup \{(l_i, \langle \epsilon,\ \$,\ \lambda v.true,\ R\rangle, l_{(i+1) \mod m})\}$
    else if $C_i = waitfor((e_1, d_1), ..., (e_p, d_p))$ then
        forall $j \in \{1,\ ...,\ p\}$ do
            $\Delta\ \leftarrow\ \Delta \cup \{(l_i, \langle e_j,\ \$,\ \lambda v.true,\ \emptyset\rangle, d_j)\}$
    forall $j \in \{1,\ ...,\ n\}$ do
        $\Delta\ \leftarrow\ \Delta \cup \{(l_i, \langle \epsilon,\ \$,\ \lambda v.v(X_j) \geq T_j,\ \emptyset\rangle, failed)\}$

return $(Q,\ \Sigma,\ \Delta,\ l_1,\ failed,\ X,\ I)$
```

Algorithm 4: Generation of a plan automaton

Chapter 5

Game Solving

This chapter describes an efficient algorithm that computes the winning controller strategy as described in section 2.1.3. In combination with the discretisation ideas as shown in 2.2.2, the product state space is explored on-the-fly.

In the input specification, it is described which behaviour of the plant is forbidden. For example, the assertions describe certain combinations of component states that must be avoided. Therefore, the *combined* state space must be considered. This makes it possible that undesired state combinations can be excluded.

A plant state is the combination of the configurations of the components, the configurations of the plans, and the current clock assignment. Rather than combining the automata step by step (i.e., first A_1 with A_2, then the resulting automata with A_3 and so on...), all automata are combined at a time.

5.1 Zenoness

Recall that the basis for finding winning strategies are safety games, i.e. that within a controller winning strategy, no fail state can be reached. But what happens if there were loops in the strategy? Hereby, we distinguish between two types of loops: loops where time gradually elapses and loops where no time elapses. The latter loops are called *zeno-loops* and represent a fundamental issue in synthesis since such loops can be infinitely often executed without exceeding a time bound.

Of course, since we are using timed automata, time always elapses in the clocks. However, this is only technically true. If a clock is not used within some parts of a timed automaton, no time elapses effectively with respect to that clock. Also, if a transition in a loop resets a clock, no time elapses for that clock as well.

One way to avoid zenoness is just by not allowing zeno specifications. A precomputation step could determine if there are zeno-loops and, if so, reject the input. Another possibility is to declare zeno-loops in particular as fail states and only to avoid them in the winning strategy. This can be done by keeping all visited states since the last goal state in mind. If the plant part of a new discovered state is also contained in the already visited states, then that new found state becomes a fail state. If a goal is reached, this set of visited states is cleared again. Even though this causes a blow up in theory, practical experience has shown that this works with real world case studies. Later in section 5.3, this technique is explained in detail.

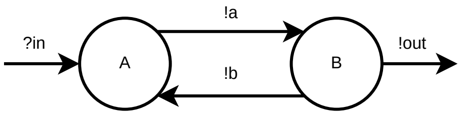


Figure 5.1: The plant can force an infinite loop

Figure 5.1 shows a simple zeno example. If the controller issues an $?in$ request, the execution comes to state A. From there, the plant may produce a spontaneous $!a$ event that brings the execution to state B. There, either an $!out$ event can occur that causes the execution to leave that zeno part, or else, a $!b$ event leads back to A. Using the zeno-loop avoiding technique as described above, one would identify the $!b$ transition as erroneous, since no plan goal has been achieved since the last visit in A. Because $!b$ can happen spontaneously, we must declare B as a fail state, which causes also A to become a fail state since there is the uncontrollable transition $!a$ leading from A to B. Thus, since A is identified as a fail state, $?in$ is discarded as a decision for the controller.

One may say that this method copes only with zeno-behaviour of the plant, while controller zenoness is not considered. This is true but it is accepted. Consider a machine that processes workpieces, which are sequentially loaded over an intake transporting belt. After the processing, the processed workpiece is unloaded and the next piece is loaded. Now, if one would specify that problem by modelling the plant in such a way that the processing consumes no time, and the plan is formulated as ”for each workpiece w: process w”, a valid program could be ”load workpiece w, do forever: process w”. Indeed, this program satisfies the specification because processing consumes no time, and hence, it can be executed infinitely often without exceeding any given upper time bound. Furthermore, this zeno-loop cannot be detected by the technique that was described above since processing is the only goal of the plan, actually. In order to avoid the generation of such senseless programs, the plan has to be formulated more precisely. For this example, an appropriate plan could be ”for each workpiece w: load w, process w, and unload w”.

5.2 Precomputation

Prior to the actual state space exploration, two precomputation steps were applied. In order to reduce the state space explosion, a clock usage analysis is done. Hereby, it is computed in which locations of a timed automaton its clocks are being actually used. For those locations where a clock c is not used, we do not need to consider any assignment that is more precisely than $0 \leq c < \infty$. This reduces the amount of distinct clock zones for that locations, and therefore, the overall amount of states.

The unique choice interval split-up of the input timed automata ensure that they do not have any implicit temporal behaviour. In [1], it was firstly explained that the usual model-checking simulation graph of a timed automaton is not suitable for synthesis, since letting time pass is neither controllable nor uncontrollable. Hence, the possibility that an uncontrollable event occurs that lies in the future becomes controllable if letting time pass is controllable as well.

5.2.1 Clock Usage Analysis

The current value of a clock c is seldom needed in each location of a timed automaton. It is straightforward to declare c as *locally* used in a location L: namely, if some execution-decisions, i.e., the invariant or the guards of the outgoing edges of L, depend on c. But when is it actually possible to declare c as *globally* unused in a certain location?

Whenever c is being resetted, its old value is lost. So, in the preceding locations before the clock reset that not locally use c, it is irrelevant what value c has (since c will be resetted anyway).

More formally, a clock constraint $\varphi : (X \to \mathbb{R}_0^+) \to \{false,\ true\}$ depends on a clock $c \in X$ if $\exists v(c) \in \mathbb{R}_0^+ \ : \ \varphi(v) = false$. In this case, $c \in \varphi$ denotes this dependence. The set of used locations $R(c)$ is computed by a backward least fixed-point computation starting at those locations that uses c locally, $R_0(c)$:

$$R_0(c) := \{s \in Q \mid c \in I(s) \ \vee \ \exists (s, \langle \alpha, \tau, \varphi, \lambda \rangle, t) \in \Delta \ : \ c \in \varphi\}$$

$$\mu R(c)\Big(R_0(c) \ \cup \ \{s \in Q \mid \exists (s, \langle \alpha, \tau, \varphi, \lambda \rangle, t) \in \Delta \ : \ t \in R(c) \ \wedge \ c \notin \lambda\}\Big)$$

A clock c is **unused** in a location L iff $L \notin R(c)$.

In figure 5.2, an example clock usage computation is showed. The node denoted with R_0 represents the location where the clock C is locally used. The nodes labelled with R show the locations where c is (transitively) used.

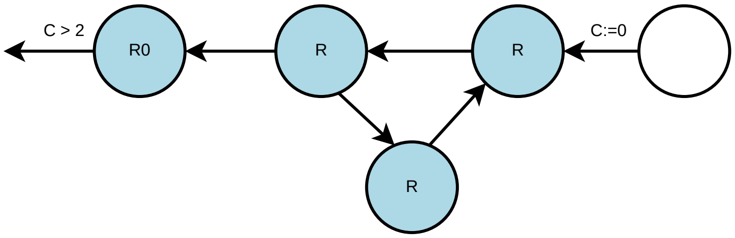


Figure 5.2: Clock usage example

5.2.2 Unique Choice Intervals

A Timed Automaton may contain implicit temporal behaviour. Consider a location L with invariant $0 \leq c \leq 4$, an outgoing edge $?a$, and an outgoing edge $!b$ that is guarded by the constraint $c > 2$. When the execution is at L and $c = 0$, then $?a$ can be taken instantly while $!b$ becomes firstly possible if $c > 2$.

Now, it might turn out that $!b$ leads to some fail state. If L would be treated as one whole location, it would have to be removed in order to avoid the occurrence of $!b$ at all. But this would be inappropriate because the controller can issue an $?a$ request as long as $c \leq 2$ and prevent the occurrence of $!b$ either. So it gets clear why a precomputation is needed that splits the locations with implicit decisions into explicit ones.

Formally speaking, the set of the decision-intervals $D_c \subseteq 2^{2^{\mathbb{R}_0^+}}$ of a clock c of a timed automaton induce a set of *unique choice intervals* $U(c)$ such that

$$\dot{\bigcup_{u \in U(c)}} u = [0, \infty)$$

This *closure-property* means that the unique choice intervals do not overlap and range over the complete domain of clock-values. D_c is computed by collecting all constraints of the clock c in the invariants and guards of A. The following computational steps describe how to obtain a valid partition such that the closure property is satisfied.
First of all, the bounds of the decision intervals are indexed and collected into an ordered set $B_0 \subset \mathbb{N} \cup \{\infty\}$:

$$B_0 := \bigcup_{d \in D_c} \{l(d)\} \cup \{u(d)\}$$

where

$$l(d) := \begin{cases} 4\inf(d) & : \inf(d) = \min(d) \\ 4\inf(d) + 2 & : else \end{cases}$$

and

$$u(d) := \begin{cases} \infty & : \nexists \sup(d) \\ 4\sup(d) + 1 & : \sup(d) = \max(d) \\ 4\sup(d) - 1 & : else \end{cases}$$

Note that the lower bound-indexes are always even, while the upper bound-indexes are always odd. The case $\inf(d) = \min(d)$ (or $\sup(d) = \max(d)$, respectively) stands for a closed interval bound, i.e. that the bound lies within the interval. Based on B_0, the index-set $B = (b_0, b_1, ..., b_n)$ of $U(c)$ is computed as follows:

$$\begin{array}{llll} B := B_0 & \cup & \{b_{i+1} - 1 & \mid \forall b_i, b_{i+1} \in B_0 \, : \, b_i \mod 2 \equiv 0 \, \wedge \, b_{i+1} \mod 2 \equiv 0\} \\ & \cup & \{b_i + 1, \; b_{i+1} - 1 & \mid \forall b_i, b_{i+1} \in B_0 \, : \, b_i \mod 2 \equiv 1 \, \wedge \, b_{i+1} \mod 2 \equiv 0\} \\ & \cup & \{b_i + 1 & \mid \forall b_i, b_{i+1} \in B_0 \, : \, b_i \mod 2 \equiv 1 \, \wedge \, b_{i+1} \mod 2 \equiv 1\} \end{array}$$

For convenience, it is defined that $\infty \mod 2 \equiv 1$. It is obvious to see that B contains only sequences of bound-indexes of the form

$$\forall b_{2i}, \; b_{2i+1} \in B \; : \; b_{2i} \mod 2 \equiv 0 \quad \wedge \quad b_{2i+1} \mod 2 \equiv 1$$

Therefore, the $(b_{2i}, \; b_{2i+1})$-pairs represent the unique choice intervals of $U(c)$:

$$U(c) := \{l^{-1}(b_{2i}) \; \cap \; u^{-1}(b_{2i+1}) \mid i \in \{0...n \div 2 - 1\}\}$$

where

$$l^{-1}(i) := \begin{cases} \{x \in \mathbb{R}_0^+ \mid x \geq i \div 4\} & : \quad i \mod 4 \equiv 0 \\ \{x \in \mathbb{R}_0^+ \mid x > i \div 4\} & : \quad else \end{cases}$$

and

$$u^{-1}(i) := \begin{cases} \mathbb{R}_0^+ & : \quad i = \infty \\ \{x \in \mathbb{R}_0^+ \mid x \leq (i+1) \div 4\} & : \quad (i+1) \mod 4 \equiv 2 \\ \{x \in \mathbb{R}_0^+ \mid x < (i+1) \div 4\} & : \quad else \end{cases}$$

Note that $\div$ stands for integer-division.

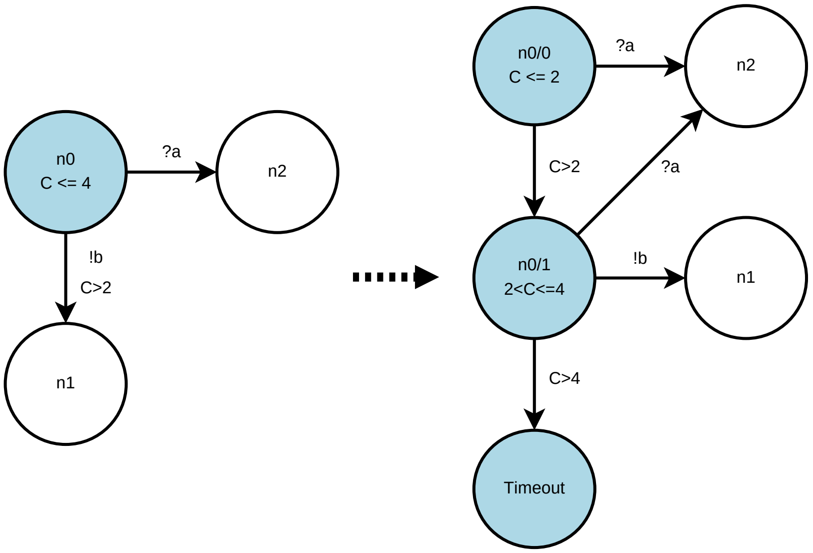


Figure 5.3: Unique choice split-up

In the overall synthesis process, the unique choice split-up takes place after the clock usage analysis and prior to the state space exploration. It must be done for each plant component, each clock, and for each location. Of course, only these locations where the clocks are actually used must be considered for split up. Figure 5.3 shows the split-up for the example that was described above. For each choice interval, a new sub-location is added. These new sub-locations are linked by explicit *delay-edges*. Formally speaking, the sub-location for the interval u_i is linked with the succeeding sub-location for u_{i+1} with the event $\langle\, \varepsilon,\ D,\ \lambda v.v(c) \in u_{i+1},\ \emptyset\, \rangle$.

Let I be the value-interval of the invariant of a location L that has to be split up. If $\sup(I) < \infty$, i.e. that the invariant has an upper bound, then the closure-property would not be satisfied, since $\forall x > \sup(I) \ : \ \nexists u \in U(c) \ : \ x \in u$. Indeed, if an invariant has an upper bound for a clock c, then this means that it was specified that it is impossible that c exceeds this limit. But on the other hand, it is also impossible that every exceptional behaviour is covered by the specifications.

Imagine a location that models a machine process that takes some time. This ”being-in-process” location A would have a guarded outgoing edge leading to some ”process-done” location B. While the guard of this edge models the lower bound of the process-time $c \geq min$, the invariant of A ensures that the execution remains in A only until the upper bound of the process-time has not yet exceeded $c \leq max$. Most of the running-time of this machine, the process actually takes that specified amount of time. But in the (seldom) case that something goes wrong (e.g. due to malfunctioning hardware), then c would advance beyond the upper time bound.

Of course, one could model such an exceptional behaviour manually additional to each ”being-in-process” location. But this would, on the one hand, imply more complexity for the programmer (modeller), and on the other hand, modelling such exceptions as uncontrollable reactions that lead to a fail state would be inappropriate because all those ”being-in-process” locations would not be part of the controller winning strategy since it might happen that this exception holds, sometimes.

So it gets clear why another type of exceptional delay-edge is needed; a delay-edge that leads to a fail state (explicit or timeout) becomes a *timeout-edge*. A location can be part of the winning strategy despite the fact that it has (uncontrollable) timeout-edges. Automatically, timeout-edges can be generated by inverting $U(c)$:
Let $T \subseteq \mathbb{R}_0^+$ be the timeout interval of a set of unique choice intervals $U(c)$, then

$$T := \mathbb{R}_0^+ \setminus \bigcup_{u \in U(c)} u$$

If $T \neq \emptyset$, then a timeout-edge is generated from the sub-locations, representing the last unique choice interval to a dedicated timeout-location with the event $\langle\, \varepsilon,\ T,\ \lambda v.v(c) \in T,\ \emptyset\,\rangle$.

Later, in the final synthesised program, these timeout-edges are treated as exceptions. Comparing with the ”fleeing-robber”-analogon, timeout-edges can be seen as trapdoors in the corridors: after some time, they open spontaneously and let the robber fall through such that he gets into the floor that lies right under the current one (the sub-location with the succeeding choice interval). So falling downwards corresponds to advancing in time. The robber can take a lift (that can move only upwards) to return to the higher-level floors again. This corresponds to the resetting of some clocks.

5.3 Winning Controller State Space

The winning controller state space is the basis for any synthesised program. It contains all winning strategies of the controller such that no decision may lead to a state, in which the plant can produce an uncontrollable event that brings the machine into an undesired state. As already mentioned in 2.1.2, the computational model is a two-player safety game. Computing all reachable good states in a first pass and then, reversely, removing all fail states from that set in a second pass would be, because of the giant state space, inappropriate. That is why instead of that, a combined forward/backward algorithm is used to find the winning controller state space. Since each distinct state is visited at most twice (discovery and removal), the complexity of the algorithm is linear to the size of the combined state space.

5.3.1 Basic Functions and Operators

In the following, let $q = (q_1, q_2, ..., q_n)$ be a tuple of n locations such that each q_i belongs to an timed automaton $A_i = (Q_i, \Sigma_i, \Delta_i, root_i, X_i, I_i)$.
The function $M : Q^* \rightarrow 2^{\Sigma \times \{?,!\}}$ returns the set of all controllable and uncontrollable messages of q:

$$M(q) := \bigcup_{i=1}^{n} \{(\alpha, \tau) \mid (x, \langle \alpha, \tau, \varphi, \lambda \rangle, y) \in \Delta_i \ \wedge \ x = q_i \ \wedge \ \tau \in \{!, ?\}\}$$

The function $D : Q^* \rightarrow 2^{C(X) \times \{D,T\}}$ returns the set of all delay and timeout-events of q:

$$D(q) := \bigcup_{i=1}^{n} \{(\varphi, \tau) \mid (x, \langle \alpha, \tau, \varphi, \lambda \rangle, y) \in \Delta_i \ \wedge \ x = q_i \ \wedge \ \tau \in \{D, T\}\}$$

The operator $\sim$ returns $true$ if two messages are synchronised.

$$(\alpha_1, \tau_1) \sim (\alpha_2, \tau_2) := \begin{cases} true & : \quad \alpha_1 \notin \Sigma \wedge \alpha_2 \notin \Sigma \ \wedge \ com(\alpha_1, \alpha_2) \neq \emptyset \\ true & : \quad ((\tau_1 = \$ \ \wedge \ \tau_2 \neq \$) \ \vee \ (\tau_1 \neq \$ \ \wedge \ \tau_2 = \$)) \ \wedge \ (\alpha_1 = \alpha_2) \\ \mathit{false} & : \quad \text{else} \end{cases}$$

Recall that if an α is not in Σ, then this α is a clock-zone, representing the condition for an explicit delay or timeout-edge. $com(\alpha_1, \alpha_2)$ yields the set of commonly used clocks in both clock-zones α_1 and α_2. Note that $\sim$ is commutative:

$$(\alpha_1, \tau_1) \sim (\alpha_2, \tau_2) = (\alpha_2, \tau_2) \sim (\alpha_1, \tau_1)$$

The function Λ returns the set of all clocks that are being simultaneously resetted among the locations in q when a given event is executed.

$$\Lambda(q, (\alpha, \tau)) := \bigcup_{i=1}^{n} \lambda \ : \ (x, \langle \alpha', \tau', \varphi, \lambda \rangle, y) \in \Delta_i \ \wedge \ x = q_i \ \wedge \ (\alpha, \tau) \sim (\alpha', \tau')$$

The function $sink$ returns $true$ if the given state s has only outgoing timeout or no edges at all in the given Δ.

$$sink(s, \Delta) := \begin{cases} false & : \quad \exists (x, \langle \alpha, \tau, \lambda \rangle, y) \in \Delta : \ x = s \ \wedge \ \tau \neq T \\ true & : \quad \text{else} \end{cases}$$

The set of all applicable events to q is computed by

$$\Phi(q) := \{\langle \alpha, \tau, \Lambda(q, (\alpha, \tau)) \rangle \mid (\alpha, \tau) \in M(q) \cup D(q)\}$$

The function $apply$ applies a given event to a given discrete combined state such that the components of the state are updated accordingly.

$$apply((q, z), \langle \alpha, \tau, \lambda \rangle) := \begin{cases} (q', z[\lambda \leftarrow 0]) & : \quad \alpha \in \Sigma \\ (q, (z^{\Uparrow} \cap \alpha)[\lambda \leftarrow 0]) & : \quad \text{else} \end{cases}$$

where $q' = (q_1', q_2', ..., q_n')$ such that

$$q_i' := \begin{cases} y & : \quad \exists (x, \langle \alpha, \tau, \varphi, \lambda \rangle, y) \in \Delta_i : \ x = q_i \ \wedge \ (\alpha, \tau) \in M(q) \\ q_i & : \quad \text{else} \end{cases}$$

5.3.2 Forward Exploration

The forward exploration algorithm (5) explores the product state space of the plant and the plan automata by computing the least fixed-point of winning combined configurations (states) that are reachable from the root nodes of the plant and plan automata. Whenever this exploration reaches a fail state, the reverse state removal algorithm is called and all configurations that lead to this found fail state are removed from the state space, which was explored so far.

The traversal order of the exploration algorithm is *Breadth-First Search* (BFS). This means that, starting at the root state, all neighbouring states that are not yet explored are pushed in a FIFO-queue. In the next loop cycle, the exploration continues at those new found states. The loop ends when there are no new states in the search queue.

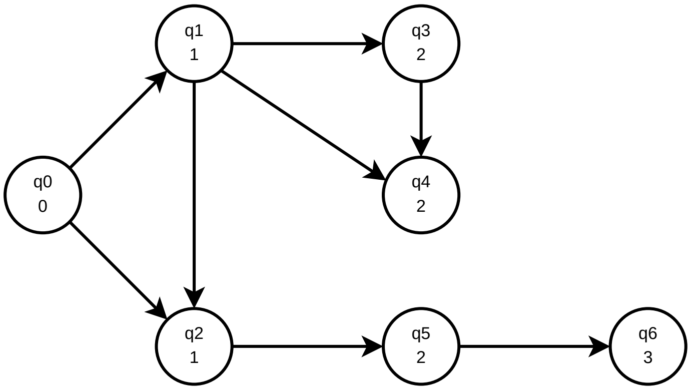


Figure 5.4: Breadth-First Search

Figure 5.4 shows the BFS traversal order with a simple example graph. The upper label of the nodes are the actual node names, while the lower labels denote the number of iterations when the nodes are (firstly) discovered. Trivially, the root node q_0 is explored after 0 iterations. After 1 iteration, q_1 and q_2 are found. Then, after 2 iterations, q_3, q_4, and q_5 are found. And finally, after 3 iterations, q_6 is found.

Let $q = (e_1, ..., e_n,\ p_1, ..., p_m)$, where each e_i is a node of a plant automaton and each p_j is a node of a plan automaton. Then, the two functions $env : (Q^E_1 \times ... \times Q^E_n \times Q^P_1 \times ... \times Q^P_m) \rightarrow (Q^E_1 \times ... \times Q^E_n)$ and $plan : (Q^E_1 \times ... \times Q^E_n \times Q^P_1 \times ... \times Q^P_m) \rightarrow (Q^P_1 \times ... \times Q^P_m)$ are defined as follows:

$$env((e_1, ..., e_n,\ p_1, ..., p_m)) \ := \ (e_1, ..., e_n)$$

$$plan((e_1, ..., e_n,\ p_1, ..., p_m)) \ := \ (p_1, ..., p_m)$$

$$env(q, z) \ := \ env(q)$$

$$plan(q, z) \ := \ plan(q)$$

Input : Basis containing plant automata $\{E_1, E_2, ..., E_n\}$ and plan automata $\{P_1, P_2, ..., P_m\}$. Each E_i is a timed automaton with $E_i = (Q_i^E,\ \Sigma_i^E,\ \Delta_i^E,\ root_i^E,\ X_i^E,\ I_i^E)$ and each P_i is a timed plan automaton with $P_i = (Q_i^P,\ \Sigma_i^P,\ \Delta_i^P,\ root_i^P,\ failed_i^P,\ X_i^P,\ I_i^P)$. Safety predicate $A : Q^* \rightarrow \{false,\ true\}$. Set of state guards $\mathcal{G} = \{G_1, G_2, ...\}$, where each G_i is a function $G_i : Q^* \rightarrow \Sigma^G$ that associates combined states with a guard, namely an uncontrollable event.

Output: Controller winning state space (S, Σ, Δ, s_0)

```
$root \leftarrow (root_1^E,\ root_2^E,\ ...,\ root_n^E,\ root_1^P,\ root_2^P,\ ...,\ root_m^P)$
$(S,\ \Sigma,\ \Delta,\ s_0) \leftarrow (\{(root,\ Z_0)\},\ \bigcup_{i=1}^n \Sigma_i^E \cup \bigcup_{i=1}^m \Sigma_i^P \cup \Sigma^G,\ \emptyset,\ (root,\ Z_0))$
$fail \leftarrow \emptyset$
$L \leftarrow \{(s_0, \{env(root)\})\}$
while $L \neq \emptyset$ do
    $L' \leftarrow \emptyset$
    forall $(q, z), vis) \in \{((q', z'), vis') \in L \mid (q', z') \notin fail\}$ do
        forall $e = \langle \alpha, \tau, \lambda \rangle \in \Phi(q)$ do
            $(new = (q^{new}, z^{new})) \leftarrow apply((q, z), e)$
            if $new$ is valid then
                if $new \in S$ then
                    $\Delta \leftarrow \Delta \cup \{((q, z), e, new)\}$
                else
                    $S \leftarrow S \cup \{new\}$
                    if $\exists G \in \mathcal{G} : G(env(q^{new})) = g$ then
                        $q^g \leftarrow$ createNewState()
                        $S \leftarrow S \cup \{(q^g, z)\}$
                        $\Delta \leftarrow \Delta \cup \{((q, z), \langle g, \%, \emptyset \rangle, (q^g, z))\}$
                        $\Delta \leftarrow \Delta \cup \{((q^g, z), e, new)\}$
                    else
                        $\Delta \leftarrow \Delta \cup \{((q, z), e, new)\}$
                    $vis' \leftarrow \begin{cases} vis \cup \{env(q^{new})\} & : \quad plan(q) = plan(q^{new}) \\ \{env(q^{new})\} & : \quad else \end{cases}$
                    $L' \leftarrow L' \cup \{(new, vis')\}$
            else if $\tau = !$ then
                $(S, \Delta, fail) \leftarrow$ removeReverse($S, \Delta, fail, (q, z)$)
                continue with next item in $L$
            else if $\tau = D$ then
                $\Delta \leftarrow \Delta \cup \{((q, z), \langle \alpha, T, \lambda \rangle, (q^g, z))\}$
        if isSink($(q, z), \Delta$) then
            $(S, \Delta, fail) \leftarrow$ removeReverse($S, \Delta, fail, (q, z)$)
    $L \leftarrow L'$
return $(S, \Sigma, \Delta, s_0)$
```

Algorithm 5: State space exploration main algorithm

The root node of the combined state space is the combination of all plant root nodes $root_i^E$, all plan root nodes $root_j^P$, and the zero clock zone Z_0 (lines 1-2). The message alphabet Σ of the state space is the union of the message alphabets of all plant automata Σ_i^E, all plan automata Σ_j^P, and the set of the guard messages Σ^G (line 2). The set of found fail states $fail$ is initialised as the empty set (line 3). The set L represents the BFS search-queue. The search-items that are stored in L are tuples, consisting of a new explored state and a set of already visited combined plant nodes. L is initialised with the root state s_0 and the plant part of the root node as the only item of the already visited set (line 4).

The main loop runs until no new states were added anymore, i.e. that L is empty (line 5). For each new found state in line 7, all neighbouring states $new = (q^{new}, z^{new})$ are considered that are reachable by applying all possible applicable events (line 9). The functions Φ, as described in section 5.3.1, return the set of all applicable message- and delay-events. The function $apply$ returns the new combined state that evolves from the origin state when applying an event. In line 10, the new found combined state new is valid if all of the following conditions hold true:

1. $new \notin fail$
2. $A(env(new)) = true$
3. $plan(q) \neq plan(q^{new}) \ \vee \ env(q^{new}) \notin vis$

The first condition just means that the new state must not be part of the set of the already known fail states. Condition 2 ensures that the new found state must be a safe one (i.e., no explicit specified fail state). Condition 3 is a bit more sophisticated; it says that redundant movements of the plant are not allowed without any progress of the plans. It gets clearer if one negates condition 3: a state is *invalid* if

$$\begin{array}{rl} & \neg(plan(q) \neq plan(q^{new}) \vee env(q^{new}) \notin vis) \\ \equiv & plan(q) = plan(q^{new}) \wedge env(q^{new}) \in vis \end{array}$$

So a state is actually invalid if we reach an already visited combined plant state without that any plan state has changed meanwhile. This is the realisation of the zeno-avoidance technique as sketched in 5.1.

Figure 5.5 shows a simple combined state space, consisting of some plant states (e_0, e_1,...), plan states (p_0, p_1), and clock zones (z_0, z_1). The initial state is $(e_0,\ p_0,\ z_0)$. From there, the controller can make an $?a$ transition to get to $(e_1,\ p_0,\ z_0)$. Then, time can elapse via the $^\circ D$ transition, which leads to $(e_1,\ p_0,\ z_1)$. A spontaneous reaction $!b$ may follow that brings the plant to e_2 and the current combined state to $(e_2,\ p_0,\ z_1)$. Because the current goal of the plan does not care about $?a$ or $!b$, the plan component of the combined state remains at p_0. Thus, the controller must not issue another $?a$ request from e_2 back to e_1, since this action would not bring a progress w.r.t. the plan. Instead, it turns out that the controller has to wait until $!c$ occurs, which is the current goal. Now from $(e_3,\ p_1,\ z_1)$, the controller can execute $?a$ and return to e_1, since a goal was reached.

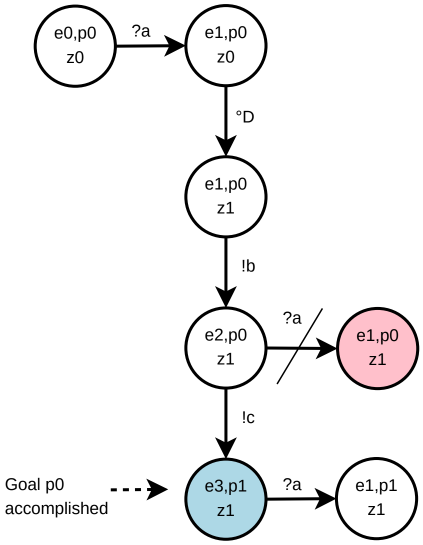


Figure 5.5: Forward exploration of reachable states

If new is actually valid, then it is checked if it was already explored in the past (line 11). Is this also true, only a new transition is added (line 12). Otherwise, if it is a new state, then it is added to the set of states S (line 14) and a new search-item (new, vis') is added to L' (line 23). If a plan has reached its next goal, vis is being resetted (line 22). Otherwise, if no goal is reached, the plant part of q^{new} is added to the set of already visited plant states of the search item. If there exists a guard associated with q^{new}, an intermediate state q_g is additionally generated that ensures that new is entered only if the guard-condition allows that (lines 16-19).

If new is not valid and e is controllable, then we can avoid new just by discarding e. But if e is uncontrollable, then the origin state (q, z) is also invalid since we can get from there to new spontaneously. In order to remove every other state, which may also lead to (q, z), the reverse state removal algorithm is called with origin (q, z) (line 25). After that, since (q, z) is no longer part of the state space, no other events applicable to (q, z) have to be considered yet and we may continue with the next search item in L (line 26). In the case that new is not valid and e is a delay-edge, then we add a timeout-edge with the same clock constraint instead of e (line 28).

In case that (q, z) is a sink, i.e. that there are no or only timeout-edges going out from (q, z), then (q, z) becomes invalid as well since no goals can be reached anymore from that state (line 29). Consequently, the reverse state removal algorithm must be called again (line 30).

5.3.3 Reverse Fail State Removal

Once an invalid state has been discovered in the forward exploration, all states that may lead to this invalid state must be removed from the so far explored state space. In order to avoid all invalid states in the future, they are also added to the set of already known fail states.

Algorithm 6 represents the function $removeReverse(S,\ \Delta,\ fail,\ origin)$. It backtracks on all transitions in the transition system $(S,\ \Delta)$ that lead to $origin$ and removes them.

```
Input : So far explored state space $S$ with transitions $\Delta$. Node $origin$, identifying
        the origin of the removal process. Set of fail states $fail$.
Output: Triple $(S', \Delta', fail')$ containing the adjusted state space with transitions
        and the supplemented set of fail states.

1  $(S', \Delta, fail') \leftarrow (S, \Delta, fail \cup \{origin\})$
2  $L \leftarrow \{origin\}$
3  while $L \neq \emptyset$ do
4      $L' \leftarrow \emptyset$
5      forall $s \in L$ do
6          $fail' \leftarrow fail' \cup \{s\}$
7          forall $\delta = (x, \langle\alpha, \tau, \lambda\rangle, y) \in \{(x', e', y') \in \Delta \mid x' \notin fail' \wedge y' = s\}$ do
8              switch $\tau$ do
9                  case !
10                     $L' \leftarrow L' \cup \{x\}$
11                 case ? or %
12                     $\Delta' \leftarrow \Delta' \setminus \delta$
13                     if $sink(x, \Delta')$ then
14                         $L' \leftarrow L' \cup \{x\}$
15                 case $\circ$
16                     $\Delta' \leftarrow \Delta' \cup \{(x, \langle\alpha,\ , \lambda\rangle, timeout\}$
17                     if $sink(x, \Delta')$ then
18                         $L' \leftarrow L' \cup \{x\}$
19         $\Delta' \leftarrow \Delta' \setminus \{(x', e', y') \in \Delta' \mid x' = s\}$
20         $S' \leftarrow S' \setminus \{s\}$
21     $L \leftarrow L'$
22 return $(S', \Delta', fail')$
```

Algorithm 6: Reverse fail state removal

In line 3, the iteration runs until the least fixed-point of invalid states leading to $origin$ is found. Our interest lies only in the transitions between a non fail state x and a fail state y (line 7). We call those *fail transitions*. If such a fail transition δ is uncontrollable (lines 9-10), then the origin state x is identified as a further fail state. This is caused by the fact that a plant in state x may spontaneously execute δ and the controller cannot do anything to prevent that. So x must be already avoided. If δ is controllable or a guarded transition (lines 11-14), then x is not necessarily also a fail state. Initially, it is sufficient to remove only δ from the state space (line 12). Doing so, the controller will not be able to execute that transition. If there were no further transitions (other than timeout-edges) starting from x, then x has now become a sink and must also be removed since no goals can be reached from x anymore (lines 13-14).

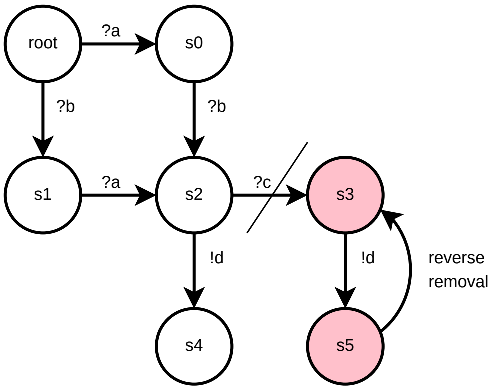


Figure 5.6: Reverse state removal

Figure 5.6 shows an example reverse fail state removal. Starting at $root$, either $?a$ before $?b$ or, vice versa, first $?b$ then $?a$ can be executed. Both alternatives lead to the state s_2. Now, the controller can execute c to get to state s_3. In s_3, only the spontaneous reaction $!d$ can occur, that lead to state s_5. It turns out that s_5 is a fail state. So, the reverse state removal is called with origin s_5. Since the exploration got from s_3 to s_5 by the uncontrollable transition $!d$, s_3 must also be removed. Furthermore, s_3 was reached from s_2 by executing the controllable transition $?c$. Therefore, the option of executing $?c$ in s_2 must be removed. Since s_2 has yet another outgoing transition $!d$ to s_4, we can stop removing here.

5.3.4 Special ”Guarded State” Transition Type

In line 18 of algorithm 5, it would be inappropriate to declare the new guard-transition from (q, z) to (q^g, z) as uncontrollable. Doing so, wrong fail states would be identified. Consider the following problem: One can get from a state A to a state C by executing a request $?y$. In the specification, a guard is declared that matches C. It says that C can only be entered if it is ensured that a condition x holds true. Then, in the final state space, a transition from A to B having $!x$ and a transition from B to C having $?y$ are added. If it turns out later that C is a fail state, the following objects will be removed:

1. the state C,

2. the $?y$ transition from B to C,

3. the guard state B (since B is now a sink), and finally

4. the state A (since the uncontrollable $!x$ transition leads from A to B)

Without protecting C by the guard x, only a transition from A to C carrying $?y$ would be added. Now, if C is identified as a fail state, only the state C and the $?y$ transition are removed but not the state A (unless A becomes a sink, of course). So it is obvious that a special transition type for guards is needed. Figure 5.7 shows an illustration.

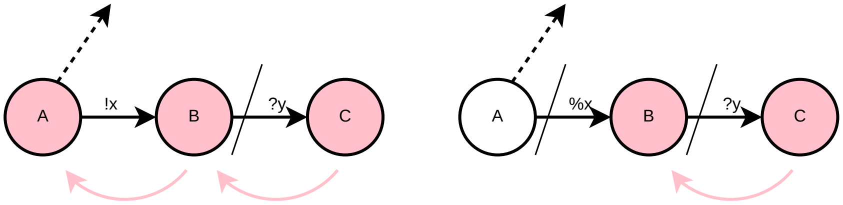


Figure 5.7: !- vs. %-state-guards

Chapter 6

Code Generation

In this chapter, it will be shown how a generation of a controller program can be done from a winning controller state space.

6.1 Intermediate Controller Language

Before any precise controller code of a real-world PLC system is compiled, an intermediate controller program is generated first. Later in section 6.7, it is shown how this code can be compiled into concrete assembler code.

$$\begin{array}{rcl}
\mathit{program} & ::= & \mathit{command} \\
\mathit{command} & ::= & \mathit{label}\ '\!:\!'\ \mathit{instruction}\ [\ \mathit{resetclocks}\ ]\ ';' \\
\mathit{instruction} & ::= & (\ \mathit{do} \mid \mathit{goto} \mid \mathit{if} \mid \mathit{waituntil}\ ) \\
\mathit{do} & ::= & \mathbf{do}\ '('\ \mathit{outmessage}\ ')' \\
\mathit{goto} & ::= & \mathbf{goto}\ \mathit{label} \\
\mathit{if} & ::= & \mathbf{if}\ '('\ \mathit{condition}\ ')'\ \mathbf{then\ goto}\ \mathit{label} \\
\mathit{waituntil} & ::= & \mathbf{waituntil}\ '('\ \mathit{condition}\ ')' \\
\mathit{condition} & ::= & \mathit{inmessage} \mid \mathit{delay} \\
\mathit{delay} & ::= & \mathit{clocklist}\ \mathbf{wait}\ \mathit{constant} \\
\mathit{resetclocks} & ::= & \mathbf{and\ reset\ clocks}\ \mathit{clocklist} \\
\mathit{clocklist} & ::= & '\{'\ \mathit{clock}\ \{','\ \mathit{clock}\ \}\ '\}'
\end{array}$$

A controller program consists of a sequence of commands. Each command must have a label (line number) and an instruction. Optionally, commands may also have a sequence of reset clocks. There are four instructions:

1. `DO (x)` Execute the action `x` / send a request `?x` to the plant.
2. `GOTO l` Unconditional jump to `l` / continue program execution at the command with label `l`.

3. `IF (x) THEN GOTO l` Conditional jump to `l` / if `x` is $true$, continue program execution at `l` or else continue at the succeeding line.
4. `WAITUNTIL (x)` Halt program execution until `x` becomes $true$. Then continue at the succeeding line.

Indeed, practical experience has shown that the four basic instructions DO, $GOTO$, IF, and $WAITUNTIL$ are sufficient to model a wide range of controller programs. A condition that is queried in the instructions IF and $WAITUNTIL$ may either be an uncontrollable message, sent by the plant or a clock-constraint.

Semantics

A controller program is defined as a vector of controller commands $P = (C_1, C_2, ..., C_n)$. Each command is a triple $C \subseteq L \times S \times 2^X$ containing a label (line number), an instruction, and a set of clocks that should be resetted when executing that command. The set L must provide a partial ordering and support the following operation:

$$\forall l_1, l_2 \in L : l_1 < l_2 \; \exists l' \in L : l_1 < l' < l_2$$

In other words, L has to be **dense**. The symbol L_0 denotes the label of the first command in a program. There are four instructions:

$$S := \{DO(action), IF(cond, then), GOTO(dest), WAITUNTIL(cond)\}$$

The semantics is defined by the function

$$\mathcal{E} : (\Sigma \rightarrow \{false,\ true\}) \times (\Sigma \rightarrow \perp) \times (2^X \rightarrow \perp) \times L \times I \times L \rightarrow L$$

$\mathcal{E}(in, out, reset, l_1, s, l_2, \lambda)$ returns the label of the command that should be executed next by the controller where (l_1, s) is the current command and l_2 is the label of the subsequent command. The function in corresponds to the input-layer of a controller. It returns $true$ if the given event was sent by the plant and can be received by the controller. The function out is purely imperative, which means that it returns nothing. It corresponds to the output-layer of a controller that takes an event and sends it to the plant. The function $reset$ represents an abstraction of the timer control: $reset(\lambda)$ resets the clocks/timers in λ.

The precise denotational semantics is defined as follows:

$$
\begin{array}{rcl}
\mathcal{E}(in, out, reset, l_1, DO(action), l_2, \lambda) & := & \textbf{let } out(action);\ reset(\lambda) \textbf{ in } l_2 \\
\mathcal{E}(in, out, reset, l_1, IF(cond, dest), l_2, \lambda) & := & \textbf{if } in(cond) \textbf{ then} \\
 & & \quad \textbf{let } reset(\lambda) \textbf{ in } dest \\
 & & \textbf{else} \\
 & & \quad l_2 \\
\mathcal{E}(in, out, reset, l_1, GOTO(dest), l_2, \lambda) & := & \textbf{let } reset(\lambda) \textbf{ in } dest \\
\mathcal{E}(in, out, reset, l_1, WAITUNTIL(cond), l_2, \lambda) & := & \textbf{if } in(cond) \textbf{ then} \\
 & & \quad \textbf{let } reset(\lambda) \textbf{ in } l_2 \\
 & & \textbf{else} \\
 & & \quad l_1
\end{array}
$$

A real controller works as shown in algorithm 7.

```
Input  : Controller program $P = (C_1, C_2, ..., C_n)$. Input/Output functions $in$ / $out$.
         Timer control function $reset$.
Output: Nothing, since this algorithm never ends.
$l \leftarrow L_0$
while $true$ do
    $(l_1, s_1, \lambda_1) \leftarrow P[l]$
    $(l_2, s_2, \lambda_2) \leftarrow succ(P[l])$
    $l \leftarrow \mathcal{E}(in, out, reset, l_1, s_1, l_2, \lambda_1)$
```

Algorithm 7: Principle of a real controller

Note that a single $WAITUNTIL$ instruction can also be expressed by an IF- and a $GOTO$-instruction. More precisely, this:

```
L1: WAITUNTIL (x);
L2: (succeeding command);
```

is semantically equivalent to that:

```
L1: IF (x) THEN GOTO L3;
L2: GOTO L1;
L3: (succeeding command);
```

However, the $WAITUNTIL$ instruction is useful for post-optimisations, as we see later in section 6.6. Furthermore, many industrial controllers have an analogous instruction, which makes the final compilation easier.

Example code

Consider two hardware components H_1 and H_2. H_1 can move between the two resting states A and B, while H_2 can move between X and Y. It is possible that both components can move at the same time. Bringing the machine in a state where H_1 is at B and H_2 is at Y, it is more efficient to move H_1 and H_2 simultaneously than moving them sequentially.
The following controller code will do this movement:

```
0:  DO (moveto_B)
1:  DO (moveto_Y)
2:  IF (reached_B) THEN GOTO 5
3:  IF (reached_Y) THEN GOTO 7
4:  GOTO 2
5:  WAITUNTIL (reached_Y)
6:  GOTO 8
7:  WAITUNTIL (reached_B)
8:  ...
```

In the lines 0 and 1, the controller sends the two requests `moveto_B` and `moveto_Y`. Now, the two components H_1 and H_2 will start moving. The lines 2-4 represent a wait-block, that is, a sequence of IF instructions with a succeeding $GOTO$ instruction jumping to the beginning of that block. If `reached_B` occurs before `reached_Y`, then the controller will jump to line 5, where it waits for `reached_Y`. If `reached_Y` is received first, then the controller will jump to line 7, waiting for `reached_B`. Figure 6.1 shows a flow-chart of this controller program.

6.2 Basic Functions and Operators

Firstly, in this section, some basic functions and operators are defined that makes the succeeding code generation and -optimisation algorithms more compact and easier to understand.
The following access operations on controller programs are defined:
Let $P = (..., (l_{i-1}, s_{i-1}, \lambda_{i-1}), (l_i, s_i, \lambda_i), (l_{i+1}, s_{i+1}, \lambda_{i+1}), ...)$ be a controller program, represented by a vector over triples consisting of a label, instruction, and reset-clocks. Furthermore, let $l_{i-1} < l_i < l_{i+1}$, then

$$\begin{aligned} P[l_i] &:= (l_i, s_i, \lambda_i) \\ P[l_i++] &:= (l_{i+1}, s_{i+1}, \lambda_{i+1}) \\ P[l_i--] &:= (l_{i-1}, s_{i-1}, \lambda_{i-1}) \end{aligned}$$

The inserting operator on controller programs is defined as follows:
Let $P = (..., (l_{i-1}, s_{i-1}, \lambda_{i-1}), (l_{i+1}, s_{i+1}, \lambda_{i+1}), ...)$ and $l_{i-1} < l_i < l_{i+1}$, then

$$P[l_i] \leftarrow (s_i, \lambda) \quad := \quad (..., (l_{i-1}, s_{i-1}, \lambda_{i-1}), (l_i, s_i, \lambda_i), (l_{i+1}, s_{i+1}, \lambda_{i+1}), ...)$$

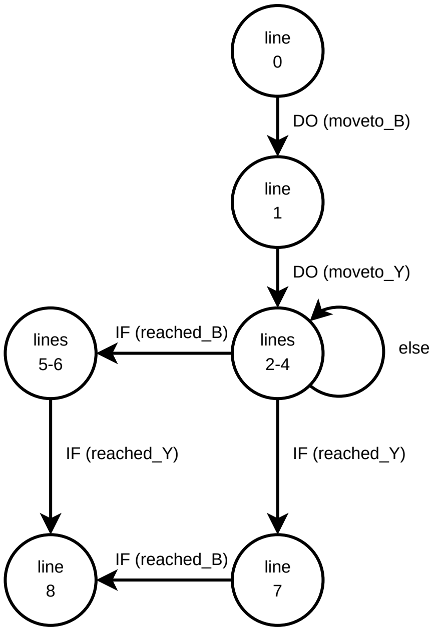


Figure 6.1: Controller program as a flow-chart

The replace operator on controller programs is defined as follows:
Let $P = (..., (l_{i-1}, s_{i-1}, \lambda_{i-1}), (l_i, s_i, \lambda_i), (l_{i+1}, s_{i+1}, \lambda_{i+1}), ...)$ and $l_{i-1} < l_i < l_{i+1}$, then

$$P[l_i] \leftarrow (s'_i, \lambda'_i) \quad := \quad (..., (l_{i-1}, s_{i-1}, \lambda_{i-1}), (l_i, s'_i, \lambda'_i), (l_{i+1}, s_{i+1}, \lambda_{i+1}), ...)$$

The remove operator on controller programs is defined as follows:
Let $P = (..., (l_{i-1}, s_{i-1}, \lambda_{i-1}), (l_i, s_i, \lambda_i), (l_{i+1}, s_{i+1}, \lambda_{i+1}), ...)$ and $l_{i-1} < l_i < l_{i+1}$, then

$$P[l_i] \leftarrow \perp \quad := \quad (..., (l_{i-1}, s_{i-1}, \lambda_{i-1}), (l_{i+1}, s_{i+1}, \lambda_{i+1}), ...)$$

The function $R_0 : P \times L \rightarrow \mathbb{N}$ returns the *basic* reference-count of a given label. It returns 1 if the label is the root entry-point or if the preceding command was no GOTO command.

$$R_0(P, l) := \begin{cases} 1 & : \quad l = L_0 \\ 1 & : \quad \exists (l, s, \lambda) = P[l--] : \; s \neq GOTO(.) \\ 0 & : \quad \text{else} \end{cases}$$

Remember that L_0 denotes the label of the first command in the program.
The function $R_{jump} : P \times L \rightarrow \mathbb{N}$ returns the number of commands that refer a given label.

$$R_{jump}(P, l) := \left| \{ (l, s, \lambda) \in P \mid s = IF(., l) \; \vee \; s = GOTO(l) \} \right|$$

The function $R : P \times L \rightarrow \mathbb{N}$ returns the *total* reference count of a given label.

$$R(P, l) := R_0(P, l) + R_{jump}(P, l)$$

6.3 From Strategies to Controller Programs

A state space represents a description of the behaviour of a controllable timed system. In each state, the plant listens for some external requests and/or executes some spontaneous reaction by its own. Switching the point of view to the controller side, the requests are now possibilities to influence the behaviour and the reactions are non-deterministic alternatives that must be observed all the time, in order to be up-to-date with the state of the plant.

An execution cycle of a modern programmable logic controller works as follows: First, all states of the sensors are read and stored in an internal *in-buffer*. Subsequent read instructions will access this in-buffer instead of directly querying the sensors. Then, some instructions are executed until some wait condition is reached. Here, all write requests are buffered in an *out-buffer*. The duration of executing an instruction is negligible small (< 1 msec). Finally, the out-buffer is flushed (i.e. that the actuators are controlled at a time).

In the definition of timed automata (section 2.2), it was defined that transitions are instantaneous, meaning that no time can elapse when taking a non-delay transition. Only staying at some node makes it is possible that time can pass by. Looking at the working principle of a real logic controller, indeed, it can be assumed that when executing a non-delay transition, no time elapses.

The basic idea of constructing a program that controls a non-deterministic plant, represented as a state space, looks as follows:

- Each state in the state space corresponds to a state in the controller program.
- Each transition in the state space corresponds to a command and a jump to another state in the program.
- Uncontrollable transitions correspond to querying sensors.
- Controllable transitions correspond to activating actuators.
- Resetting clocks in the state space is interpreted as resetting timer-objects in the program.
- Querying clock constraints corresponds to querying timers.

Querying timers means testing if they have passed a certain limit. In a controller state, when there is only one uncontrollable edge, this state should evaluate to ”wait until some time has passed”, which is actually controllable again. Before a request from the controller can be sent to the plant, it must be tested firstly if a spontaneous reaction has occurred. Therefore, being at a certain program state, the input-signals must be checked first and only after that, a request can be sent.

6.4 Selection of Controllable Transitions

In the previous section, it was established that sending a request corresponds to activating an actuator and jumping to some other program state. Since the found winning strategy is non-deterministic, i.e., in some states there might be several controller decisions, the controller could pick any arbitrary request. Indeed, a bad selection could not bring the machine into some undesired state, but time could be unnecessarily wasted, anyway.

Therefore, a selection-heuristic is introduced that works as follows: Having the choice between some requests to execute in a certain state, then the request, which leads as fastest to the next state where the plan advances, is to be taken. Thus, it is assumed that the machine responds in an optimal way. Note that it is not possible to do that selection already during the exploration phase. Because it might turn out that some of the requests are invalid and must be removed. So we must keep the other valid requests in the state space as long as we are not finished.

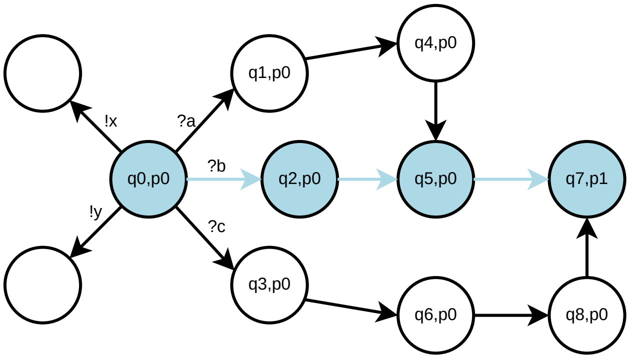


Figure 6.2: Controller request selection heuristic

Figure 6.2 shows an example state space. The current state is $(q_0,\ p_0)$. Now, one has to choose one of the three requests $?a$, $?b$, or $?c$. Selecting $?a$ or $?c$, the nearest state where the plan changes, $(q_7,\ p_1)$ is four transitions away. When selecting $?b$, one gets to $(q_7,\ p_1)$ with only three transitions. Therefore, $?b$ is selected and can stay in the state space while $?a$ and $?c$ are removed.

Algorithm 8 traverses a given state space in a BFS manner and removes all multiple request transitions from the states such that the new state space has at most one request transition per state. It works as follows: The new root state s'_0 corresponds to the original root state s_0 (line 1). L represents the BFS search set, which is initialised with $\{s_0\}$ (line 2). The main loop runs as long as new states were added (lines 3-15). For each state s in L, s is also added to S' (line 6) and the two sets $\Delta^!_s$ and $\Delta^?_s$ are computed (lines 7-8). These two sets contain the outgoing uncontrollable (!)-, guarded- (%), delay- (°), and controllable ($?$) edges starting from s. If $\Delta^?_s$ has more than one element, the (optimal) request is chosen w.r.t. the above mentioned heuristic (line 10). Then, all other request edges are removed from $\Delta^?_s$ (line 11). In line 12, the (possible modified) $\Delta^?_s$ and $\Delta^!_s$ are added to Δ'. In the lines 13-14, the neighbouring states are added to L' except those which are already in the new state space.

The function $nextGoal()$ is computed by the BFS based algorithm 9. It works as follows: Here, the search set L contains 2-tuples consisting of the current search state and the distance to the origin state s_0. It is initialised with the first neighbouring state s'_0 and the

```
Input  : Control state space $(S, \Sigma, \Delta, s_0)$.
Output : State space with at most one outgoing request transition per state
         $(S', \Sigma', \Delta', s'_0)$.
$(S',\ \Sigma',\ \Delta',\ s'_0) \leftarrow (\emptyset,\ \Sigma,\ \emptyset,\ s_0)$
$L \leftarrow \{s_0\}$
while $L \neq \emptyset$ do
    $L' \leftarrow \emptyset$
    forall $s \in L$ do
        $S' \leftarrow S' \cup \{s\}$
        $\Delta^!_s \leftarrow \{(x, \langle \alpha, \tau, \lambda \rangle, y) \in \Delta \mid x = s \wedge (\tau =! \vee \tau = D \vee \tau = T)\}$
        $\Delta^?_s \leftarrow \{(x, \langle \alpha, \tau, \lambda \rangle, y) \in \Delta \mid x = s \wedge \tau =?\}$
        if $|\Delta^?_s| > 1$ then
            Pick a $(x, e, y) \in \Delta^?_s$ such that
            $\forall (x', e', y') \in \Delta^?_s : nextGoal(\Delta,\ (x, e, y)) \leq nextGoal(\Delta,\ (x', e', y'))$
            $\Delta^?_s \leftarrow (x, e, y)$
        $\Delta' \leftarrow \Delta' \cup \Delta^!_s \cup \Delta^?_s$
        forall $(x, e, y) \in \Delta^!_s \cup \Delta^?_s : y \notin S'$ do
            $L' \leftarrow L' \cup \{y\}$
    $L \leftarrow L'$
return $(S',\ \Sigma',\ \Delta',\ s'_0)$
```

Algorithm 8: Request transition selection

distance 1 (line 1). The set vis contains all states that were already visited. It is initialised with the empty set (line 2). Again, the main loop iterates as long as no new states were found anymore (lines 3-12). For each search item (s, d), it is checked if the plan part of s differs from the plan part of s_0 (line 7). If it is so, the algorithm is done and d is returned (line 11). But if the both plan parts are equal, then the BFS will be continued on the neighbours of s (line 9). If no state can be found whose plan part differs from the origin state, then the exception $\perp$ is returned (line 13). Note that when calling $nextGoal()$ with a Δ from a winning control state space, the function will always return a valid result since there can be no dead ends.

6.5 Intermediate Code Generation

The actual code generation phase is quite straightforward since the produced code does not have to be compact or optimal. The subsequent optimising techniques that are applied to that code yield finally a more satisfying representation. The set of the rational numbers is chosen as the label domain: $L = \mathbb{Q}$ with $L_0 = 0$.

Algorithm 10 works as follows: At first, for each state in the input control state space, an integer label is generated (lines 1-5). These state/label pairs are stored in a hash-map.

```
Input  : Set of transitions $\Delta$ and origin transition $\delta \in \Delta$ with $\delta = (s_0, e, s'_0)$.
Output: Shortest distance $d \in \mathbb{N}_0 \cup \{\bot\}$ to the next changing plan state.
$L \leftarrow \{(s'_0,\ 1)\}$
$vis \leftarrow \emptyset$
while $L \neq \emptyset$ do
    $L' \leftarrow \emptyset$
    forall $(s,\ d) \in L$ do
        $vis \leftarrow vis \cup \{s\}$
        if $plan(s)$ has advanced w.r.t. $plan(s_0)$ then
            return $d$
        else
            forall $(x, e, y) \in \Delta : x = s \wedge y \notin vis$ do
                $L' \leftarrow L' \cup \{(y,\ d + 1)\}$
    $L \leftarrow L'$
return $\bot$
```

Algorithm 9: Search for the nearest state that shows a progress in its plan part

The integer labels represent the *entry-points* of the states in the code. The root state has always the entry-point 0.

In the following, for each transition starting from a state $s \in S$, a corresponding command is generated. All commands that are produced under a certain s are called the *command-block* of s. Taking a transition $(s,\ e,\ y)$ in the input state space corresponds to executing e and then jumping to $hash[y]$ in the output program.

The main loop iterates over all states in S (lines 7-23). For each state, the uncontrollable transitions are synthesised first (lines 11-13). This is important because the controller must react on certain plant behaviours *before* it can execute any action by its own. For all uncontrollable transitions, an IF-instruction is generated that queries the message of the event. The state s may have at most one controllable transition, which is synthesised as a DO-instruction and inserted after the IF-instructions (lines 14-23). If there are were no controllable transition starting from s, then the controller must remain in the command-block of s until any uncontrollable event becomes $true$. This is achieved by inserting a GOTO-instruction at the end of the command-block. The destination of that command is the entry-point of the current command-block again (line 23).

Assuming that ”first uncontrollable, then controllable transitions”-semantics, it is important that the command-blocks do not overlap. Which means that each command-block must have only one entry-point at its first command. The controller may only enter another command-block via jumping to other entry-points. l_0 represents the entry-point of the current state s. $l_1 = l_0 + 1$ represents the entry-point the subsequent command-block. So the labels of the command-block of q must lie in $[l_0, l_1)$. This is done by inserting every new command at label $(l_1 + l)/2$ and reassigning this value back to l, where initially l is set to

```
Input  : Winning control state space, given by $(S, \Sigma, \Delta, s_0)$.
Output: Intermediate controller program $P = (C_1, C_2, ..., C_n)$.
$n \leftarrow 1$
forall $s \in S \setminus \{s_0\}$ do
    $hash[s] \leftarrow n$
    $n \leftarrow n + 1$
$hash[s_0] \leftarrow 0$
$P \leftarrow ()$
forall $s \in S$ do
    $l_0 \leftarrow hash[s]$
    $l_1 \leftarrow l_0 + 1$
    $l \leftarrow l_0 - 1$
    forall $(x, \langle \alpha, \tau, \lambda \rangle, y) \in \Delta \ : \ x = s \ \wedge \ (\tau = ! \ \vee \ \tau = \% \ \vee \ \tau = \circ)$ do
        $l \leftarrow (l_1 + l)/2$
        $P[l] \leftarrow (IF(\alpha, hash[y]), \lambda)$
    $\Delta_s^? \leftarrow \{(x, \langle \alpha, \tau, \lambda \rangle, y) \in \Delta \mid x = s \wedge \tau = ?\}$
    if $\Delta_s^? = \{(x, \langle \alpha, \tau, \lambda \rangle, y)\}$ then
        $l \leftarrow (l_1 + l)/2$
        $P[l] \leftarrow (DO(\alpha), \lambda)$
        $l \leftarrow (l_1 + l)/2$
        $P[l] \leftarrow (GOTO(hash[y]), \emptyset, \emptyset)$
    else
        $l \leftarrow (l_1 + l)/2$
        $P[l] \leftarrow (GOTO(l_0), \emptyset, \emptyset)$
return $P$
```

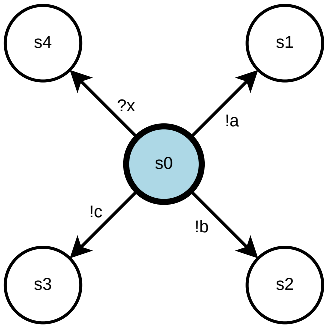
Algorithm 10: Intermediate code generation

$l_0 - 1$.

6.6 Post Optimisations

When talking about code optimality, code size optimality is actually meant. Running time optimality is implicitly assumed due to the bounded liveness constraints, as specified in the plan. The produced intermediate code holds a lot of optimisation potential w.r.t. the code size.

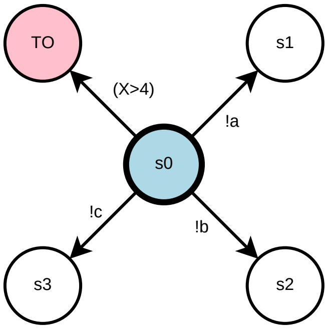
The main reason why optimisation takes place after (and not during) the construction of the state space is that the control state space as a whole is needed. For example, it is necessary to know all reachable states in order to compute the indegree of a state. This indegree corresponds to the reference-count, which is an important part of the precondition



↦

```
1:      IF (A) THEN GOTO 2
1.5:    IF (B) THEN GOTO 3
1.75:   IF (C) THEN GOTO 4
1.875:  DO (X)
1.9375: GOTO 5
```

Figure 6.3: Code extraction example 1



↦

```
1:      IF (A) THEN GOTO 2
1.5:    IF (B) THEN GOTO 3
1.75:   IF (C) THEN GOTO 4
1.875:  IF (X>4) THEN TIMEOUT
1.9375: GOTO 1
```

Figure 6.4: Code extraction example 2

for many optimisations.

In this section, some basic simplification rules are introduced, which are applied step-by-step by a fixed-point algorithm to the initial non-optimal controller program. We have reached code optimality when no simplification rule is applicable anymore. This multi-pass approach is necessary because some optimisations that reduce the code size may lead again to new optimisation potential. Algorithm 11 shows the main post-optimisation algorithm.

```
   Input  : Non-optimal intermediate controller program $P = (C_1, C_2, ..., C_n)$.
            Dependency relation $D$.
   Output: Optimal intermediate controller program.
1  repeat
2      $P' \leftarrow P$
3      $P \leftarrow simplify(P', D)$
4  until $P = P'$
5  return $P$
```

Algorithm 11: Controller program post-optimisation algorithm

The function $simplify$ in line 3 applies the following simplification rules.

6.6.1 WAITUNTIL-replacement

Whenever a command-block only consists of an IF- and a GOTO-instruction, then it can be substituted to a command-block containing a WAITUNTIL-instruction and another GOTO-instruction with the same destination as the IF-instruction. Such command-blocks are generated when a state in the state space has only one transition that is uncontrollable.

$$\begin{gathered}(...,(l_0, IF(cond, dest), \lambda), (l_1, GOTO(l_0), \emptyset), ...)\\ \mapsto \\ (...,(l_0, WAITUNTIL(cond), \lambda), (l_1, GOTO(dest), \emptyset), ...)\end{gathered}$$

This replacement seems quite senseless, since we do not reduce the code-size in the first place. But looking at the next simplification rules, it gets more clearly that in the further optimisation passes standalone WAITUNTIL-instructions can be much better handled than IF- with GOTO-instructions.

Example:

```
1: IF (A) THEN 10                1: WAITUNTIL (A)
                        ↦
2: GOTO 1                        2: GOTO 10
```

6.6.2 Inlining

The idea is quite simple: GOTO-instructions that refer to entry-points of command-blocks that only contain one WAITUNTIL- or DO-instruction and a GOTO-instruction, can be replaced by the referenced command-block.

Let s be an arbitrary non IF-instruction and $R(P, l_2) = 1$, then

$$\begin{gathered}(...,(l_0, GOTO(l_2), \emptyset, \emptyset), (l_1, ...), ..., (l_2, s, \lambda), (l_3, GOTO(dest), \emptyset, \emptyset), ...)\\ \mapsto \\ (...,(l_0, s, \lambda), ((l_0 + l_1)/2, GOTO(dest), \emptyset, \emptyset), (l_1, ...), ...)\end{gathered}$$

Example:

```
                                 1:   DO (A)
1: GOTO 10                       1.5: GOTO 20
2: DO (B)                        2:   DO (B)
...                     ↦        ...
10: DO (A)                       10:  DO (A)
11: GOTO 20                      11:  GOTO 20
```

6.6.3 Reference-inlining

When an IF- or GOTO-instruction refer to another GOTO-instruction, then the destination of the actual instruction can be replaced by the destination of the referenced GOTO-instruction.

$$\begin{gathered}(...,(l_0, GOTO(l_1), \lambda), ..., (l_1, GOTO(dest), \emptyset), ...)\\ \mapsto \\ (...,(l_0, GOTO(dest), \lambda), ..., (l_1, GOTO(dest), \emptyset), ...)\end{gathered}$$

respectively:

$$(...,(l_0, IF(cond, l_1), \lambda), ..., (l_1, GOTO(dest), \emptyset), ...)$$
$$\mapsto$$
$$(...,(l_0, IF(cond, dest), \lambda), ..., (l_1, GOTO(dest), \emptyset), ...)$$

Example:

```
1: IF (A) THEN 10                1: IF (A) THEN 20
...                      ↦       ...
10: GOTO 20                      10: GOTO 20
```

6.6.4 Unreachable command-block removal

Entry-points of command-blocks, whose total reference-count is zero can never be reached during execution. Therefore, they can be removed without substitution. Let l_1 be a label of P and $R(P, l_1) = 0$, then

$$(...,(l_0, s_0, \lambda_0), (l_1, s_1, \lambda_1), (l_2, s_2, \lambda_2), ...)$$
$$\mapsto$$
$$(...,(l_0, s_0, \lambda_0), (l_2, s_2, \lambda_2), ...)$$

Example:

```
10: GOTO 13
11: DO (A)                       10: GOTO 13
12: DO (B)               ↦       12: DO (B)
13: DO (C)                       13: DO (C)
```

6.6.5 Redundant GOTO removal

A GOTO-instruction is redundant if it refers to the very next command. Therefore, a redundant GOTO can be removed because this would not change the execution order of the commands in the controller program. Let l_1 be a label of P and $R(P, l_1) \leq 1$, then

$$(...,(l_0, s_0, \lambda_0), (l_1, GOTO(l_2), \emptyset), (l_2, s_2, \lambda_2), ...)$$
$$\mapsto$$
$$(...,(l_0, s_0, \lambda_0), (l_2, s_2, \lambda_2), ...)$$

Example:

```
10: DO (A)
11: GOTO 12                      10: DO (A)
12: DO (B)               ↦       12: DO (B)
13: DO (C)                       13: DO (C)
```

6.6.6 Redundant IF removal

Some events are in relationship with each other. This means that a specific event may only occur if another related event occurred before. Hence, some IF-instructions in the program can be removed because we can surely assume that the condition will never become $true$. Also, when a condition of an IF-instruction is once $true$, it will remain $true$ until some other event occurs, which influences that condition. Note that letting time pass is also an event, i.e., one can model that the result of a test is influenced just by waiting.

These dependencies are a matter of the problem definition and cannot be found automatically. Thus, they must be provided a priori. See section **??** for a formal language description of the dependency relation. We define the ”remove-if-dependent” operator $\ominus : 2^{\Sigma} \times D \times \Sigma \rightarrow 2^{\Sigma}$:

$$M \ominus (D, \alpha) := M \setminus \{\beta \in M \mid \alpha \text{ and } \beta \text{ are dependent w.r.t. } D\}$$

Algorithm 12 shows a recursive approach for finding and removing redundant IF-instructions in a given controller program P with a dependency relation D.

```
Input  : Controller program $P = (C_1, C_2, ..., C_n)$ containing redundant
         IF-instructions. Dependency relation $D$.
Output : Adjusted controller program $P' = (C'_1, C'_2, ..., C'_n)$ containing no
         redundant IF-instructions.
$(P', vis) \leftarrow removeIF(P, D, 0, \emptyset, \emptyset, \emptyset)$
return $P'$
```

Algorithm 12: Removal of redundant IF-instructions

Before looking at the function $removeIF$, we define two auxiliary functions: the function $B : P \times L \rightarrow 2^{C}$ returns all commands within a given command-block, referenced by its entry-point:

$$B(P, l) := \begin{cases} \{(l, s, \lambda)\} \cup B(P, l') & : \quad (l, s, \lambda) = P[l] \wedge s \neq GOTO(x) \wedge \exists P[l++] = (l', s', \lambda') \\ \{P[l]\} & : \quad \text{else} \end{cases}$$

The function $W : 2^{C} \rightarrow \{true, false\}$ returns $true$ if the given command-block contains only IF-instructions (except the last GOTO-instruction). Then we say that the given block is a *wait-block*:

$$W(\{(l_1, s_1, \lambda_1), ..., (l_n, s_n, \lambda_n)\}) := \begin{cases} true & : \quad s_n = GOTO(l_1) \wedge \nexists s \in \{s_1, ..., s_{n-1}\} : \ s \neq IF(x, y) \\ false & : \quad \text{else} \end{cases}$$

Example wait-block:

```
1: IF (A) THEN 10
2: IF (B) THEN 15
3: IF (C) THEN 20
4: GOTO 1
```

The recursive function $removeIF(P, D, l_0, vis, T, F)$ works as follows: Redundant IF-instructions should be removed out of the given controller-program P. Where D is a dependency-relation, l_0 the entry-label of the command-block to examine, vis the set of already visited labels, T is a set of conditions, which must be $true$, and F is a set of conditions, which must be $false$ due to prior jumping decisions.

At the beginning, it is checked if l_0 was already visited (lines 1-2). If it is so, the unchanged program is returned. If the command-block referenced by l_0 is a wait-block (i.e. that it waits for some events to occur) and the reference-count of l_0 is greater than 2, then the two sets T and F are resetted to the empty set (line 5). Because then, we know that this command-block is referenced by at least three other commands. Two of them do we know: the basic reference of l_0 and (since this is a wait-block) the loop GOTO-instruction at the end of the block.

If l_0 has only two references, then we can be sure that this command-block is not referenced elsewhere. We do not need to reset T and F completely, but we must remove all conditions that are related to $TIME$ (line 7). This is because we are in a wait-block and time can pass as long the controller is waiting for any event to occur. If this is no wait-block and the reference-count of l_0 is greater then 1, we must completely reset T and F, since it is very likely that the other referencing command does not have the same T and F sets (line 9).

The main loop iterates over all commands of the given command-block (lines 11-33). Here, the label of each passed command is added to vis. Then, in dependence on the instruction s, the following steps are performed: If s is a GOTO-instruction, $removeIF$ is called recursively on the destination label (line 15). If s is a DO-instruction, all conditions that depend on the action of this command are removed from T and F. If s is a WAITUNTIL-instruction, all conditions that depend on $TIME$ are removed from T and F (line 22). If the condition of this instruction is in T, then we can remove the current command from P, since the instruction (WAITUNTIL($true$)) would not have any effect (line 24).

If s is an IF-instruction with condition $cond$ and $cond$ is in F, then we can also remove the command (line 27), since the controller would never jump to the destination label (IF ($false$) THEN GOTO dest). Otherwise, if $cond$ is in T, we can replace the IF-instruction by a GOTO-instruction (line 30), since the succeeding commands in that block are never reached (IF ($true$) THEN GOTO dest). If $cond$ is neither in F nor in T, we must recurse again on the destination label and extend T with $cond$ for that function call (line 32). After that, we add $cond$ to F, since in this branch we did not take the IF-instruction and can, therefore, assume that $cond$ is $false$ (line 33).

```
if $l_0 \in vis$ then
    return $(P, vis)$
if $W(B(P, l_0))$ then
    if $R(P, l_0) > 2$ then
        $(T, F) \leftarrow (\emptyset, \emptyset)$
    else
        $(T, F) \leftarrow (T \ominus (D, \mathbf{TIME}), F \ominus (D, \mathbf{TIME}))$
else if $R(P, l_0) > 1$ then
    $(T, F) \leftarrow (\emptyset, \emptyset)$
$P' \leftarrow P$
foreach $(l, s, \lambda) \in B(P, l_0)$ do
    $vis \leftarrow vis \cup \{l\}$
    switch $s$ do
        case $GOTO(dest)$
            return $removeIF(P', D, dest, vis, T, F)$
        case $DO(action)$
            $(T, F) \leftarrow (T \ominus (D, action), F \ominus (D, action))$
        case $WAITUNTIL(cond)$
            $(T, F) \leftarrow (T \ominus (D, \mathbf{TIME}), F \ominus (D, \mathbf{TIME}))$
            if $cond \in T$ then
                $P'[l] \leftarrow \bot$
        case $IF(cond, dest)$
            if $cond \in F$ then
                $P'[l] \leftarrow \bot$
            else
                if $cond \in T$ then
                    $P'[l] \leftarrow (GOTO(dest), \lambda)$
                else
                    $(P', vis) \leftarrow removeIF(P', D, dest, vis, T \cup \{cond\}, F)$
                    $F \leftarrow F \cup \{cond\}$
return $(P', vis)$
```

Function `removeIF(`P, D, l_0, vis, T, F`)`

6.7 Assembler Code Generation

The final step in the synthesis algorithm is the compilation of IEC 1131-3 conforming assembler code [21] from the optimised intermediate code. Such an assembler code can be uploaded without any modifications into a real Siemens S7 programmable logic controller (PLC), which is the current industrial standard.

6.7.1 Target System

As we have seen, prior to the assembler code generation, an intermediate program is generated. Because of this abstraction, we are able to implement a wide range of target systems. In order to give a proof of concept of the developed synthesis approach, a compiler was implemented that translates the intermediate programs to IEC 1131-3 conforming assembler code [21].

The current standard PLC is a Siemens S7. Using the special programming software Siemens Step7, one is able to upload IEC 1131-3 assembler code into such a controller. The machine-sensory is attached as HIGH/LOW input-signals on the controller. The controller executes the synthesised assembler program that reads these input-signals and controls some output-signals. These output-signals are linked with machine-actuators that perform some movements. Reading and writing of input and output-signals works as follows:

- Before the controller program (also called as the main cycle) is executed, all input-signals are cached in an input-buffer. The main cycle does not read any input-signals directly from the sensors but from the input-buffer. During one cycle, the input-signals do not change.
- All writing requests are cached in an output-buffer. After the termination of the main cycle, this buffer is flushed such that all actuators are controlled at the same time.

In the memory of an S7 controller, there are so called data- and function-blocks that represent the data- and code-segments. The synthesised program is placed into such a code-segment.

6.7.2 IEC 1131-3 Code Compilation

Because of the input/output-caching functionality of a Siemens S7 controller, as described in the last section, it is necessary that the main cycle must terminate if it waits for a specific event from the sensors. Since only then, state-changes of the input-signals can be noticed by the controller. Thus, the assembler program must not contain loops. For example, it is not allowed to compile the following ”wait-for” intermediate code

```
...
10: IF (<sensor X>) THEN GOTO 20
11: GOTO 10
...
20: ...
```

just by syntactical replacement of equivalent assembler functions:

```
...
aaaa: A    <sensor X>
      JC   bbbb
      JU   aaaa
...
bbbb: ...
```

This would result in a hang-up of the controller because the query-command `A <sensor X>` will never return a distinct result since the input-buffer does not get refreshed. Note that the assembler command `JC` stands for a conditional, while `JU` stands for an unconditional jump.

Unfortunately, an S7-controller has no built-in function that allows to terminate the current cycle and resume later on the same position. Also, no program counter, containing the current position in the code-segment, is accessible by any assembler command. Therefore, as a matter of fact, the synthesised assembler program has to store the current position in a helper variable, a so called marker-byte. Now, on a re-entrance, depending on that position-variable, the program can jump to the command where the execution was terminated during the last cycle.

The compilation of an intermediate GOTO-command depends on its destination; if the destination address lies *behind* the current position, then this GOTO-command is compiled as an unconditional jump (`JU`). If the destination address lies *before* the current position, in order to avoid possible loops, an assembler code is generated that loads the destination address into the position-variable and then terminates the cycle. The compilation of IF-commands works analogously. The conditions of the IF-commands and the actions of the DO-commands are compiled with respect to a given assembler look-up table in which all symbolic events are mapped to some concrete assembler commands.

The labels of an assembler program may only contain letters, not digits, and can have at most a length of four. Because it is not possible to store the labels as dynamical references in a marker byte, a so called ”Jump to Labels” (`JL`) code-sequence must be generated at the beginning. There, the contents of the position-variable is queried from the dedicated marker byte, and then, depending on the (numerical) value, a jump to the corresponding label is made.

The following example intermediate program:

```
0 :   WAITUNTIL (occurred_pressed);
1 :   DO (turn_on_led);
2 :   WAITUNTIL (occurred_released);
3 :   DO (turn_off_led);
4 :   GOTO 0;
```

with the following symbolic look-up table:

```
occurred_pressed  = A I124.0
occurred_released = AN I124.0
```

```
turn_on_led      = SET;S Q125.6
turn_off_led     = SET;R Q125.6
```

is compiled to this assembler program:

```
      L     MB0
      JL    ud
      JU    aaaa
      JU    aaac
      JU    end
ud:   JU    end

aaaa: L     0
      T     MB0
      A     I124.0
      JCN   end
      SET
      S     Q125.6
aaac: L     1
      T     MB0
      AN    I124.0
      JCN   end
      SET
      R     Q125.6
      L     0
      T     MB0
      JU    end

end:  NOP   0
```

`I124.0` is an input-signal, `Q125.6` is an output-signal, and in the marker byte `MB0`, the current program position is stored.

Chapter 7

Practical Experience

7.1 Tool Implementation

Within the scope of this diploma thesis, the synthesis algorithm has been also implemented in C++. The back-end synthesis functions were efficiently programmed using STL functions and classes. For the parser routines, the standard tools *flex* and *yacc* [17] were used. As a front-end API, *wxWidgets* [22] was used such that a platform independent GUI implementation was possible. The tool graphviz from AT&T [15] was used as a rendering back-end for the plant components.

7.2 Real World Examples

The following sections describe three real world examples that could be successfully solved with the developed synthesis tool. At first, the lamp example is shown, which has, on the one hand, a quite simple plant but, on the other hand, shows a common problem when one tries to model instantaneous reactions. The second example shows a modelling approach for a gear checking machine, which is a standard example for an industrial machine. The last example shows how a program for a round table can be generated.

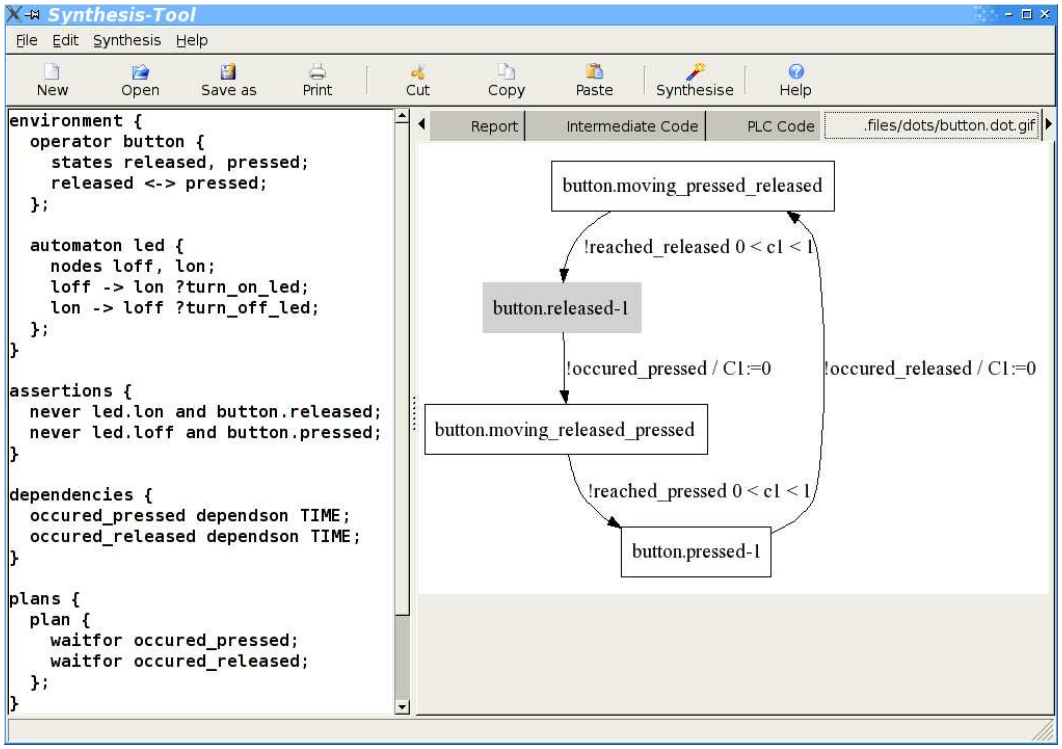
7.2.1 Lamp

A simple example that shows a fundamental issue in modelling reactive components is the Lamp Example. There are two components: a lamp and a button. The Lamp has the two states ON and OFF while the button can be in a RELEASED or PRESSED state. It is required that the lamp should be turned ON when the button is PRESSED and, vice versa, it should be turned OFF when the button is RELEASED again.

So the conjunctive assertions are formulated as follows:

- **never**: lamp is turned OFF **and** button is PRESSED
- **never**: lamp is turned ON **and** button is RELEASED

The plan is defined as:


Synthesis-Tool
File Edit Synthesis Help
New Open Save as Print Cut Copy Paste Synthesise Help
Report Intermediate Code PLC Code .files/dots/button.dot.gif
button.moving_pressed_released
!reached_released 0 < c1 < 1
button.released-1
!occured_pressed / C1:=0
!occured_released / C1:=0
button.moving_released_pressed
!reached_pressed 0 < c1 < 1
button.pressed-1

```
environment {
  operator button {
    states released, pressed;
    released <-> pressed;
  };

  automaton led {
    nodes loff, lon;
    loff -> lon ?turn_on_led;
    lon -> loff ?turn_off_led;
  };
}

assertions {
  never led.lon and button.released;
  never led.loff and button.pressed;
}

dependencies {
  occured_pressed dependson TIME;
  occured_released dependson TIME;
}

plans {
  plan {
    waitfor occured_pressed;
    waitfor occured_released;
  };
}
```

Figure 7.1: Screenshot of the synthesis tool

- **wait for** button becomes PRESSED
- **wait for** button becomes RELEASED

Modelling the plant, a first straightforward approach would be to model both components, lamp and button, as two-states automata as shown in figure 7.2.

One observers that turning the lamp ON and OFF is controllable while changing the state of the button is uncontrollable since this depends on external user interaction that is, in fact, unpredictable. The combined state space looks as shown in figure 7.3.

It is obvious that, starting at (OFF,RELEASED), there is no strategy that leads to the next target state, (ON,PRESSED), without entering a fail state. So this first straightforward specification is actually unrealisable, which can also be demonstrated using the developed synthesis tool. Recall that our target-controller can only *react* on certain plant events. Therefore, it is necessary to give the controller a chance of reacting to !press. Since we can make use of timed automata, we can introduce a clock C to give the controller the time to react on !press: if !press occurs at location RELEASED, before the execution enters the target state PRESSED, an intermediate state RELEASED′ is entered having the invariant $C < 1$. From RELEASED′, the intermediate !press′-transition, having the guard $C > 0$, leads to PRESSED. The same is done for the PRESSED → RELEASED case. This new (reactive) button is shown in figure 7.4.

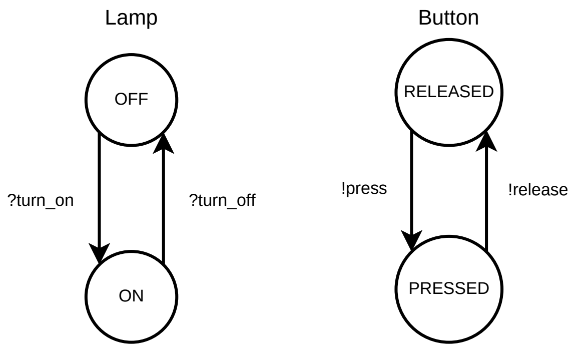


Figure 7.2: Lamp Example: naive approach

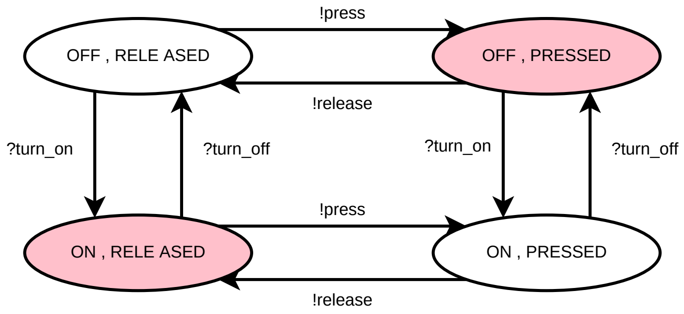


Figure 7.3: Lamp Example: naive approach combined state space

As one can see, there exists a winning strategy, now. Since as long as $C = 0$, the controller has enough time to send a `?turn_on`-request after a `!press`-event was noticed. After that, it can let time pass until $C > 0$, i.e., it waits an arbitrary small time $\epsilon > 0$. Then, the intermediate `!press'`-event happens immediately that brings the plant to `(ON,PRESSED)`.

The final intermediate controller program is generated as follows:

```
0 :   WAITUNTIL (press);
1 :   DO (turn_on);
2 :   WAITUNTIL (release);
3 :   DO (turn_off);
4 :   GOTO 0;
```

Note that all clock, delay, and intermediate events are discarded.

7.2.2 Gear Checking Machine

The objective of a gear checking machine is to load workpieces (gears) that come over an intake transporting-belt and to classify them. After the classification, the gears are unloaded

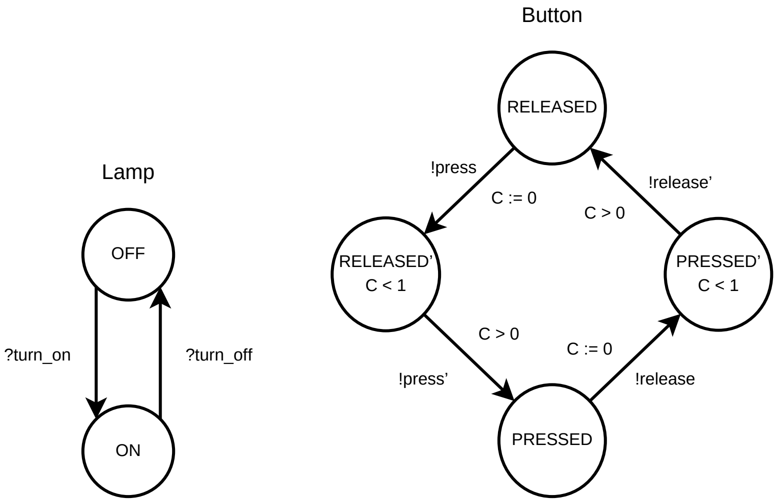


Figure 7.4: Lamp Example: reactive approach

in a dedicated outtake. A gripper arm transports the gears within the machine to the various locations. It can be controlled independently in two directions: vertically (up or down) and horizontally (intake, classification, or outtake). There are also some other components as well: a mandril that fixes the loaded gear during classification, a PC-software that performs the actual classification, a loader that loads the gear into the gripper at the intake, and an unloader that unloads the gear from the gripper into an outtake.

The mechanical setup of the machine induces some constraints on the components:

- The gripper can move horizontally **only if** it is in the upper position
- The software can classify **only if** the mandril is strained
- The gripper can move vertically **only if** the mandril is relaxed (since only then, the workpiece is released and can be transported away)
- The loader can load **only if** the horizontal position of the gripper is at the intake **and** the vertical position is down
- The software can classify **only if** the horizontal position of the gripper is at the processing station **and** the vertical position is down
- The mandril can change its state **only if** the horizontal position of the gripper is at the processing station **and** the vertical position is down
- The unloader can unload **only if** the horizontal position of the gripper is at the outtake **and** the vertical position is down

Note that the last four constraints define, which component is enabled when the gripper (and so the workpiece) is at a certain position.

The plant can be modelled in a straightforward manner. The horizontal movement of the gripper is given by a hardware unit (as introduced in section **??**) that has four resting states (`intake`, `process`, and `outtake`) plus the implicitly generated intermediate states (e.g. `moving_intake_process`). The vertical movement is also modelled by a hardware unit. But this one consists only of the two resting states `up` and `down`. The mandril is the third hardware unit that contains the resting states `relaxed` and `strained`. The software, loader, and unloader are defined by explicit timed automata that are shown in figure 7.5.

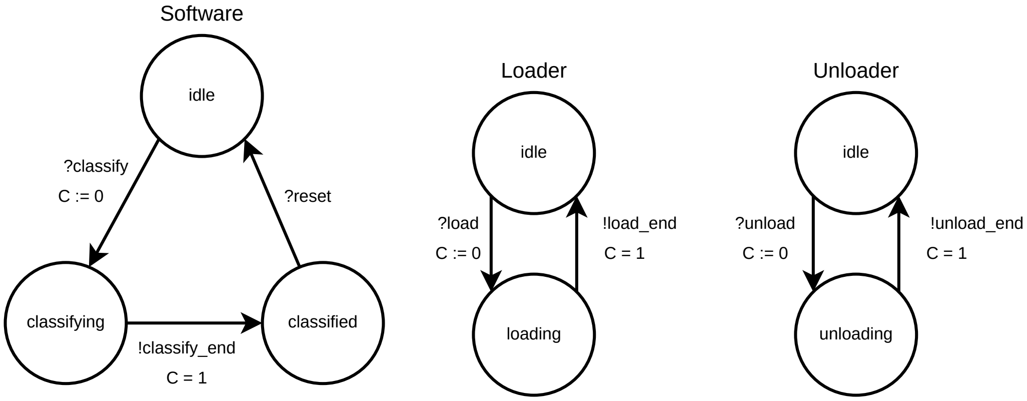


Figure 7.5: Components of the Gear Checking Machine

The actions of the components are modelled such that time elapses when they are executed. For the sake of simplicity, the plant is modelled such that each action will take exactly one time unit. Because it is not allowed that a processing cycle of the machine takes arbitrarily much time, a global timeout-clock is part of the goal-definitions that lets the plan fail as soon as this clock exceeds 15 time units. The precise plan-definition is defined as:

```
plans {
  plan {
    clocks timeout 15;

    reset timeout;
    waitfor loaded;
    waitfor classify_end;
    waitfor unloaded;
  };
}
```

Based on that, the synthesis tool generates:

```
0 :   DO (load);
```

```
1 :   WAITUNTIL (loaded);
2 :   DO (moveto_up);
3 :   WAITUNTIL (reached_up);
4 :   DO (moveto_process);
5 :   WAITUNTIL (reached_process);
6 :   DO (moveto_down);
7 :   WAITUNTIL (reached_down);
8 :   DO (moveto_strained);
9 :   WAITUNTIL (reached_strained);
10 :  DO (classify);
11 :  WAITUNTIL (classify_end);
12 :  DO (moveto_relaxed);
13 :  DO (reset_software);
14 :  WAITUNTIL (reached_relaxed);
15 :  DO (moveto_up);
16 :  WAITUNTIL (reached_up);
17 :  DO (moveto_outtake);
18 :  WAITUNTIL (reached_outtake);
19 :  DO (moveto_down);
20 :  WAITUNTIL (reached_down);
21 :  DO (unload);
22 :  WAITUNTIL (unloaded);
23 :  DO (moveto_up);
24 :  WAITUNTIL (reached_up);
25 :  DO (moveto_intake);
26 :  WAITUNTIL (reached_intake);
27 :  DO (moveto_down);
28 :  WAITUNTIL (reached_down);
29 :  GOTO 0;
```

7.2.3 Round Table

A round table is an often occurring design-pattern in mechanical engineering. It consists of n bins that are located on the rim of a cycling disc. The bins are uniformly distributed such that the angle between two bins is $\gamma = \frac{360°}{n}$. If the table gets a `?cycle`-request, it rotates by γ in clockwise direction such that the bin that was at position $i\gamma$ is now at $(i+1)\gamma$. At each position, a certain action can be applied to the bin. Each bin can hold a workpiece that is, by cycling the table, processed through the various processing stations of the machine.

In this example, we have a round table with eight bins ($n = 8$ / $\gamma = 45°$). The workpieces are marbles having the colors red, green, and blue. The positions have the following functionality:

- At position 0, a new marble is loaded into the bin.
- At position 2, the color of the loaded marble is determined by a sensor.

- At position 4, the marble is unloaded into the ”red”-outtake.
- At position 5, the marble is unloaded into the ”green”-outtake.
- At position 6, the marble is unloaded into the ”blue”-outtake.
- At position 7, the marble is unloaded into the ”undef”-outtake.
- At the positions 1 and 3, nothing happens.

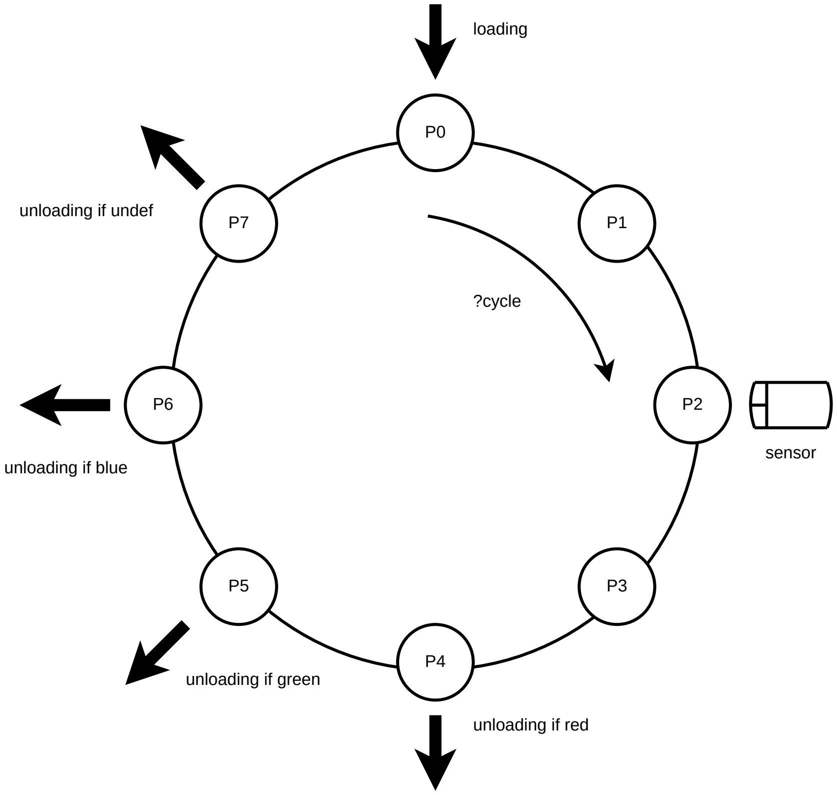


Figure 7.6: Round table with eight bins

The formal specification can be given in a straightforward manner; the round table itself is represented by an automaton having the two locations `IDLE` and `CYCLING`. At `IDLE`, it waits for a `?cycle` request that brings the table-automaton into `CYCLING`. At `CYCLING`, a `!cycling_end`-event back to `IDLE` can occur spontaneously, indicating that the next position is reached.

All the bins are modelled by automata that have the same structure. Only the initial locations vary. They have eight position-locations `POS0` to `POS7` and a `RESTART` location. The position-locations are linked sequentially with a `$cycle_end`-transition such that every time the table rotates for one position, the bin-automata advance in the next position-location as well. POS7 is not linked directly with POS0. In order to notice whenever a bin returns to position POS0, after POS7 an intermediate location RESTART is added. From there, via a !restart-transition, the bin-automata reach POS0 again.

The sensor and the classification information is represented by a marble-automaton. Starting at location UNDEF, it waits for a restart-event to get into the location EMPTY. There, it waits for a load_end-event to get into the location UNCLASSIFIED. Being in that location, the controller can send a ?classify-request, and after some time, a !classify_end is sent by the plant back to the controller, signalising that the classification of the color is done. Then, the automaton will be in the location VALIDATING and the controller can request a ?store_class that safes the measured color information into the controllers' memory. After that, at location CLASSIFIED, the controller can send either ?unload_a, ?unload_b, ?unload_c, or ?unload_d to go into UNLOADING_AT_A, UNLOADING_AT_B, UNLOADING_AT_C, or UNLOADING_AT_D, respectively. The unloading process ends with the feedback message !unloading_end. Then, the automaton is in the location UNLOADED where a prioritised !unloaded event occurs that signalises the plan that the marble was unloaded. All plant automata are shown in section A.

Let $0 \leq i \leq 7$, then the conjunctive assertions are formulated as follows:

- **never**: table is CYCLING **and** any marble is CLASSIFYING, VALIDATING, or UNLOADING
- marble i can CLASSIFYING **only if** bin i is in POS2
- marble i can UNLOADING_AT_A **only if** bin i is in POS4
- marble i can UNLOADING_AT_B **only if** bin i is in POS5
- marble i can UNLOADING_AT_C **only if** bin i is in POS6
- marble i can UNLOADING_AT_D **only if** bin i is in POS7

The four last assertions associate the four outtakes A, B, C, and D to the corresponding positions.

For each bin i, the state guards specify whenever a marble should be unloaded at a certain position:

- marble i in UNLOADING_AT_A **along with** bin i in POS4 **is guarded by** c0_is_blue
- marble i in UNLOADING_AT_B **along with** bin i in POS5 **is guarded by** c0_is_green
- marble i in UNLOADING_AT_C **along with** bin i in POS6 **is guarded by** c0_is_red
- marble i in UNLOADING_AT_D **along with** bin i in POS7 **is guarded by** c0_is_undef

For each bin i, the plan says:

- **wait for** bin i sends restart

- **wait for** any unloader sends unloaded **or**, if bin i sends restart prior to any unloaded event, then the plan has failed

The precise specification code is given in A. The intermediate code for controlling one bin in this round table of eight positions is given in A.

7.3 Benchmarks

The following benchmarks have been measured on an Intel Pentium III Mobile 1.2 GHz with 512 MB RAM, running Linux 2.6.12, compiled with GCC 3.3, and -O3 optimisations enabled.

Example	Visited	Winning	Code size	Time [sec]
Lamp	11	10	5	0
GCM	1660	483	30	6.0
RT 1/8	60	51	53	0
RT 2/8	206	138	141	0.4
RT 3/8	656	330	312	1.1
RT 4/8	2118	841	773	5.5
RT 5/8	Out of memory			

In this table, for each example from the sections before, the number of totally visited states, the winning states, the number of lines of the intermediate program, and the actual generation time are shown. RT n/8 stands for round table with eight positions but only n bins considered. GCM means gear checking machine. The round table examples are untimed, while the lamp and the gear checking examples are timed.

The round table results with one to four bins illustrate how wrong controller decisions at the beginning are firstly detected at the very end of a complete cycle; Recall that, hereby, the controller has to decide whether a bin at a certain position in the cycle should be unloaded or not. The plan was formulated quite vaguely: ”Each bin must be unloaded within one complete cycle”. As a result of this specification, some missed unloading requests can only be detected very deeply in the decision tree by the game solving algorithm. Thus, the round table example with five bins exceeds the memory limit.

All actions of the components in the gear checking example consume time. Therefore, clock zone operations are needed that lead to an increase of the overall time. The discrepancy between code size and winning states is due to the effectiveness of the applied post optimisations.

Chapter 8

Conclusions and Outlook

8.1 Conclusions

This diploma thesis presents an innovative synthesis approach that uses optimal model-checking techniques that have already been successfully implemented in standard model-checking tools. With a new developed specification language, the user can easily specify component based industrial problem setups. Along with assertions and goal-definitions, a controller program is automatically generated, or else, if there exists no valid program, it is reported that the specification is unrealisable.

The two basic computational models are timed automata and safety games that were adapted from related work and appropriate extended in order to match the purpose of this thesis, synthesising industrial controller programs. A nested forward/backward fixed-point algorithm finds a non-deterministic winning strategy for the controller in the spanned safety game. It runs on-the-fly on the locations of the plant automata and the clock assignments, symbolised by clock zones, encoded by difference bound matrices.

Because the standard simulation graph of a timed automaton is too abstract for a linear game solving algorithm, a precomputation transforms that implicit representation to an explicit one. A local, component based, approach for doing this is firstly introduced in this thesis. Hereby, instead of transforming the whole product state space, the various plant components are transformed independently from each other. Thus, the overall running time is linear to the size of the product state space.

Instead of directly generating concrete assembler code, a generic intermediate code is synthesised first. Thus, it is possible to reduce the code-size by applying generic optimisation steps. Because of this abstraction, the synthesis approach is applicable to a wide range of platforms since porting means just implementing an appropriate compiler.

In order to get a proof of concept, a prototype was implemented in C++. At the end of the synthesis process, a compilation step compiles the intermediate code in concrete IEC 1131-3 assembler that can be uploaded into a real Siemens S7-300 programmable logic controller, which is the current industrial standard. In cooperation with the Laboratory of Process Automation at Saarland University, real world problem tasks could be solved and implemented on a training S7-300.

A typical life-cycle of an industrial machine looks roughly like this:

1. Create specification.
2. Do manual programming.
3. If verification reports an error, goto 2.
4. Put machine into production (until customer changes specification, then goto 1).

With the synthesis approach no manual programming is necessary anymore. Similarly to the classical development approach, a formal specification must be created that describes the actual problem task. But after that, in contrast to the classical approach, the program generation works completely automatically. This brings not only a boost with the initial development, but also with later customisations. Furthermore, the question, whether or not a certain component-setup is sufficient to solve an industrial problem task, can be answered at the very beginning, since it can be automatically checked if the specification is realisable or not.

8.2 Outlook

Indeed, for the various parts of the synthesis algorithm, there exists some improving potential. In this section, some ideas are sketched that outline possible future work. Independent to each extension, the primary requirement that the synthesis process should run completely automatically, must be always maintained.

8.2.1 Language

A major task in the future will be the improvement of the specification language such that modelling of industrial program tasks will become as intriguing as possible. In order to do so, a preprocessing step prior to the actual parsing of the specification could be applied. By using macros and preprocessor directives, the language could be easily extended by loops and conditionals.

In order to increase expressivity, the specification language could be extended by further data types other than time, which is the only data type at the moment, actually. Possible new data-sensitive elements range from simple scalar variables like integers or floats up to complex objects like sets, lists, or vectors.

8.2.2 Computational Models

In this thesis, an extension to the classical theory of the timed automata is introduced. By distinguishing between controllable, uncontrollable, and synchronisation events, one obtains timed game automata that form the basis for the synthesis algorithm. Indeed, with these modelling techniques, one can model a wide range of industrial problem tasks. However, practical experience with the implemented synthesis tool has shown that some further extensions might be useful to the user. For example, in addition to synchronising two components via events one could also include a state-based synchronisation.

8.2.3 State Space Exploration

The core part of the synthesis algorithm is the state space exploration. The overall running time is dominated by that process. Thus, speeding up the exploration phase results in a major speed up of the whole synthesis algorithm.

With the synthesis approach that is shown in this thesis, asynchronous components that interleave each other form the basis of the state space algorithm. This means, if, for example, two events A and B may occur concurrently, then on the one hand, the ”first A, then B”-case must be considered as well as the ”first B, then A”-case. Thus, both paths must be part of the combined state space, in order to have an exhaustive combined model. However, in practice, it is often not necessary to consider all combinations of parallel occurring events. Using such an a priori knowledge, which events that run in parallel do not need to be considered in every possible combination, can be used to drastically reduce the size of the combined state space, and thus, the size of the generated program and particularly the overall running time of the synthesis algorithm. This technique is called *partial order reduction* [9] that is already quite well known for model-checking. A possible future work could be integrating such techniques in synthesis.

As shown in section 5.3, all controller decisions that lead to a valid state are kept in the state space. After the exploration, the deterministic winning strategy is obtained by removing all non-optimal controller decisions out of the explored state space. A further future work could be bringing the selection heuristic into the actual exploration phase. This could be done by extending the pure *breadth-first-search* to an *informed depth-first-search* algorithm. Hereby, when there are several controllable actions in a discovered state, only one of them is traversed, the others are only marked as possible alternatives for that state, in case that the picked decision becomes invalid.

8.2.4 Code Generation

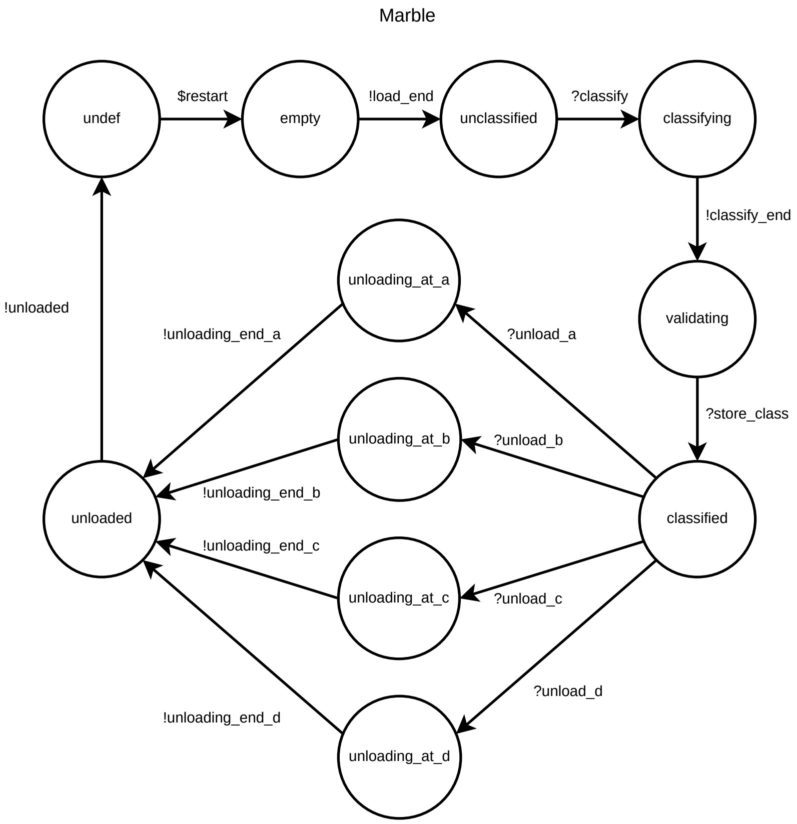
Looking at the generated intermediate programs, one observes that there exists still minimisation potential. For example, one could introduce the concept of subprograms by identifying identical code-parts and replacing them with a function call to a corresponding subprogram. Also, in correspondence to the ideas of including new data-sensitive elements as mentioned above, the size of the intermediate programs could also be reduced by making use of a variable environment.

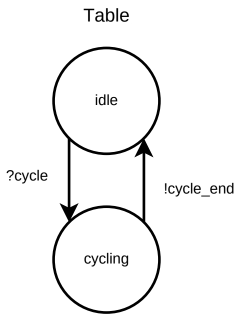
In order to get a more canonical assembler code, one could extend the assembler-compilation at the end of the synthesis algorithm such that *abstract syntax* is generated instead of *concrete syntax*, as it is done right now. Based on that abstract syntax, a second, assembler-code based, optimisation could be applied.

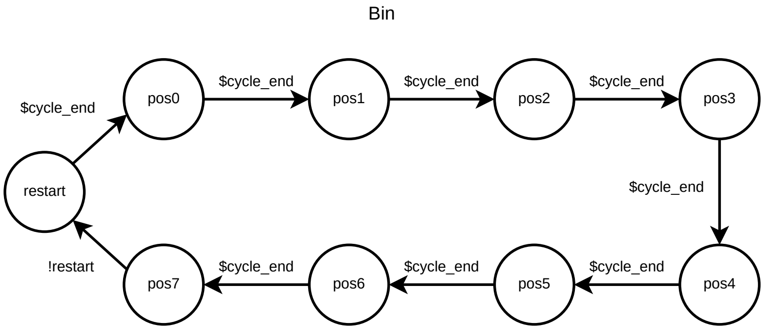
Appendix A

Round Table

Components







Specification

```
plant {
  automaton table {
    nodes idle, cycling;
    idle -> cycling ?cycle;
    cycling -> idle !cycle_end;
  };

  automaton pos_0 {
    nodes restart, p0, p1, p2, p3, p4, p5, p6, p7;
    p0 -> p1 $cycle_end;
    p1 -> p2 $cycle_end;
    p2 -> p3 $cycle_end;
    p3 -> p4 $cycle_end;
    p4 -> p5 $cycle_end;
    p5 -> p6 $cycle_end;
    p6 -> p7 $cycle_end;
    p7 -> restart $cycle_end;
    restart -> p0 !restart instant;
  };

  automaton marble_0 {
    nodes
      undef, empty, unclassified,
      classifying, validating, classified,
      unloading_at_a, unloading_at_b,
      unloading_at_c, unloading_at_d,
      unloaded;

    undef -> empty $restart;
    empty -> unclassified !load_end instant;
    unclassified -> classifying ?classify;
    classifying -> validating !classify_end;
    validating -> classified ?store_class;

    classified -> unloading_at_a ?unload_a;
    classified -> unloading_at_b ?unload_b;
    classified -> unloading_at_c ?unload_c;
    classified -> unloading_at_d ?unload_d;

    unloading_at_a -> unloaded !unloading_end_a;
    unloading_at_b -> unloaded !unloading_end_b;
    unloading_at_c -> unloaded !unloading_end_c;
    unloading_at_d -> unloaded !unloading_end_d;

    unloaded -> undef !unloaded instant;
  };
}

assertions {
  never table.cycling and
    (marble_0.classifying or
     marble_0.validating or
     marble_0.unloading*);
  marble_0.classifying onlyif pos_0.p2;
  marble_0.unloading_at_a onlyif pos_0.p4;
  marble_0.unloading_at_b onlyif pos_0.p5;
  marble_0.unloading_at_c onlyif pos_0.p6;
  marble_0.unloading_at_d onlyif pos_0.p7;
}

guards {
  marble_0.unloading_at_a and pos_0.p4 guardedby c0_is_blue;
  marble_0.unloading_at_b and pos_0.p5 guardedby c0_is_green;
  marble_0.unloading_at_c and pos_0.p6 guardedby c0_is_red;
  marble_0.unloading_at_d and pos_0.p7 guardedby c0_is_undef;
```

```
}

dependencies {
  c0_is_blue dependson store_class_0;
  c0_is_green dependson store_class_0;
  c0_is_red dependson store_class_0;
  c0_is_undef dependson store_class_0;
}

plans {
  plan {
    waitfor restart;
    waitfor unloaded, restart -> failed;
  };
}
```

Intermediate Code

```
0 :   DO (cycle);
1 :   WAITUNTIL (cycle_end);
2 :   DO (cycle);
3 :   WAITUNTIL (cycle_end);
4 :   DO (classify);
5 :   WAITUNTIL (classify_end);
6 :   DO (store_class);
7 :   DO (cycle);
8 :   WAITUNTIL (cycle_end);
9 :   DO (cycle);
10 :  WAITUNTIL (cycle_end);
11 :  IF (c0_is_blue) THEN GOTO 26;
12 :  DO (cycle);
13 :  WAITUNTIL (cycle_end);
14 :  IF (c0_is_green) THEN GOTO 35;
15 :  DO (cycle);
16 :  WAITUNTIL (cycle_end);
17 :  IF (c0_is_red) THEN GOTO 38;
18 :  DO (cycle);
19 :  WAITUNTIL (cycle_end);
20 :  WAITUNTIL (c0_is_undef);
21 :  DO (unload_d);
22 :  WAITUNTIL (unloading_end_d);
23 :  DO (cycle);
24 :  WAITUNTIL (cycle_end);
```

```
25 :  GOTO 0;
26 :  DO (unload_a);
27 :  WAITUNTIL (unloading_end_a);
28 :  DO (cycle);
29 :  WAITUNTIL (cycle_end);
30 :  DO (cycle);
31 :  WAITUNTIL (cycle_end);
32 :  DO (cycle);
33 :  WAITUNTIL (cycle_end);
34 :  GOTO 23;
35 :  DO (unload_b);
36 :  WAITUNTIL (unloading_end_b);
37 :  GOTO 30;
38 :  DO (unload_c);
39 :  WAITUNTIL (unloading_end_c);
40 :  GOTO 32;
```

List of Figures

List of Algorithms

Bibliography

[1] Karine Altisen and Stavros Tripakis. Tools for controller synthesis of timed systems. 2002.

[2] Rajeev Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Stanford, CA, USA, 1992.

[3] Rajeev Alur. Timed automata. 1998. NATO ASI Summer School on Verification of Digital and Hybrid Systems.

[4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[5] Tobias Amnell, Elena Fersman, Paul Pettersson, Hongyan Sun, and Wang Yi. Code synthesis for timed automata. *Nordic Journal of Computing*, 2003.

[6] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243, 1995.

[7] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 546–550, London, UK, 1998. Springer-Verlag.

[8] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games, 2005.

[9] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, jan 2000.

[10] T.H. Corman, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1989.

[11] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 197–212. Springer, 1989.

[12] A. Pnueli E. Asarin, O. Maler and J. Sifakis. Controller synthesis for timed automata. In *Proc. System Structure and Control*. Elsevier, 1998.

[13] Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *IEEE Symposium on Logic in Computer Science*, pages 321–330, June 2005.

[14] S. Finn, M. Fourman, M. Francis, and R. Harris. Formal system design—interactive synthesis based on computer-assisted formal reasoning. In Luc Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Volume 1*, pages 97–110, Houthalen, Belgium, November 1989. Elsevier Science Publishers, B.V. North-Holland, Amsterdam.

[15] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[16] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[17] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc (2nd ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1992.

[18] X. Liu and S. Smolka. Simple linear-time algorithm for minimal fixed points. pages 53–66. Springer, 1998.

[19] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems. 1995.

[20] Zohar Manna. *STeP: the Standford Temporal Prover*, 1994.

[21] Siemens AG. *SIMATIC - Statement List (STL) for S7-300 and S7-400 Programming Reference Manual*, 12/2002 edition.

[22] Julian Smart, Kevin Hock, and Stefan Csomor. *Cross-Platform GUI Programming with wxWidgets*. Prentice Hall, 2005.

[23] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.