Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor's Thesis

Inequality Constraints On Synchronisation Counters

submitted by

Walid Haddad

on Juli 09 2007

Supervisor

Prof. Bernd Finkbeiner, Ph.D.

Advisor

Klaus Dräger

Reviewers

Prof. Bernd Finkbeiner, Ph.D.
Prof. Dr. Reinhard Wilhelm

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, Juli 09, 2007

Walid Haddad

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, Juli 09, 2007

Walid Haddad

Table of Contents

Abstract

The state space of concurrent systems can grow exponentially in the number of system components. Thus, to a certain extent, it may be impossible to explore the entire state space with limited time and memory, a fact known as *the state space explosion problem* [Val98]. This problem arises, for instance, when dealing with reachability analysis. An approach to check synchronization behavior on concurrent systems, with the aim of possible error detection caused by this behavior, is one of the reachability problems. Dealing with this problem by checking all possible states of the system needs a state space that is constructed by the synchronous-product of all processes; this leads to the state space explosion problem.

The aim of this thesis is to attack the state space explosion problem by an approach which avoids an exhaustive construction of its state space, in order to perform reachability analysis. The approach intends to introduce *synchronization counters* for each process and generate inequality constraints on these counters. The generation of these inequality constraints is done by applying certain algorithms and computations on the parallel composed automata representing processes in a system. As a result, our approach can avoid the generation of the whole state space of a system and bypass the state space explosion problem, in order to reach the desired goal.

Keywords: concurrent systems, synchronization, reachability problem, state space explosion.

1 Introduction

1.1 Motivation and Organization

Concurrent systems [BG05] are composed of components that can operate concurrently and communicate with each other. They can operate in real time and thus can be difficult to implement, specify and validate.

A traditional problem faced by the analysis of concurrent systems is the exploration of their state space. Reachability analysis is a powerful formal method for analysis of concurrent systems, which generates the state space of the concurrent program and checks the property of interest on that state space.

In this thesis, we are interested in synchronization behavior which takes place between components (processes) of a concurrent system. Instead of generating the whole state space in order to check the reachability on states, which is part of *dynamic* analysis, our approach will propose a *static* analysis technique based on generating inequality constraints for each state of a single process.

The idea is to generate constraints where each state of a single process in a system will get a finite set of these constraints, i.e. inequality constraints; with the help of these constraints we will be able to decide (for instance) if a state in the state space of a system is not reachable under circumstances.

First, we will generate what we will denote as *cycle images* over cycles in a finite automaton (which represents a process in a system); a cycle image includes the count of each defined synchronization channel along a cycle (which we will define later). With the help of these cycle images we will be able to generate the constraints; we will denote them as inequality constraints on *synchronization counters*.

For example, let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two processes and a, b be the defined channels in a system $\mathcal{R}$ and let z_1 be a state in $\mathcal{P}_1$ and z_2 in $\mathcal{P}_2$. Suppose the inequality constraints (the ones we want to generate) will be as follows:

$$\mathcal{I}_1\colon\ a - b \geq 2$$
$$\mathcal{I}_2\colon\ b - a \geq -1$$

where $\mathcal{I}_1$ and $\mathcal{I}_2$ are inequality constraints for z_1 and z_2, respectively; $\mathcal{I}_2$ can be written as $a - b \leq 1$; we deduce that any state in $\mathcal{R}$ which includes the process states z_1 and z_2 is not reachable because $\mathcal{I}_1$ and $\mathcal{I}_2$ are incompatible.

Going more into details, for the cycle images we construct inequalities of the form:

$$\mathrm{a}_1\mu_1 + ... + \mathrm{a}_n\mu_n \geq 0,$$

where n is the number of channels, μ_1,..., μ_n are the synchronization counter variables (we want to find concrete values for those) and a_1,..., a_n are positive coefficients (the values in the cycle images). After finding the concrete values for the synchronization counter variables, we are able then to identify inequality constraints on the synchronization counters of a finite automaton for each state in it; the inequality constraints will look like the inequalities given in the example above.

The questions that now arise are: How can we compute cycle images for processes of a system and for which cycles? How to identify the inequality constraints on the synchronization counters of a process? How useful are the resulting information about the processes and what can be deduced from them?

The outline of this thesis is as follows:

Chapter 1 is introductory and contains definitions of most basic notation and terminology used throughout this thesis. We define automata as used in this thesis and how to combine a set of automata in order to form synchronized automata. As a modeling example, we will provide a short overview on the modeling language used in this thesis.

In chapter 2, we start with some basic definitions and continue with detailed computation steps of cycle images of finite automata; first we determine which cycles should be considered and then we generate the cycle images for these cycles.

Chapter 3 deals with the final steps, more precisely, with the help of the cycle images we will show how to generate inequality constraints on synchronization counters for each process; we will discuss also the use of these inequality constraints.

In chapter 4, we summarize the whole approach and give a final outlook.

In the appendix, we include some experimental results based on this approach.

1.2 Preliminaries

Before we go into details about our approach, we will first introduce some basic definitions that are required in the following sections of this thesis.

1.2.1 Nondeterministic Finite Automaton - NFA

Formally, an NFA is a 5-tuple, ($\mathcal{S}$, Σ, δ, S_0, F), such that

- $\mathcal{S}$ is a finite set of states.
- Σ is a finite set of symbols, called the alphabet of the NFA.
- $\delta : \mathcal{S} \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^{\mathcal{S}}$ is the transition function.
- $\mathrm{S}_0 \subseteq \mathcal{S}$ is a set of initial states.
- $\mathcal{F} \subseteq \mathcal{S}$ is a finite set of accepting states.

The transition function δ, is also called a next state function, can be identified with the relation $\rightarrow \subseteq \mathcal{S} \times (\Sigma \cup \{\epsilon\}) \times \mathcal{S}$ given by $\mathrm{s} \xrightarrow{\alpha} \mathrm{s}'$ iff $\mathrm{s}' \in \delta(\mathrm{s}, \alpha)$. This means that the automaton can move from state s to state s' if it receives the input symbol α.

The accepting states are defined in the finite set $\mathcal{F} \subseteq \mathcal{S}$. A *run* is a finite sequence of states such that: $\mathrm{s}_0 \in \mathrm{S}_0$ and $\mathrm{s}_i \xrightarrow{\alpha} \mathrm{s}_{i+1}$ for all $0 \leq \mathrm{i} < \mathrm{n}$. An *accepting run* is one where the last state (s_n) in its sequence is an accepting state.

The sequence of input symbols given to the finite automaton, starting from the initial state s_0, is "accepted", if there is an accepting run. Otherwise it is not accepted.

Let $\mathcal{M}$ be an NFA. $\mathcal{M}$ can produce a language, over the finite set Σ. The language is the set of all strings for which an accepting run by $\mathcal{M}$ exists and it is written as $\mathcal{L}(\mathcal{M})$. A set of strings is called regular if it is equal to $\mathcal{L}(\mathcal{M})$ for some $\mathcal{M}$.

Automaton $\mathcal{M}$ is called *deterministic* if $|S_0| \leq 1$ and $|\delta(s, \alpha)| \leq 1$ for all states $\mathrm{s} \in \mathcal{S}$ and $\alpha \in \Sigma$. (It is called *total* deterministic if $|S_0| = 1$ and $|\delta(s, \alpha)|$=1 for all states $\mathrm{s} \in \mathcal{S}$ and $\alpha \in \Sigma$).

Fig 1.1 shows an example of a nondeterministic finite automaton $\mathcal{M}$, such that: $\mathcal{S} = \{s_0, s_1, s_2\}$, $\Sigma = \{A, B\}$, $\mathrm{S}_0 = \{s_0\}$ and $\mathrm{F} = \{s_1\}$.

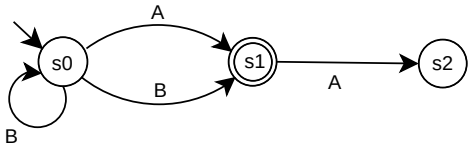

Figure 1.1: An example finite automaton $\mathcal{M}$.

For the transition function, we get the following definitions: $\delta(s_0, \text{A})=\{s_1\}$, $\delta(s_0, \text{B}) = \{s_0, s_1\}$, $\delta(s_1, \text{A}) = \{s_2\}$, $\delta(s_1, \text{B}) = \{\}$, $\delta(s_2, \text{A}) = \{\}$, $\delta(s_2, \text{B})=\{\}$. As a result, $\mathcal{L}(\mathcal{M}) = \{B^*A, B^*B\}$.

1.2.2 Communicating Finite Automata

A communicating automaton is an automaton which can communicate by synchronization on actions with other communicating automata through parallel composition of the whole. In this thesis, we will be using a communication scheme based on binary channels; a communication which takes place between two automata is called a *binary communication*.

The set of actions Σ consists of send and receive actions along the channels c $\in$ C, where C is the set of defined channels of the system. c! corresponds to the sending action over a channel c (state $\xrightarrow{c!}$state) and c? corresponds to the receiving action over c (state $\xrightarrow{c?}$state). For the sake of simplicity, we assume that for every channel c there is exactly one automaton $\mathcal{A}_i$ with c! $\in \Sigma_i$ and one automaton $\mathcal{A}_j$, j $\neq$ i, with c? $\in \Sigma_j$.

Let us define actions as $\alpha ::= a \mid \overline{a} \mid \tau$ where, considering a channel c, a is a synchronization action of the form c! (c?) and $\overline{a}$ denotes its complement c? (c!). τ represents an atomic internal execution step.

Consider two communicating automata $\mathcal{A}_1 = (\mathcal{S}_1, \Sigma_1, \delta_1, \text{S}_0{}^1, \mathcal{F}_1)$ and $\mathcal{A}_2 = (\mathcal{S}_2, \Sigma_2, \delta_2, \text{S}_0{}^2, \mathcal{F}_2)$ running in parallel. The *product* of the two communicating automata is a new automaton $\mathcal{B} = (\mathcal{S}_1 \times \mathcal{S}_2, \Sigma_1 \cup \Sigma_2, \delta', \text{S}_0^1 \times S_0^2, \mathcal{F}_1 \times \mathcal{F}_1)$ where δ' is defined as $(\mathcal{S}_1 \times \mathcal{S}_2) \times (\Sigma \cup \{\tau\}) \rightarrow 2^{\mathcal{S}_1 \times \mathcal{S}_2}$.

For all $s_1, s_1' \in \mathcal{S}_1$, $s_2, s_2' \in \mathcal{S}_2$ where a is an action, the following inference rules indicate the possible behavior of the transition function δ':

(i) $$\frac{s_1 \xrightarrow{a}_1 s_1' \qquad \overline{a} \notin \Sigma_2}{(s_1, s_2) \xrightarrow{a} (s_1', s_2)} \;,\; \frac{s_2 \xrightarrow{a}_2 s_2' \qquad \overline{a} \notin \Sigma_1}{(s_1, s_2) \xrightarrow{a} (s_1, s_2')}$$

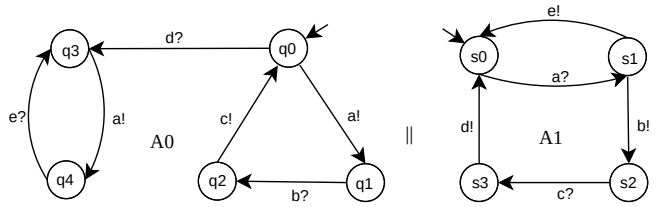

Figure 1.2: Example of two communicating processes modeled as automata $\mathcal{A}_0$ and $\mathcal{A}_1$.

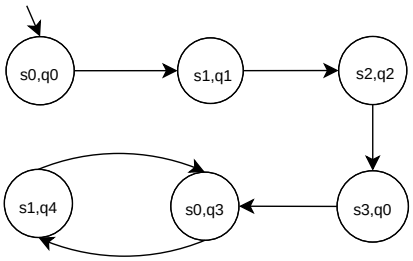

Figure 1.3: The product automaton of $\mathcal{A}_0$ and $\mathcal{A}_1$ - all transitions are τ-transitions.

(ii) $$\frac{s_1 \xrightarrow{\tau}_1 s_1'}{(s_1, s_2) \xrightarrow{\tau} (s_1', s_2)} \;,\; \frac{s_2 \xrightarrow{\tau}_2 s_2'}{(s_1, s_2) \xrightarrow{\tau} (s_1, s_2')}$$

(iii) $$\frac{s_1 \xrightarrow{a}_1 s_1' \qquad s_2 \xrightarrow{\overline{a}}_2 s_2'}{(s_1, s_2) \xrightarrow{\tau} (s_1', s_2')}$$

(i) and (ii) represent a one-sided transition step from either $\mathcal{A}_1$ or $\mathcal{A}_2$. (iii) corresponds to communication via *rendezvous*.

Fig.1.2 shows an example of two deterministic finite automata which can be composed in parallel; the corresponding product automaton is shown in Fig.1.3.

So far we defined finite automata and communications which may occur among two automata composed in parallel. Next, we will state the more general use of such automata defined in a system or a network of finite automata.

1.2.3 Networks of Finite Automata

It is convenient to describe a system as a parallel composition of automata. We have the same two kinds of transitions as before. In the first case, one of the components might do an internal action and the other would be when two components do an action together (synchronization step). Let N = $\mathcal{A}_0 \parallel \mathcal{A}_1 \parallel \mathcal{A}_2 \parallel ... \parallel \mathcal{A}_n$ be a network of finite automata, which are composed into one system (the $\parallel$ - operator is used to denote the parallel composition; it is associative and commutative). Each automaton is defined as $\mathcal{A}_i = (\mathcal{S}_i, \Sigma_i, \delta_i, \mathrm{s}_0^i, \mathcal{F}_i)$ (i= 1...n). We generalize the idea in section 1.2.2 to have the product $\mathcal{P}$ of all automata in the system where

$$\mathcal{P} = (\mathcal{S}_1 \times ... \times \mathcal{S}_n, \Sigma'_1 \cup ... \cup \Sigma'_n, \delta', \mathrm{S}_0^1 \times ... \times S_0^n, \mathcal{F}_1 \times ... \times \mathcal{F}_n)$$

where $\delta' : \mathcal{S}' \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^{\mathcal{S}'}$ is the transition function.

Similarly, as defined in section 1.2.2, we define the transition function δ' for the general case. Note that rendezvous may only occur between two automata in a system in each transition step.

1.2.4 Modeling a Network of Finite Automata - Example: Uppaal

There are several proposed modeling languages with which a system (or a network of finite automata) can be modeled such as Promela, UPPAAL modeling language,... etc. For this thesis, we chose the UPPAAL modeling language as an example. In UPPAAL [1], a system is modeled as a network of several extended timed automata[2] in parallel. It is designed mainly to check reachability and invariant properties by exploring the state space of a system. For motivation regarding the use of some examples which will rise up later in this work, there are only few things we need to know about the UPPAAL modeling language before we proceed, such as: How are systems defined in UPPAAL ?

First, note that the UPPAAL syntax used for declarations is similar to the syntax used in the C programming language. Fig. 1.5 shows an example.

Channels are declared as ***chan*** *c*. The synchronization label can be of the form c!, c? or an empty label.

[1] UPPAAL is a model checker for verification of real-time systems jointly developed by Uppsala University and Aalborg University. For more information on the UPPAAL project visit the official website under: `http://www.uppaal.com`

[2] A timed automaton is a finite-state machine, extended with clock variables, where clocks progress synchronously. Note: the time feature of automata will not be dealt with throughout this thesis.

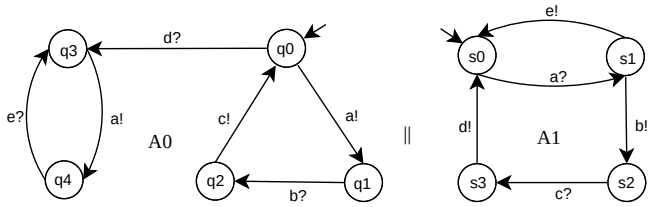

Figure 1.4: $\mathcal{A}_0$ and $\mathcal{A}_1$.

```
chan a, b, c, d, e;
process P0() {
state
    q4, q3, q2, q1, q0;
init q0;
trans
    q4 -> q3 { sync e?; ... },
    q3 -> q4 { sync a!; ... },
    q0 -> q3 { sync d?; ... },
    q2 -> q0 { sync c!; ... },
    q1 -> q2 { sync b?; ... },
    q0 -> q1 { sync a!; ... };
}

process P1() {
state
   s2, s3, s1, s0;
init s0;
trans
   s1 -> s0 { sync e!; ... },
   s3 -> s0 { sync d!; ... },
   s2 -> s3 { sync c?; ... },
   s1 -> s2 { sync b!; ... },
   s0 -> s1 { sync a?; ... };
}

system P0, P1;
```

Figure 1.5: The Uppaal-textual-version of a system with processes representing the finite automata shown in Fig.1.4, where P0 and P1 correspond to $\mathcal{A}_0$ and $\mathcal{A}_1$, respectively.

Process declaration begins with **process** which is followed by the name of the process and possibly some parameters. All other definitions of a process are declared inside an {}-block. Inside this block the states are declared as **state** *StateName* and the initial state as **init** *InitStateName*. Transitions in a process are declared as **trans** followed by *StateName* $\longrightarrow$ *StateName* and an {}-block including the labellings and probably some more information related to a transition; here we need only to distinguish the synchronization labellings which are declared as **sync** c! or **sync** c?. Finally, processes which must be composed into a system are declared as **system** followed by a list of processes. For more information on the UPPAAL modeling language see [GBY02] and [ADLly].

Now we can proceed to the next chapters; in this introductory chapter we defined communicating automata representing processes and the way they interact when composed together. Also we represented a modeling language which should assist in understanding how communicating processes are modeled in a system but also in understanding the experimental example (in the appendix) based on this modeling language.

2
Cycle Images For Finite Automata

In the first chapter, we have introduced how a system can be modeled (in Uppaal). For our approach, we need first to extract some information from a system model (for example by parsing the system configuration file which includes the system model in Uppaal code).

The first step of our approach, is to generate what we denoted as the cycle images of a finite automaton. A cycle image should include the count of each synchronization channel along a cycle. With the help of the cycle images we will be able to generate inequality constraints on *synchronization counters* (more about this in the next chapter).

Now to compute the cycle images, we will be involved in different sequential steps. A finite automaton can be represented as a directed graph where vertices of the graph represent states and directed edges represent the transition steps. For each finite automaton, we need to identify its *strongly connected components*; we need these to form one strongly connected component from all the strongly connected components which are reachable from the initial state (more in details later). For this formed strongly connected component, we apply an algorithm to detect cycles. Finally, we generate the cycle images and represent inequalities based on them in the following chapter; let's remind here that the first form of inequalities we want to generate have the form:

$$a_1\mu_1 + ... + a_n\mu_n \geq 0,$$

where n is the number of channels, μ_1,..., μ_n are the synchronization counter variables and a_1,..., a_n are positive coefficients (the values of the cycle images). This chapter will deal with computing these coefficients.

Note that we will use the example in Fig.1.2 as a running example to illustrate each step and the different manipulations that may take place by

applying some algorithms throughout this work. Note also that we will consider, for simplicity, processes which are represented as nondeterministic finite automata but have one initial state.

2.1 Identifying Strongly Connected Components

For all the finite automata in a system, the strongly connected components can be identified by applying Tarjan's algorithm. The vertices of a directed graph will get some identifiers, which are distinct natural numbers and will uniquely determine to which strongly connected component the vertex belongs. What are strongly connected components? How can we identify them?

Definition 2.1. *(Strongly Connected Component (SCC)). Let $\mathcal{G}$=($\mathcal{V}$, $\mathcal{E}$) be a directed graph, where $\mathcal{V}$ and $\mathcal{E}$ are the sets of vertices and edges of $\mathcal{G}$, respectively. A subgraph $\mathcal{D} \in \mathcal{G}$ is strongly connected if for every pair of vertices u, v $\in \mathcal{D}$ there is a path from u to v. Mutual reachability is an equivalence relation on $\mathcal{V}$ and its equivalence classes are called maximal strongly connected components.*

In Fig.2.1, automaton $\mathcal{A}_0$ has two SCCs, one of them containing the states q_3 and q_4; the other SCC containing q_0, q_1 and q_2. Automaton $\mathcal{A}_1$ forms one SCC.

Definition 2.2. *(Terminal Strongly Connected Component (TSCC)). A strongly connected component $\mathcal{G}'$=($\mathcal{V}'$, $\mathcal{E}'$) of a graph $\mathcal{G}$=($\mathcal{V}$, $\mathcal{E}$) is said to be terminal if there is no edge in $\mathcal{E}$ leading from a vertex s $\in \mathcal{G}'$ to a vertex s' $\notin \mathcal{G}'$.*

Notice in Fig.2.1, states q_3 and q_4 in automaton $\mathcal{A}_0$ and the edges between them form a terminal strongly connected component.

2.1.1 Tarjan's Algorithm

Tarjan's algorithm is one known elegant algorithm that finds the strongly connected components of a directed graph in $O(n + e)$ time, where n is the number of vertices and e is the number of edges of the input graph, i.e. it has time complexity that is linear in the size of the graph.

The basic idea of this algorithm is a depth-first search beginning from a start vertex. The algorithm can be considered to contain two interleaved traversals of the graph. The depth-first search traverses all edges and constructs a depth-first *spanning forest*. Once a so called *root* of a strongly connected component

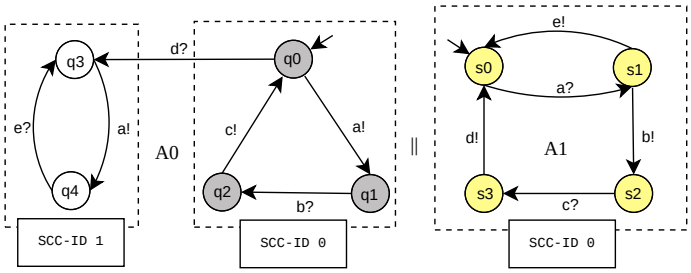

Figure 2.1: Automata $\mathcal{A}_0$ and $\mathcal{A}_1$ with distinguished SCCs.

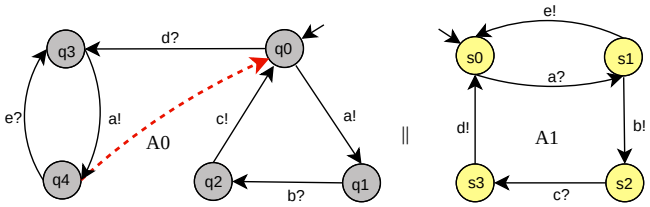

Figure 2.2: Automata $\mathcal{A}_0$ and $\mathcal{A}_1$ after combining the reachable SCCs.

is found, all its descendants that are not elements of components which were previously found are labeled as elements of this component. Another interleaved traversal uses a stack storing each vertex when being entered by the depth-first search. Once a root of a component is exited, all nodes down to the root are removed from the stack and they form the strongly connected component which corresponds to the root.

Tarjan's algorithm can be found in [Tar72] and some improved approaches of the algorithm in [NSS94].

2.2 Combining Reachable SCCs

After applying Tarjan's Algorithm to find the strongly connected components, each vertex (state) is labeled with an identifier (SCC-ID). Let $\mathcal{G}$=($\mathcal{V}$, $\mathcal{E}$) be a directed graph. An SCC-ID identifies to which SCC in $\mathcal{G}$ a vertex belongs (Fig.2.1).

The initial state of a finite automaton is represented as the starting vertex in the corresponding directed graph; we consider only having one initial state. The starting vertex belongs to an SCC $\mathcal{S}_0$. The SCC $\mathcal{S}_0 \subseteq \mathcal{G}$ needs to be combined with all reachable SCCs, starting from the initial state, to form one

single SCC. The reason behind this procedure is that combing the reachable SCCs (from the starting vertex) will insure having cycles (to be detected later) over all edges, which means that we would have inequalities that respect all possible interferences of synchronization actions over the whole structure. Note that unreachable SCCs are ignored. How can we combine the reachable SCCs?

The idea is to identify those SCCs which are terminal (TSCCs) by detecting SCCs from which any other SCC in $\mathcal{G}$ is not reachable. For each TSCC, we add a new edge from a representative of this TSCC to the starting vertex (we call its SCC the init-SCC). As a result, adding the edges leads to the formation of a new SCC which includes the starting vertex and combines the reachable parts to the Init-SCC.

Obviously, if only one reachable SCC is detected, i.e. the init-SCC, then no new edges need to be introduced to the directed graph and the SCC we want to extract is, in this case, the Initial-SCC itself.

Fig.2.2 shows for the example (which was introduced in the above sections) the detected TSCC in automaton $\mathcal{A}_0$ which gets a new edge; combining it to the Init-SCC.

2.3 Computation of the Minimum Circuit Basis

In the previous section, we have dealt with directed graphs and the idea of forming a strongly connected component considering the reachability from the starting vertex. Let the starting vertex of a graph $\mathcal{G}$ represent the initial state of a finite automaton having one initial state. The idea of combining all detected SCCs together, by adding edges from TSCCs to the Init-SCC, allows to extract information (i.e. the cycle images) over some cycles. For generating the cycle images, some questions will arise, such as: Which cycles should be taken into consideration? And how do we obtain the corresponding cycle images?

2.3.1 Basis Notations

Consider a directed graph $\mathcal{G}= (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ a set of directed edges, which is a set of ordered pairs of vertices, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. Let $e \in \mathcal{E}$, then $e = (u, v)$ (or also denoted by $u \rightarrow v$) we call u the *initial* vertex and v the *terminal* vertex of edge e. A *chain* $\mathbf{c}$ in $\mathcal{G}$ is a alternating sequence of vertices and edges defined as:

$$\mathbf{c} = (\mathrm{x}_0, \mathrm{e}_1, \mathrm{x}_1, \mathrm{e}_2, \mathrm{x}_2, ..., \mathrm{e}_{q-1}, \mathrm{x}_{q-1}, \mathrm{e}_q, \mathrm{x}_q)$$

	q_4	q_3	q_2	q_1	q_0
$q_4 \rightarrow q_3$	-1	1	0	0	0
$q_3 \rightarrow q_4$	1	-1	0	0	0
$q_0 \rightarrow q_3$	0	1	0	0	-1
$q_2 \rightarrow q_0$	0	0	-1	0	1
$q_1 \rightarrow q_2$	0	0	1	-1	0
$q_0 \rightarrow q_1$	0	0	0	1	-1
$\overbrace{q_4 \rightarrow q_0}^{new}$	-1	0	0	0	1

Table 2.1: Incidence matrix of $\mathcal{A}_0$

	s_3	s_2	s_1	s_0
$s_1 \rightarrow s_0$	0	0	-1	1
$s_3 \rightarrow s_0$	0	-1	0	1
$s_2 \rightarrow s_3$	-1	1	0	0
$s_1 \rightarrow s_2$	1	0	-1	0
$s_0 \rightarrow s_1$	0	0	1	-1

Table 2.2: Incid. matrix of $\mathcal{A}_1$

such that for all k, $e_k = (x_{k-1}, x_k)$ or $e_k = (x_k, x_{k-1})$ (e_k is called a forward or backward edge respectively), x_0 is the initial vertex and x_q is the terminal vertex of the chain.

Let's define the following:

- A chain is *closed* if its terminal vertex is the same as its initial vertex.
- A *simple* chain is one that does not contain the same edge twice.
- *Elementary* chains are chains where each vertex x appears only once in the sequence of a chain aside from the terminal/initial vertex.
- The length of a chain is determined by the number q of its edges.
- A *walk* is a chain in which each (directed) edge is a forward edge.
- A *path* is defined as a simple walk.
- A closed simple chain is a *cycle* and a closed path is a *circuit*.

2.3.2 The Cycle Space of a Directed Graph

Definition 2.3. *The incidence matrix $\mathcal{M}$ of a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an $(n \times e)$-matrix where n and e are the number of vertices and edges respectively; it consists of entries which are either $\mathcal{M}_{ij} = -1$ if $x_j \in \mathcal{E}$ leaves $s_i \in \mathcal{V}$ or if it enters $s_i \in \mathcal{V}$ then $\mathcal{M}_{ij} = +1$, and 0 otherwise.*

Tables 2.1 and 2.2 show the incidence matrices related to automata $\mathcal{A}_0$ and $\mathcal{A}_1$. We will only consider the incidence matrix of $\mathcal{A}_0$ as a running example for the following computations.

Definition 2.4. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. The cycle space $\mathcal{S}$ of $\mathcal{G}$ is defined as the subspace of $\Re^{|\mathcal{E}|}$ generated by the cycles of $\mathcal{G}$.*

Proposition 2.1. [GLS03] $v \in \mathcal{S} \Longleftrightarrow \mathcal{M}v = 0$

The cycle space of a strongly connected graph has a spanning set which consists of directed circuits. In the following, we introduce the notion of a *minimum circuit basis* and how to compute it. Later on the minimum circuit basis of a directed graph will be essential for the later computations, i.e. when we deal with the generation of cycle images.

2.3.3 Minimum Circuit Basis For Directed Graphs

Definition 2.5. *A circuit Basis is defined as a spanning set of the cycle space $\mathcal{S}$ of a graph $\mathcal{G}= (\mathcal{V}, \mathcal{E})$ consisting only of elementary circuits. The minimum circuit basis is a circuit basis with minimal length, where the length of a basis $\mathcal{B}$ is defined as:*

$$\mathcal{L}(\mathcal{B}) = \sum_{\mathcal{C}\in\mathcal{B}} |\mathcal{C}|$$

Proposition 2.2. [Ber85] A strongly connected directed graph has a circuit basis.

2.3.4 Algorithm on Finding the Minimum Circuit Basis of a Strongly Connected Component with Directed Edges

Before going into details, some basic definitions for the algorithm are needed:

Definition 2.6. *Let X_1,X_2, ...,X_n be n vectors. We say that X_1,X_2, ...,X_n are linearly dependent vectors iff there exist scalars c_1,c_2, ...,c_n, not all zero, such that*

$$c_1X_1 + c_2X_2 + \ldots + c_nX_n = 0$$

If such scalars do not exist, then the vectors are said to be linearly independent.

Definition 2.7. *The orthogonal complement of a subspace $\mathcal{S}$ of $\Re^n$ is the set of all vectors v in $\Re^n$ such that v is orthogonal to every vector in $\mathcal{S}$. Knowing*

that the span of a finite set of vectors is a subspace, the orthogonal complement of a matrix $\mathcal{H}$ which consists of a finite set of vectors (columns) is defined as the span of all the vectors that are orthogonal to the span of the vectors in $\mathcal{H}$. Let $\mathcal{H}^{\perp}$ denote the orthogonal complement of a matrix $\mathcal{H}$. We have then $\mathcal{H}^{\top}\mathcal{H}^{\perp}=0$.

Definition 2.8. *(Permuted Column Echelon Form - PCEF)A matrix $\mathcal{M}$ is said to be in its Permuted Column Echelon Form if it satisfies the following properties:*

- *Each non-zero column contains a 1 at a row i, all other entries of the row i are zero; we say, a column has an isolated entry*
- *All columns that contain only zeros are placed at the right end of the matrix, if they exist.*

The algorithm to compute the minimum circuit basis consists of finding the basis of the orthogonal complement of the incidence matrix and the elimination of negative entries in the obtained matrix, which results in is the minimum circuit basis of the input directed graph.

To compute the orthogonal complement of a matrix $\mathcal{M}$, which is supposed to be an incidence matrix of a strongly connected digraph, first we are going to transform the matrix into the *Permuted Column Echelon Form* and then construct the vectors (columns) of the basis $\mathcal{B}$ of the orthogonal complement.

I. Computing the Permuted Column Echelon Form.

The algorithm starts with an input matrix $\mathcal{M}$; we want to transform the matrix $\mathcal{M}$ into a matrix $\mathcal{M}'$ in PCEF. The PCEF of $\mathcal{M}$ can be computed by some column operations, i.e. for each column in $\mathcal{M}$, we detect an entry e=1 (or e=-1) and we want to isolate e in its row r; the column is added or subtracted to a finite set of columns in such a way to replace all the columns (but not the column of e) which have non-zero entires in r by columns which will have 0 entries in r. Columns having a negative isolated value (i.e. -1) instead of 1, can be multiplied by -1 in order to have the column in the desired form. The whole thing is done with sequential transformation steps considering each column until each column satisfies the desired property as stated in the definition of the PCEF.

Tables 2.3 and 2.4 shows the incidence matrix $\mathcal{M}_0$ of the digraph corresponding to automaton $\mathcal{A}_0$ (Fig. 2.3) and the PCEF $\mathcal{M}'_0$ of $\mathcal{M}_0$.

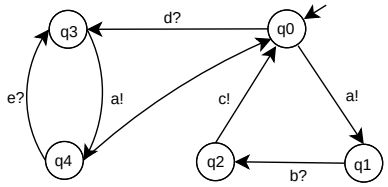

Figure 2.3: Automaton $\mathcal{A}_0$ (with the new edge $q_4 \rightarrow q_0$).

$q_4 \rightarrow q_3$	-1	1	0	0	0
$q_3 \rightarrow q_4$	1	-1	0	0	0
$q_0 \rightarrow q_3$	0	1	0	0	-1
$q_2 \rightarrow q_0$	0	0	-1	0	1
$q_1 \rightarrow q_2$	0	0	1	-1	0
$q_0 \rightarrow q_1$	0	0	0	1	-1
$q_4 \rightarrow q_0$	-1	0	0	0	1

Table 2.3: Incidence matrix $\mathcal{M}_0$ of $\mathcal{A}_0$

$q_4 \rightarrow q_3$	[1]	0	0	0
$q_3 \rightarrow q_4$	-1	0	0	0
$q_0 \rightarrow q_3$	0	[1]	0	0
$q_2 \rightarrow q_0$	0	0	[1]	0
$q_1 \rightarrow q_2$	0	0	-1	-1
$q_0 \rightarrow q_1$	0	0	0	[1]
$q_4 \rightarrow q_0$	1	-1	0	0

Table 2.4: PCEF of $\mathcal{M}_0$

II. Computing a Basis of the Orthogonal Complement of the Incidence Matrix.

After computing the PCEF $\mathcal{M}'$ of $\mathcal{M}$, we perform the following computations to output a basis $\mathcal{B}$ of the orthogonal complement of $\mathcal{M}$ (Note that: zero columns in $\mathcal{M}'$ can be eliminated from the matrix):

Let r_1, ..., r_k be the rows in matrix $\mathcal{M}'$, which are not those of the obtained isolated entries. For the construction of the basis vectors of the orthogonal complement, k vectors need to be constructed. Each of these k vectors is associated to one of the rows r_1, ...or r_k in matrix $\mathcal{M}'$; which means that the associated vector to r_m $(1 \leq m \leq k)$ contains the entry 1 at the column index where r_m was detected and the entries at column indices which are the same as the indices of the rows r_x for all $1 \leq x \leq k$ and x $\neq$ m, become all zeros.

To complete the construction of the vectors of the basis of the orthogonal complement of $\mathcal{M}$, all other indices of the basis vectors (i.e. those which are the same as the indices where the isolated elements were detected) get their values as follows:

If in a row i, column j has an isolated entry c (i.e. 1) and in row r_m

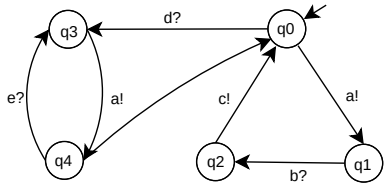

Figure 2.4: Automaton $\mathcal{A}_0$ (with the new edge $q_4 \rightarrow q_0$).

$(1 \leq m \leq k)$ column j has a value d then the basis vector which has the entry 1 in row r_m, gets the value $-(d/c)$ at the index i, knowing that c has the value 1.

Lemma 2.1: The above computational steps form a basis of the space of all the vectors which are orthogonal to the vectors (columns) of $\mathcal{M}$. The resulting matrix $\mathcal{B}$ is a basis of the orthogonal complement of the matrix $\mathcal{M}$.

Proof. The first step where the input matrix is transformed into its PCEF includes a finite sequence of column operation (addition/subtraction of columns); the vector space stays the same in $\mathcal{M}'$ as in $\mathcal{M}$. Then vectors (columns) were constructed in such a way to have each of them orthogonal to all the vectors in $\mathcal{M}'$; i.e. the dot product of a constructed column and any column in $\mathcal{M}'$ must result in 0. Following Proposition 2.1, we know that these vectors are in the cycle space of the directed graph of $\mathcal{M}$. Because the vector space does not change between $\mathcal{M}$ and $\mathcal{M}'$, we have that any constructed vector is also orthogonal to the columns of $\mathcal{M}$. Knowing that each constructed vector is orthogonal to all the columns of $\mathcal{M}$ and notice that the algorithm constructs a complete set of vectors for a basis, then these vectors form a basis of the orthogonal complement of the matrix $\mathcal{M}$.

❑

Table 2.5 shows the resulting orthogonal complement $\mathcal{B}$ of $\mathcal{M}_0$ corresponding to $\mathcal{A}_0$ in Fig. 2.4.

Proposition 2.3. [DMC94] *Let $\mathcal{B}$ be a basis of a vector space $\mathcal{S}$. If any vector v in $\mathcal{B}$ is replaced by the sum of v and a linear combination of the vectors in $\mathcal{B} - \{v\}$, then the resulting set of vectors is again a basis of $\mathcal{S}$.*

$q_4 \rightarrow q_3$	1	0	-1
$q_3 \rightarrow q_4$	1	0	0
$q_0 \rightarrow q_3$	0	0	1
$q_2 \rightarrow q_0$	0	1	0
$q_1 \rightarrow q_2$	0	1	0
$q_0 \rightarrow q_1$	0	1	0
$q_4 \rightarrow q_0$	0	0	1

Table 2.5: Matrix $\mathcal{B}$

III. Eliminating Negative Entries.

The vectors in matrix $\mathcal{B}$ may still contain negative entries, i.e. -1's. An entry of the value -1 refers to a backward edge. We are searching for circuits, this means, for every vector in $\mathcal{B}$ we have to eliminate the negative entries because we want to avoid having different directions indicated by the vector (i.e. only forward edges) to possibly obtain a circuit. Elimination of these entries must occur by linear combinations between some vectors. The resulting vectors will form a minimum circuit basis.

To get rid of the negative entries (-1's), we do the following:

For all columns in matrix $\mathcal{B}$, as long as negative entries exist, select a row at index i with negative entries; for all columns v_j with negative entries $-c_j$ in the row at index i and all columns v_k with positive entries d_k also in row index i compute the linear combination of v_j and v_k ($s_{jk} = c_j * v_k + d_k * v_j$), which has a zero entry in row i. Substitute the columns v_j by the linear combinations s_{jk}. If the selected row has m negative and n positive entries, then m columns will be substituted by $m*n$ columns, thats why it is more efficient to always select a row at index i so that $(m*n) - n$ is minimal. Columns which contain negative entries, but no positive entries in the same row of the negative entry in the matrix exist are eliminated. We denote the resulting matrix as $\mathcal{B}'$.

Lemma 2.2: *It is always possible to get rid of negative entries in the matrix for which negative entries are to be eliminated after step (II).*

Proof. First, linear combinations only take place when having a positive entry (namely 1) at the same row index of the negative elements. So, if such a positive entry exists then after a finite number of linear combination with the same vector (column) the negative entry will be replaced by a the entry 0.

$q_4 \rightarrow q_3$	1	0	0
$q_3 \rightarrow q_4$	1	0	1
$q_0 \rightarrow q_3$	0	0	1
$q_2 \rightarrow q_0$	0	1	0
$q_1 \rightarrow q_2$	0	1	0
$q_0 \rightarrow q_1$	0	1	0
$q_4 \rightarrow q_0$	0	0	1

Table 2.6: Matrix $\mathcal{B}'$

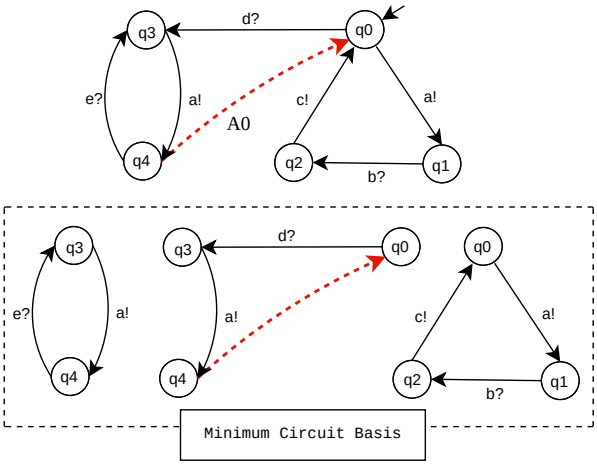

Figure 2.5: Automata $\mathcal{A}_0$ and its minimum circuit basis.

Also when having a situation where no positive entries in the same row of the negative entry is detected, then the column is simply eliminated.

❑

With these steps the minimum circuit basis can be obtained by the help of these resulting vectors (columns), which indicate the existence (when having a non-zero entry) of an edge in a circuit; the edges indicated by the positive entries in each resulting vector construct when combined together a circuit.

Table 2.6 depicts the matrix consisting of the vectors of the minimum circuit basis of automaton $\mathcal{A}_0$ - see also Fig.2.5.

Why does this algorithm work? On finding the PCEF of a matrix $\mathcal{M}$, notice that the entries in each row of the input matrix contain a 1 and a -1 and all the other values are 0's. Let's recall that for each column, we detect an entry e=1 (or e=-1) and we want to isolate e in its row r; the column is added or subtracted to a finite set of columns in such a way to replace all the columns (but not the column of e) which have non-zero entires in r by columns which will have 0 entries in r. We notice that it is always possible, after some finite transformation steps, to isolate the entry e. For the next columns, the same follows and it is important to note that the rows which were taken into consideration before and which got isolated elements will not be affected by transformation steps which will follow. This implies that the algorithm will always find a PCEF after some finite steps. Also note that performing column operations does not change the column space.

The idea of constructing the basis of the orthogonal complement of a matrix $\mathcal{M}'$ will also always terminate as the number of vectors to be constructed cannot exceed the number of rows in $\mathcal{M}'$ according to the algorithm and the values are determined by having the dot product of a constructed vector with a vector in $\mathcal{M}$ or $\mathcal{M}'$. About the existence of negative entries the algorithm performs some finite steps to have these entries eliminated and to get a basis of vectors which are still orthogonal to vectors in $\mathcal{M}$ and $\mathcal{M}'$.

Lemma 2.3: *The generated circuits form together a circuit basis of the input directed graph.*

Proof. According to Lemma 2.1, from the input incidence matrix a basis of the orthogonal complement for this matrix is constructed. Later on negative entries were eliminated from such a matrix by a finite number of linear combinations. From Proposition 2.3 it follows that the set of vectors in the resulting matrix is again a basis of the vector space, i.e. a circuit basis because only forward edges are included and this fact is based on Lemma 2.2.

❑

The first two steps of the algorithm take polynomial time but for the third step we studied a worst-case where the computation may grow exponentially in the number of the columns. Transforming the input matrix into its PCEF is bounded by $\mathcal{O}((n-1)nm)$ where n is the number of vertices (columns) and m is the number of edges (rows). The second step, where a basis of the orthogonal complement of the input matrix is constructed, i.e. k vectors in the size of a column are constructed (k is the number of rows with no isolated elements)

and so the whole computation is bounded by $\mathcal{O}(mk)$ (Note: we assumed that we know the row indices of isolated entries from the first step).

For the final step, namely the elimination of negative entries, detecting the negative entries considers the whole matrix and detection must also consider the columns resulting from the linear combinations; we have that for each negative entry e in column c and in a row r, we replace c by a set of linear combinations with at most $n-1$ columns where n is the number of columns. There is a case where the generation of vectors resulting from the linear combinations will grow exponentially because the linear combinations will also get positive values which need to be considered in the steps to follow where we have to generate new columns formed by linear combinations.

2.3.5 Cycle Images For Circuits

The circuit vectors contained in the minimum circuit basis are the circuits for which we construct the cycle images. Consider the edges of $\mathcal{G}$ and check for the existence of each edge in a circuit, this is when the circuit vector has a (non-zero) positive value at the corresponding index, a channel counter is then incremented by the detected positive value at the index which is mapped to the corresponding channel in a channel counter array.

In other words, we are counting distinguished synchronizations over circuits contained in the minimum circuit basis and for each of these circuits a cycle image is generated.

Table 2.8 depicts the resulting cycle images for automaton A_0; c_1, c_2 and c_3 are cycle images corresponding to the vectors v_1, v_2 and v_3 in Table 2.7, respectively.

In this chapter we computed the cycle images of circuits in the minimum circuit basis of the directed graph which corresponds to a finite automaton. We are now ready to illustrate in the next chapter how we are going to generate inequality constraints on synchronization counters with the help of the obtained cycle images.

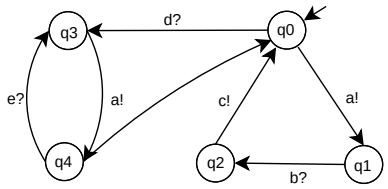

Figure 2.6: Automaton $\mathcal{A}_0$ (with the new edge $q_4 \to q_0$).

	v_1	v_2	v_3
$q_4 \xrightarrow{e?} q_3$	1	0	0
$q_3 \xrightarrow{a!} q_4$	1	0	1
$q_0 \xrightarrow{d?} q_3$	0	0	1
$q_2 \xrightarrow{c!} q_0$	0	1	0
$q_1 \xrightarrow{b?} q_2$	0	1	0
$q_0 \xrightarrow{a!} q_1$	0	1	0
$q_4 \xrightarrow{-} q_0$	0	0	1

Table 2.7: Matrix $\mathcal{B}'$

	c_1	c_2	c_3
$a \mapsto 0$	1	1	1
$b \mapsto 1$	0	1	0
$c \mapsto 2$	0	1	0
$d \mapsto 3$	0	0	1
$e \mapsto 4$	1	0	0

Table 2.8: Cycle images for $\mathcal{A}_0$

3

Synchronization Counters And Inequality Constraints

In the previous chapter, we have dealt with detecting circuits in a directed graph corresponding to a finite automaton and the computation of cycle images associated to those circuits. Now we are ready to generate inequality constraints on the synchronization counters with the help of the cycle images. Next, we would then explore the states of the corresponding finite automaton, to define inequality constraints for each state in each communicating process.

The inequality constraints will have the form:

$$\mu'_1 \mathrm{x}_1 + ... + \mu'_n \mathrm{x}_n \geq k_q$$

where μ'_1,..., μ'_n are values which we will compute using the cycle images, x_1,..., x_n are channel names (n is the number of channels) and k_q stands for a value depending on a state q.

How to generate the general inequalities? And how to define the inequality constraints for states? Before answering these questions, we will first consider some definitions and basic notations.

3.1 Some Definitions

Definition 3.1. *(Convex Polyhedron). A convex polyhedron or simply a polyhedron $\mathcal{P}$ in $\Re^d$ is the set of solutions to a (finite) system of linear inequalities in d-variables: $\mathcal{P} = \{x \in \Re^d : \mathcal{M}x \leq b\}$ where $\mathcal{M} \in \Re^{m \times d}$ and $b \in \Re^m$.*

Note that, for the case d=2, $\mathcal{P}$ is said to be a *convex polygon.*

Definition 3.2. *(Convex Combination). A convex combination of a set of vectors (points) $\{v_1, v_2, ..., v_n\}$ is an expression of the form:*

$$y = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$

where $\alpha_1 + \alpha_2 + \ldots + \alpha_n = 1$ *and* $\alpha_i \geq 0 \; \forall \; 1 \leq i \leq n$.

Definition 3.3. *(Convex Hull). The convex hull of a set* $\mathcal{X}$ *is the set of convex combinations of all points in* $\mathcal{X}$*. It is the minimal convex set containing* $\mathcal{X}$*.*

Definition 3.4. *(Convex Cone). A convex cone* $\mathcal{K}$ *is a subset of a vector space that is closed under linear combinations with positive coefficients.*

According to the *Farkas-Minkowski-Weyl theorem* [Sch86] a convex cone is polyhedral if and only if it is finitely generated.

Definition 3.5. *(Extreme Points). A point* u *in a polyhedron* $\mathcal{P}$ *is said to be an extreme point of* $\mathcal{P}$ *if there do not exist two points* u_1 *and* u_2 in $\mathcal{P}$, such that, $u = \lambda \; u_1 + (1 - \lambda) u_2$, where $0 < \lambda < 1$ and $u_1 \neq u_2$.

Definition 3.6. *(Ray). Let* $\mathcal{K} = \left\{ r \in \Re^d : \mathcal{M} r \geq 0 \right\}$ *. Any* r *in* $\mathcal{K} \setminus \{0\}$ *is a ray of* $\mathcal{K}$*.*

Definition 3.7. *(Extreme Rays). An extreme ray is a ray* r *in* $\mathcal{K}$ *for which there do not exist rays* r_1 *and* r_2 *in* $\mathcal{K}$ *with* $r_1 \neq \lambda r_2$ *for any* $\lambda > 0$*, such that,* $r = \mu \; r_1 + (1\text{- } \mu) r_2$*, where* $0 < \mu < 1$*.*

Definition 3.8. *(Conic Hull). Let* $\mathcal{D} \subseteq \Re^N$ *then the conic hull of* $\mathcal{D}$ *is a union of all rays through the origin intersecting* $\mathcal{D}$*.*

The *Minkowski-Weyl theorem* [Sch86] states that every convex polyhedron $\mathcal{P}$ has two representations, one as the intersection of a finite set of half spaces, denoted by the **H-representation** of the convex polyhedron; formally, for some real (finite) matrix $\mathcal{M}$ and real vector b (b $\in \Re^m$),

$$\mathcal{P} = \{x : \mathcal{M} x \leq b\},$$

and the other representation, known as the **V-representation** of a convex polyhedron, is defined as the *Minkowski sum* of the convex hull of a finite set of vectors u_1,..., u_s and the conic hull of finitely many directions in $\Re^d$:

$$\mathcal{P} = conv\{u_1, ..., u_s\} + cone\{r_1, ..., r_q\}$$

where $\{u_1, ..., u_s\} \subseteq \Re^d$ is the set of *vertices* or *extreme points* of $\mathcal{P}$, $\{r_1, ..., r_q\} \subseteq \Re^d$ is the set of *extreme directions*(rays) of $\mathcal{P}$, and

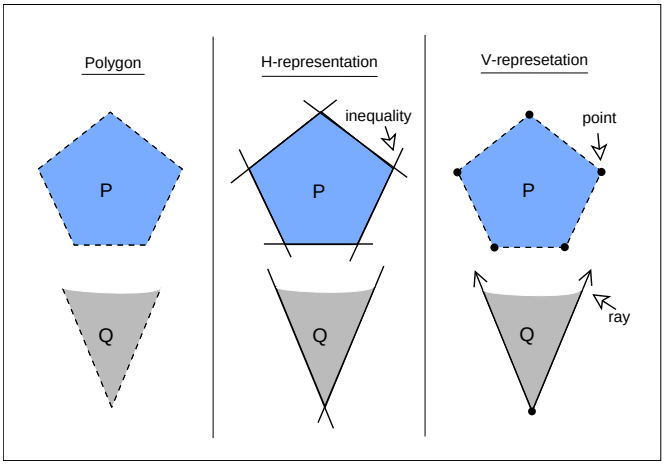

Figure 3.1: An example of two polygons in H- and V-representations.

$$conv\{u_1, ..., u_s\} = \{\sum_{i=1}^{s} \lambda_i u_i : \sum_{i=1}^{s} \lambda_i = 1, \lambda_1 \geq 0, ..., \lambda_s \geq 0\},$$
$$cone\{r_1, ..., r_q\} = \{\sum_{i=1}^{q} \mu_i r_i : \mu_1 \geq 0, ..., \mu_q \geq 0\}.$$

Fig. 3.1 shows an example of two polygons and the corresponding H- and V-representations.

At this point, we notice that the values in the cycle images, which were discussed in chapter 2, are to be coefficients a_1,..., a_n of inequalities of the form:

$$a_1\mu_1 + ... + a_n\mu_n \geq 0$$

where n is the number of channels, μ_1,..., μ_n are some variables. The set of these inequalities (half spaces), form a polyhedron, i.e. a polyhedral cone, and denote its H-representation.

3.2 Computation of Coefficients of the Inequality Constraints with lrs

lrs [Avi] is a C program which converts an H-representation of a polyhedron to a V-representation or vice versa. It is based on the *reverse search algorithm* of *Avis and Fukuda*[AF92], which uses simplex method with systematic search over sequence of bases; we refer to Avis [Avi99b] for a technical description, and [Avi99a] for some computational experience.

	c_1	c_2	c_3
a $\mapsto$ 0	1	1	1
b $\mapsto$ 1	0	1	0
c $\mapsto$ 2	0	1	0
d $\mapsto$ 3	0	0	1
e $\mapsto$ 4	1	0	0

Table 3.1: Synchronization counters for $\mathcal{A}_0$.

For our approach, we need to convert an H-representation into a V-representation; this is known as *the vertex enumeration problem*. The H-representation is given in an input format for lrs, it contains a list of inequalities, i.e. their coefficients.

The input files, which were developed for lrs in a proposed polyhedra format, are constructed as follows:

```
H-representation
begin
 m n integer

 {list of inequalities}

end
```

where m is the number of inequalities and n is the number of coefficients of each inequality.

Considering the example of the automaton $\mathcal{A}_0$, the constructed cycle images (Table 3.1) are the vectors $c_1 = (1\ 0\ 0\ 0\ 1)^T$, $c_2 = (1\ 1\ 1\ 0\ 0\ )^T$ and $c_3 = (1\ 0\ 0\ 1\ 0)^T$; thus, we obtain the following inequalities $\mathcal{I}_1$, $\mathcal{I}_2$ and $\mathcal{I}_3$ for the vectors c_1 ,c_2 and c_3, respectively:

$\mathcal{I}_1$ **:** $\mu_0 + \mu_4 \geq 0$

$\mathcal{I}_2$ **:** $\mu_0 + \mu_1 + \mu_2 \geq 0$

$\mathcal{I}_3$ **:** $\mu_0 + \mu_3 \geq 0$

The H-Representation formed by the values of the cycle images for $\mathcal{A}_0$ is then given in the following input form:

```
H-representation
begin
  3    5    integer
 1   0   0   0   1
 1   1   1   0   0
 1   0   0   1   0
end
```

lrs computes the V-representation of the input and outputs the following:

```
V-representation
begin
  ***** 5 rational
 0 -1  1  0  0
 1 -1  0 -1 -1
 0  1  0  0  0
 0  0  0  0  1
 0  0  0  1  0
end
```

The output in V-representation shows that the polyhedron has two vertices (0,−1,1,0,0) and (1,−1,0,−1,−1) and three extreme rays (0,1,0,0,0), (0,0,0,0,1) and (0,0,0,1,0).

We need these values as coefficients of inequalities, which we are going to generate.

3.3 Generation of Inequality Constraints For States

The values which we computed in the above section are the the coefficients μ'_1,..., μ'_n which we need for the inequalities which have the form:

$$\mathcal{I}'_i : \mu'_1 \mathrm{x}_1 + ... + \mu'_n \mathrm{x}_n \geq k_q,$$

where n is the number of channels, m the number of inequalities, z the number of states (vertices) in the finite automaton , x_1,..., x_n are channels, k_q stands for a value depending on a state q.

After generating the left-hand-side part of these inequalities we are going to associate inequality constraints for each state of a finite automaton. We determine the right-hand-side value (k_q) of the inequalities and what we get are the inequality constraints in the following form:

$$\mathcal{I}'_{i,j} : \mathcal{E}_i \geq k_{i,j} \text{ in } \mathrm{s}_j,$$

where $\mathcal{E}_i$ is an expression of the form $\mu'_1 \mathrm{x}_1 + ... + \mu'_n \mathrm{x}_n$ (see first paragraph), s_j is a vertex in the input graph, and for all $1 \leq i \leq m$ and $0 \leq j \leq$ *z-1*.

Let $\mathcal{G}$=($\mathcal{V}$,$\mathcal{E}$) be the input graph. To compute the $k_{i,j}$ for a related inequality $\mathcal{I}'_i$, the algorithm begins by initializing all vertices in the input graph G with temporary labels; the starting vertex is labeled by 0, all the others are labeled by a value which must be large enough that, when compared with any value during the algorithm run later, it must always be greater (for example let it be $+\infty$). The reason behind this is related to the next step of the algorithm, where vertices will get minimal values.

Now we can apply a procedure, namely explore(u, $\mathcal{I}'$), which takes at the beginning the starting vertex and an inequality $\mathcal{I}'$ as arguments; its implementation is shown in the pseudocode below. In the following we define some functions which appear in the pseudocode:

- **label(*e*):** returns the label of an edge *e*.
- **getValue(*u*):** returns a value currently assigned to a vertex *u*.
- **setValue(*h*, *u*):** assigns a new value *h* to a vertex *u*.
- **terminalVertex(*e*):** returns the terminal vertex of an edge *e*.
- **inequalityCoefficient(*l*, $\mathcal{I}'$):** returns the coefficient corresponding to an edge label *l* in an inequality $\mathcal{I}'$.

procedure $explore(u, \mathcal{I}')$

1: **for all** outgoing edges e of u **do**
2: $c \leftarrow inequalityCoefficient(label(e), \mathcal{I}')$
3: $S \leftarrow getValue(u) + c$
4: $x \leftarrow terminalVertex(e)$
5: **if** $S < getValue(x)$ **then**
6: $setValue(S, x)$
7: $explore(x, \mathcal{I}')$
8: **end if**
9: **end for**

As obvious, explore(u, $\mathcal{I}'$) performs a graph search starting from the starting vertex for which it takes $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$ time to visit each vertex once; this is always the case as the algorithm starts at the starting vertex and all the other vertices $v \in \mathcal{V}$ are labeled with $+\infty$ values which makes reduction of these

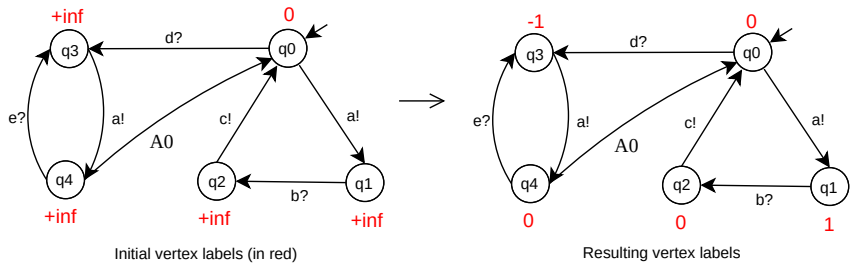

Figure 3.2: $\mathcal{A}_0$ and the specifications for inequality $\mathcal{I}'_2$: a - b - d - e $\geq k_{2,j}$ (Note: the label +inf means plus infinity).

values always possible during the first visit, this means also that there exists for every vertex $\in \mathcal{V}$ a path from the staring vertex which will map a minimal value to a vertex. Moreover, consider also the (update) steps when applying explore(x, $\mathcal{I}'$) on a vertex x (pseudocode $\rightarrow$ line 7) which can still get smaller values after being visited by another path. As it is possible that all vertices can still get minimal values and we know that the input graph is an SCC, each update step also takes $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$ time in a worst case. Knowing that, there exists a path to any vertex x' where x' will get a minimal value, so the number of possible update steps for x' is limited to the number of possible paths from the starting vertex leading to it where x' occurs only once in each path.

Remember that we should apply the algorithm for each non-specified inequality and perform the represented algorithm to obtain for each vertex (state) an associated set of inequality constraints.

For the example automaton $\mathcal{A}_0$ the following non-specified inequalities $\mathcal{I}'_i$ are constructed:

$\mathcal{I}'_1$: c - b $\geq k_{1,j}$

$\mathcal{I}'_2$: a - b - d - e $\geq k_{2,j}$

$\mathcal{I}'_3$: b $\geq k_{3,j}$

$\mathcal{I}'_4$: e $\geq k_{4,j}$

$\mathcal{I}'_5$: d $\geq k_{5,j}$

The following inequality constraints for the states q_j are obtained:

$$\mathcal{I}'_{1,j} : c - b \geq \begin{cases} 0 & \text{in } q_0 \\ 0 & \text{in } q_1 \\ -1 & \text{in } q_2 \\ 0 & \text{in } q_3 \\ 0 & \text{in } q_4 \end{cases} \tag{3.1}$$

$$\mathcal{I}'_{2,j} : a - b - d - e \geq \begin{cases} 0 & \text{in } q_0 \\ 1 & \text{in } q_1 \\ 0 & \text{in } q_2 \\ -1 & \text{in } q_3 \\ 0 & \text{in } q_4 \end{cases} \tag{3.2}$$

$$\mathcal{I}'_{3,j} : b \geq \begin{cases} 0 & \text{in } q_0 \\ 0 & \text{in } q_1 \\ 1 & \text{in } q_2 \\ 0 & \text{in } q_3 \\ 0 & \text{in } q_4 \end{cases} \tag{3.3}$$

$$\mathcal{I}'_{4,j} : e \geq \begin{cases} 0 & \text{in } q_0 \\ 0 & \text{in } q_1 \\ 0 & \text{in } q_2 \\ 0 & \text{in } q_3 \\ 0 & \text{in } q_4 \end{cases} \tag{3.4}$$

$$\mathcal{I}'_{5,j} : d \geq \begin{cases} 0 & \text{in } q_0 \\ 0 & \text{in } q_1 \\ 0 & \text{in } q_2 \\ 1 & \text{in } q_3 \\ 1 & \text{in } q_4 \end{cases} \tag{3.5}$$

Fig.3.2 shows for $\mathcal{A}_0$ and the inequality $\mathcal{I}'_2$: a - b - d - e $\geq k_{2,j}$ the initial labels and the resulting labels after performing the exploration algorithm.

For the running example we showed how the the computations are performed on a single process in a system. For sure we have to perform the computations on all the processes of a system, and the resulting inequalities which are viewed as inequality constraints allow under circumstances, for example, the knowledge on if a state in the considered system is not reachable.

For instance, consider an inequality $\mathcal{I}'$ for a process $\mathcal{P}_i$ specified for a state z_i in $\mathcal{P}_i$ in a system $\mathcal{R}$ and another inequality $\mathcal{I}''$ for a process $\mathcal{P}_j$ specified for a state z_j in $\mathcal{P}_j$ also in $\mathcal{R}$; then if $\mathcal{I}'$ and $\mathcal{I}''$ are incompatible, we know that any state in the state space of $\mathcal{R}$ which in its composition includes both basic states z_i and z_j is not a reachable state in $\mathcal{R}$.

We can also generalize this idea for a set of states in $\mathcal{P}'$ satisfying a common inequality constraint $\mathcal{I}'$ to be considered with a set of states in $\mathcal{P}''$ that have an incompatible inequality constraint $\mathcal{I}''$.

Now we can remind about the example which was stated in the introductory chapter; let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two processes and a, b be the defined channels in a system $\mathcal{R}'$ and suppose there exist a state z_1 in $\mathcal{P}_1$ and z_2 in $\mathcal{P}_2$ for which inequality constraints were obtained by our approach, such that:

$$\mathcal{I}_1\colon\ a - b \geq 2$$
$$\mathcal{I}_2\colon\ b - a \geq -1$$

$\mathcal{I}_2$ can be written as $a - b \leq 1$ ($\mathcal{I}_1$ and $\mathcal{I}_2$ are inequality constraints for z_1 and z_2, respectively); then we know that any state in the state space of $\mathcal{R}'$ which is composed of the basic process states z_1 and z_2 is not reachable because $\mathcal{I}_1$ and $\mathcal{I}_2$ are incompatible.

In this chapter we showed how inequality constraints on synchronization counters can be generated after having computed the cycle images; first we constructed inequalities having the values in cycles images as coefficients and put together they formed the H-representation of a polyhedron, then we transformed this H-representation the V-representation of the polyhedron. General inequalities were then constructed out of which we derived inequality constraints for each state in an automaton. We then showed how one could deduce information concerning reachability in a system.

4
Conclusion

4.1 Final Thoughts

It is known that it is sometimes impossible, to a certain extent, to explore the entire state space of concurrent systems with limited time and memory; based on the state space explosion problem. We succeeded in this approach to collect information regarding reachability without having to explore the whole state space of a system. Thus dealing with the problem the way our approach deals with it is a more efficient and successful.

The idea was the generation of inequality constraints on synchronization counters. More precisely, we considered each single process of a system, generated constraints on the synchronization counters for a process, and then compared these constraints together in such a way to obtain some reachability analysis; this is related to the interaction which occurs between the processes of a system.

The valuable information about the defined processes of a system which this thesis provided leads to the derivation of some kind of a system abstraction defined by inequality constraints on the synchronization counters of the processes of a system.

4.2 Further Work

Possible extensions of this thesis could deal with how to make use of the obtained results; dealing with these efficiently and deriving new ideas of how some properties of interest in relation to the reachability problem can be checked. Let us also note here that the algorithm which was used by this approach to find the minimum circuit basis had a problem denoted by the worst case

computation time which happened to be exponential; finding a more efficient algorithm for computing the minimum circuit basis (being an essential part of this approach) is also one possible future work.

A
Experimental Results

The implementation for this approach was developed in C++. The program is a tool which gets an input file in Uppaal format (.xta), parses the file, performs the computations which were discussed throughout this thesis, and outputs the results (i.e. the inequality constraints on the synchronizations for each defined process in a system). The program displays the results of almost each computational step.

We tested the program with various system models; an example execution of the program on a system model defining four processes results in the following:

Input file: input.xta (see next page)

- INPUT FILE BEGINS HERE -

```
// -------------------------------------------------------------
// CSMA/CD 3
// Carrier Sense, Multiple-Access with Collision Detection
// automatically generated by script genCSMA_CD.awk
// M. Oliver Moeller <omoeller@brics.dk>
// Wed Sep 19 11:48:20 2001
// -------------------------------------------------------------
chan begin, end, busy, cd1, cd2, cd3;

process P0 {
clock x;
state bus_idle, bus_active, bus_collision1{ x < 26 },
bus_collision2{ x <= 0 }, bus_collision3{ x <= 0 };
init  bus_idle;
trans
bus_idle -> bus_active { sync begin ?; assign x:= 0; },
bus_active -> bus_idle { sync end ?; assign x:= 0; },
bus_active -> bus_active { guard x >= 26; sync busy !; },
bus_active -> bus_collision1 { guard x < 26; sync begin ?; assign x:= 0; },
bus_collision1 -> bus_collision2 { guard x < 26; sync cd1 !; assign x:= 0; },
bus_collision2 -> bus_collision3 { guard x <= 0; sync cd2 !; assign x:= 0; },
bus_collision3 -> bus_idle { guard x <= 0; sync cd3 !; assign x:= 0; };
}

process P1 {
clock x;
state sender_wait, sender_transm{ x<= 808}, sender_retry{x < 52};
init sender_wait;
trans
sender_wait -> sender_transm { sync begin !; assign x:= 0; },
sender_wait -> sender_wait { sync cd1 ?; assign x:= 0; },
sender_wait -> sender_retry { sync cd1 ?; assign x:= 0; },
sender_wait -> sender_retry { sync busy ?; assign x:= 0; },
sender_transm -> sender_wait { guard x == 808; sync end !; assign x:= 0; },
sender_transm -> sender_retry { guard x < 52; sync cd1 ?; assign x:= 0; },
sender_retry -> sender_transm { guard x < 52; sync begin !; assign x:= 0; },
sender_retry -> sender_retry { guard x < 52; sync cd1 ?; assign x:= 0; };
}

process P2 {
clock x;
state sender_wait, sender_transm{ x<= 808}, sender_retry{x < 52};
init sender_wait;
trans
sender_wait -> sender_transm { sync begin !; assign x:= 0; },
sender_wait -> sender_wait { sync cd2 ?; assign x:= 0; },
sender_wait -> sender_retry { sync cd2 ?; assign x:= 0; },
sender_wait -> sender_retry { sync busy ?; assign x:= 0; },
sender_transm -> sender_wait { guard x == 808; sync end !; assign x:= 0; },
sender_transm -> sender_retry { guard x < 52; sync cd2 ?; assign x:= 0; },
sender_retry -> sender_transm { guard x < 52; sync begin !; assign x:= 0; },
sender_retry -> sender_retry { guard x < 52; sync cd2 ?; assign x:= 0; };
}
```

```
process P3 {
clock x;
state sender_n {x <= 2}, sender_wait, sender_transm {x<= 808},
      sender_retry {x < 52};
init sender_wait;
trans
    sender_wait -> sender_n { sync cd3 !; assign x:= 0;  },
    sender_wait -> sender_transm { sync begin !; assign x:= 0;  },
    sender_wait -> sender_wait { sync cd1 ?; assign x:= 0;  },
    sender_wait -> sender_retry { sync cd1 ?; assign x:= 0;  },
    sender_wait -> sender_retry { sync busy ?; assign x:= 0;  },
    sender_transm -> sender_wait { guard x == 808; sync end !; assign x:= 0;  },
    sender_transm -> sender_retry { guard x < 52; sync cd1 ?; assign x:= 0;  },
    sender_retry -> sender_transm { guard x < 52; sync begin !; assign x:= 0;  },
    sender_retry -> sender_retry { guard x < 52; sync cd1 ?; assign x:= 0;  };
}

system P0, P1, P2, P3;
```

- END OF INPUT FILE -

Program output:

```
PHASE 1 - File Parsing:

Nr of Processes: 4
Number of channels: 6
Defined Process-names: P0 - P1 - P2 - P3

Computations for process P0 ->

PHASE 2 - Detection of SCCs:

====================================
   Table of SCCs for process P0
====================================
   SCC-ID     |      Vertex(State)
------------------------------------
      0       |       bus_idle
      0       |       bus_active
      0       |       bus_collision1
      0       |       bus_collision2
      0       |       bus_collision3


PHASE 3 - Combining Reachable SCCs (Incidence matrix).

Incidence Matrix of SCC in P0:

 -1   1   0   0   0
  1  -1   0   0   0
  0  -1   1   0   0
  0   0  -1   1   0
  0   0   0  -1   1
  1   0   0   0  -1
```

```
PHASE 4 - Permuted Column Echelon Form:

  1   0   0   0
 -1   0   0   0
  0   1   0   0
  0   0   1   0
  0   0   0   1
 -1  -1  -1  -1

PHASE 5 - Basis of the orthogonal complement:

  1   1
  1   0
  0   1
  0   1
  0   1
  0   1

PHASE 6 - The Minimum Circuit Basis:

  1   1
  1   0
  0   1
  0   1
  0   1
  0   1

PHASE 7 - The Cycle Images:

  1   2
  1   0
  0   0
  0   1
  0   1
  0   1

PHASE 8 - Generation of Inequalities with lrs:

*lrs:lrslib v.4.2b, 2006.10.31(32bit,lrsmp.h)
*Copyright (C) 1995,2006, David Avis   avis@cs.mcgill.ca
*Input taken from file P0_H_REP.ine
*Output sent to file P0_V_REP.ext

  0   0   1   0   0   0
  0   0   0  -1   1   0
  0   0   0  -1   0   1
  1  -1   0  -2   0   0
  0   1   0   0   0   0
  0   0   0   1   0   0
```

```
PHASE 9 - State Exploration and Inequalities Specifications:

++++++++++ RESULTS FOR PROCESS: P0 ++++++++++

1 * busy >= [ bus_idle -> 0, bus_active -> 0, bus_collision1 -> 0, bus_collision
2 -> 0, bus_collision3 -> 0 ]

(-1) * cd1 + 1 * cd2 >= [ bus_idle -> 0, bus_active -> 0, bus_collision1 -> 0, b
us_collision2 -> -1, bus_collision3 -> 0 ]

(-1) * cd1 + 1 * cd3 >= [ bus_idle -> 0, bus_active -> 0, bus_collision1 -> 0, b
us_collision2 -> -1, bus_collision3 -> -1 ]

1 * begin + (-1) * end + (-2) * cd1 >=
[ bus_idle -> 0, bus_active -> 1, bus_collision1 -> 2, bus_collision2 -> 0, bus_
collision3 -> 0 ]

1 * end >= [ bus_idle -> 0, bus_active -> 0, bus_collision1 -> 0, bus_collision2
 -> 0, bus_collision3 -> 0 ]

1 * cd1 >= [ bus_idle -> 0, bus_active -> 0, bus_collision1 -> 0, bus_collision2
 -> 1, bus_collision3 -> 1 ]


Computations for process P1 ->

PHASE 2 - Detection of SCCs:

====================================
   Table of SCCs for process P1
====================================
   SCC-ID     |      Vertex(State)
------------------------------------
      0       |       sender_wait
      0       |       sender_transm
      0       |       sender_retry

PHASE 3 - Combining Reachable SCCs (Incidence matrix).

Incidence Matrix of SCC in P1:

 -1   1   0
 -1   0   1
 -1   0   1
  1  -1   0
  0  -1   1
  0   1  -1

PHASE 4 - Permuted Column Echelon Form:

  1   0
  1   1
  1   1
 -1   0
  0   1
  0  -1
```

```
PHASE 5 - Basis of the orthogonal complement:

 -1  -1   1   0
  1   0   0   0
  0   1   0   0
  0   0   1   0
 -1  -1   0   1
  0   0   0   1

PHASE 6 - The Minimum Circuit Basis:

  1   0   0   0
  0   0   1   0
  0   0   0   1
  1   0   1   1
  0   1   0   0
  0   1   1   1

PHASE 7 - The Cycle Images:

  1   1   1   1
  1   0   1   1
  0   0   0   1
  0   1   1   0

PHASE 8 - Generation of Inequalities with lrs:
*lrs:lrslib v.4.2b, 2006.10.31(32bit,lrsmp.h)
*Copyright (C) 1995,2006, David Avis   avis@cs.mcgill.ca
*Input taken from file P1_H_REP.ine
*Output sent to file P1_V_REP.ext

  1   0  -1  -1
  0   0   1   0
  0   1  -1   0
  1  -1   0   0
  0   0   1   0
  0   0   0   1

PHASE 9 - State Exploration and Inequalities Specifications:

++++++++++ RESULTS FOR PROCESS: P1 ++++++++++

1 * begin + (-1) * busy + (-1) * cd1 >=
[ sender_wait -> -1, sender_transm -> -2, sender_retry -> -4 ]

1 * busy >= [ sender_wait -> 0, sender_transm -> 0, sender_retry -> 0 ]

1 * end + (-1) * busy >=
[ sender_wait -> -1, sender_transm -> -1, sender_retry -> -1 ]

1 * begin + (-1) * end >=
[ sender_wait -> 0, sender_transm -> 1, sender_retry -> 0 ]

1 * busy >= [ sender_wait -> 0, sender_transm -> 0, sender_retry -> 0 ]

1 * cd1 >= [ sender_wait -> 0, sender_transm -> 0, sender_retry -> 0 ]
```

```
Computations for process P2 ->

PHASE 2 - Detection of SCCs:

====================================
   Table of SCCs for process P2
====================================
   SCC-ID     |      Vertex(State)
------------------------------------
      0       |       sender_wait
      0       |       sender_transm
      0       |       sender_retry

PHASE 3 - Combining Reachable SCCs (Incidence matrix).

Incidence Matrix of SCC in P2:

 -1   1   0
 -1   0   1
 -1   0   1
  1  -1   0
  0  -1   1
  0   1  -1

PHASE 4 - Permuted Column Echelon Form:

  1   0
  1   1
  1   1
 -1   0
  0   1
  0  -1

PHASE 5 - Basis of the orthogonal complement:

 -1  -1   1   0
  1   0   0   0
  0   1   0   0
  0   0   1   0
 -1  -1   0   1
  0   0   0   1

PHASE 6 - The Minimum Circuit Basis:

  1   0   0   0
  0   0   1   0
  0   0   0   1
  1   0   1   1
  0   1   0   0
  0   1   1   1
```

```
PHASE 7 - The Cycle Images:

  1   1   1   1
  1   0   1   1
  0   0   0   1
  0   0   0   0
  0   1   1   0

PHASE 8 - Generation of Inequalities with lrs:
*lrs:lrslib v.4.2b, 2006.10.31(32bit,lrsmp.h)
*Copyright (C) 1995,2006, David Avis   avis@cs.mcgill.ca
*Input taken from file P2_H_REP.ine
*Output sent to file P2_V_REP.ext

  0   0   0   1   0
  1   0  -1   0  -1
  0   0   1   0   0
  0   1  -1   0   0
  1  -1   0   0   0
  0   0   1   0   0
  0   0   0   0   1

PHASE 9 - State Exploration and Inequalities Specifications:

++++++++++++ RESULTS FOR PROCESS: P2 ++++++++++++

1 * cd1 >= [ sender_wait -> 0, sender_transm -> 0, sender_retry -> 0 ]

1 * begin + (-1) * busy + (-1) * cd2 >=
[ sender_wait -> -1, sender_transm -> -2, sender_retry -> -4 ]

1 * busy >= [ sender_wait -> 0, sender_transm -> 0, sender_retry -> 0 ]

1 * end + (-1) * busy >=
[ sender_wait -> -1, sender_transm -> -1, sender_retry -> -1 ]

1 * begin + (-1) * end >=
[ sender_wait -> 0, sender_transm -> 1, sender_retry -> 0 ]

1 * busy >= [ sender_wait -> 0, sender_transm -> 0, sender_retry -> 0 ]

1 * cd2 >= [ sender_wait -> 0, sender_transm -> 0, sender_retry -> 0 ]


Computations for process P3 ->

PHASE 2 - Detection of SCCs:

====================================
   Table of SCCs for process P3
====================================
   SCC-ID     |      Vertex(State)
------------------------------------
      1       |       sender_n
      2       |       sender_wait
      2       |       sender_transm
      2       |       sender_retry
```

```
>> New SCCs after adding new edges:

====================================
   Table of SCCs for process P3
====================================
   SCC-ID     |      Vertex(State)
------------------------------------
      2       |       sender_n
      2       |       sender_wait
      2       |       sender_transm
      2       |       sender_retry

PHASE 3 - Combining Reachable SCCs (Incidence matrix).
Incidence Matrix of SCC in P3:

  1  -1   0   0
  0  -1   1   0
  0  -1   0   1
  0  -1   0   1
  0   1  -1   0
  0   0  -1   1
  0   0   1  -1
 -1   1   0   0

PHASE 4 - Permuted Column Echelon Form:

  1   0   0
  0   1   0
  0   1   1
  0   1   1
  0  -1   0
  0   0   1
  0   0  -1
 -1   0   0

PHASE 5 - Basis of the orthogonal complement:

  0   0   0   0   1
 -1  -1   1   0   0
  1   0   0   0   0
  0   1   0   0   0
  0   0   1   0   0
 -1  -1   0   1   0
  0   0   0   1   0
  0   0   0   0   1

PHASE 6 - The Minimum Circuit Basis:

  0   0   1   0   0
  1   0   0   0   0
  0   0   0   1   0
  0   0   0   0   1
  1   0   0   1   1
  0   1   0   0   0
  0   1   0   1   1
  0   0   1   0   0
```

```
PHASE 7 - The Cycle Images:

  1   1   0   1   1
  1   0   0   1   1
  0   0   0   0   1
  0   1   0   1   0
  0   0   0   0   0
  0   0   1   0   0

PHASE 8 - Generation of Inequalities with lrs:
*lrs:lrslib v.4.2b, 2006.10.31(32bit,lrsmp.h)
*Copyright (C) 1995,2006, David Avis   avis@cs.mcgill.ca
*Input taken from file P3_H_REP.ine
*Output sent to file P3_V_REP.ext

  0   0   0   0   1   0
  1   0  -1  -1   0   0
  0   0   1   0   0   0
  0   1  -1   0   0   0
  0   0   0   0   0   1
  1  -1   0   0   0   0
  0   0   1   0   0   0
  0   0   0   1   0   0
  0   0   0   0   0   1

PHASE 9 - State Exploration and Inequalities Specifications:

++++++++++ RESULTS FOR PROCESS: P3 ++++++++++

1 * cd2 >= [ sender_n -> 0, sender_wait -> 0, sender_transm -> 0, sender_retry -
> 0 ]

1 * begin + (-1) * busy + (-1) * cd1 >=
[ sender_n -> 0, sender_wait -> -1, sender_transm -> -2, sender_retry -> -4 ]

1 * busy >= [ sender_n -> 0, sender_wait -> 0, sender_transm -> 0, sender_retry
-> 0 ]

1 * end + (-1) * busy >=
[ sender_n -> 0, sender_wait -> 0, sender_transm -> -1, sender_retry -> -1 ]

1 * cd3 >= [ sender_n -> 1, sender_wait -> 0, sender_transm -> 0, sender_retry -
> 0 ]

1 * begin + (-1) * end >=
[ sender_n -> 0, sender_wait -> 0, sender_transm -> 1, sender_retry -> 0 ]

1 * busy >= [ sender_n -> 0, sender_wait -> 0, sender_transm -> 0, sender_retry
-> 0 ]

1 * cd1 >= [ sender_n -> 0, sender_wait -> 0, sender_transm -> 0, sender_retry -
> 0 ]

1 * cd3 >= [ sender_n -> 1, sender_wait -> 0, sender_transm -> 0, sender_retry -
> 0 ]
```

References

[ADLly] Gerd Behrmann Alexandre David and Kim G. Larsen. Tutorial on Uppaal. *Department of Computer Science, Aalborg University, Denmark*, This document is updated regularly. http://www.it.uu.se/research/group/darts/uppaal/documentation.shtml. 9

[AF92] D. Avis and K. Fukuda. A Pivoting Algorithm for Convex Hulls and Vertex Enumeration of Arrangements and Polyhedra. *School of Computer Science, McGill University, Montreal, Canada*, 1992. 26

[Avi] D. Avis. Users Guide for lrs . *Version 3.2, 1997. available from lrs homepage ftp://mutt.cs.mcgill.ca/pub/C/lrs.html.* 26

[Avi99a] D. Avis. Computational Experience with the Reverse Search Vertex Enumeration Algorithm. *School of Computer Science, McGill University, Montreal, Canada*, 1999. 26

[Avi99b] D. Avis. lrs: A revised implementation of the reverse search vertex enumeration algorithm. *School of Computer Science, McGill University, Montreal, Canada*, 1999. programs lrs*.c available from ftp://mutt.cs.mcgill.ca/pub/C/. 26

[Ber85] C. Berge. Graphs. *North-Holland, Amsterdam, NL*, 1985. 15

[BG05] Howard Bowman and Rodolfo Gomez. *Concurrency Theory: Calculi an Automata for Modelling Untimed and Timed Concurrent Systems* . Springer, Berlin, 2005. 2

[DMC94] David Heckerman David M. Chickering, Dan Geiger. On finding a cycle basis with a shortest maximal cycle. *Information Processing Letters, 54:55-58*, 1994. 18

[GBY02] Alexandre David Kim G. Larsen Paul Pettersson Gerd Behrmann, Johan Bengtsson and Wang Yi. UPPAAL Implementation Secrets. *Department of Information Technology, Uppsala University, Sweden*, 2002. 9

[GLS03] P. Gleiss, J. Leydold, and P. Stadler. Circuit bases of strongly connected digraphs. 2003. 15

[NSS94] Esko Nuutila and Eljas Soisalon-Soininen. On Finding the Strongly Connected Components in a Directed Graph. *IPL 49*, 1994. 12

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986. 25

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing. Bd. 1*, 1972. 12

[Val98] A. Valmari. *The State Explosion Problem.* In Lectures on Petri Nets I: Basic Models, volume 1491 of LNCS, pages 429-528, Springer-Verlag, 1998. 1