Automatic Abstraction of Communication Sequences

Bachelor's Thesis

submitted by
Sven Bünte

Supervisor
Prof. Dr. Bernd Finkbeiner

Advisor
Dipl. Inf. Klaus Dräger

Reviewers
Prof. Dr. Bernd Finkbeiner
Prof. Dr. Reinhard Wilhelm

Saarland University
Faculty of Natural Sciences and Technology
Department of Computer Science
Reactive Systems Group
Bachelor's Program in Computer Science

October 27, 2006

Abstract

The exponential state-space blow-up that arises when analyzing reachability in networks of synchronizing processes is one of the main problems in model checking. This thesis will introduce an approach in which communication sequences are abstracted algebraically. The abstraction includes information about safety properties in such a way that if the solution set of the abstraction is empty, the stated properties hold. All involved means have at most polynomial complexity. Since the solution space is an overapproximation, the checking procedure is incomplete yet. However, the ideas and methods that will be introduced form a nice basis for doing further abstraction refinement such that valuable information about reachability might be derived conveniently in the future.

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

Saarbrücken, October 27, 2006

Sven Bünte

Contents

List of Algorithms

Chapter 1

Introduction

> "A communication sequence is a series of actions and reactions between (...) two or more people. A sequence has a beginning event ("Chuck cleared his throat") and a reaction > reaction > reaction chain until some local ending event ("Sharon left the room"). Three key types of sequences are (a) need- assertions, (b) conflict resolutions (problem solving), and (c) giving and receiving praise, like affirming, validating, and approving. (...)" [oA]

This definition is from a divorce-prevention website. I am afraid, the results of this thesis will not yield to better marriages. Nevertheless, if "people" are replaced by "modules", the definition is quite accurate to describe what we are dealing with. Whenever a person and module, respectively, is dependent on another one in a way that it needs some service or information, communication sequences are essential. They can be found in almost everything that computer science has put forth, beginning with small hardware modules up to the Internet including all layers inbetween. Verification of communication sequences is therefore not only extremely useful, it is essential when it comes to safety-critical systems. The thesis will focus on reachability analysis as one important aspect.
Chapter 2 will introduce means to model communication sequences appropriately and show how these models have to be modified in order to form a basis for an albegraic abstraction. The reason for an overapproximation comes from the exponential complexity of the reachability problem when analyzing the system's state space explicitly. After a short illustration of this drawback, the procedure on how to build and modify the abstraction will be discussed in detail. At the very end, a first attempt to refine the abstraction will be introduced.

Chapter 2

Communication Sequences

2.1 Reactive Systems

Reactive systems are a class of software and/or hardware systems which have ongoing behavior, i.e. they do not terminate [Hel05]. Examples of reactive systems include:

- Traffic lights
- Elevators
- Operating systems
- Data Communication protocols (Internet, telephone switches)
- Mobile phones

These are just a few instances. Embedded systems are usually reactive, especially controlling units in safety critical environments are. Already in 2003 more than 79% of all processors were used in embedded systems [Mar03]. Reactive systems often have to guarantee strict safety properties on the one hand but are difficult to test due to the non-terminating behavior on the other hand. Formal verification provides methods for deriving significant and reliable properties. Reasoning about reactive systems has to involve communication sequences to some extend which is one aspect that shows how important they are.

2.2 Modelling Communication Sequences

Transition Systems [Kel76] or ω-*Automata* [GTWE02] represent a very nice way to model systems in which communication sequences contribute. They are capable of accepting infinite words (i.e. reactive systems can be modeled) and their expressive power is adequate. To describe the ideas of this thesis, however, they are too expressive - finite state systems will be analyzed only. *Nondeterministic Finite State Automata* will do the job for modelling these systems conveniently with regard to automatically guaranteeing safety properties.

2.2.1 Nondeterministic Finite Automata

A nondeterministic finite state automaton (NFA) is a quintuple $A = (S, \Sigma, \Delta, S^0, F)$, where:

- S is a finite set of states,

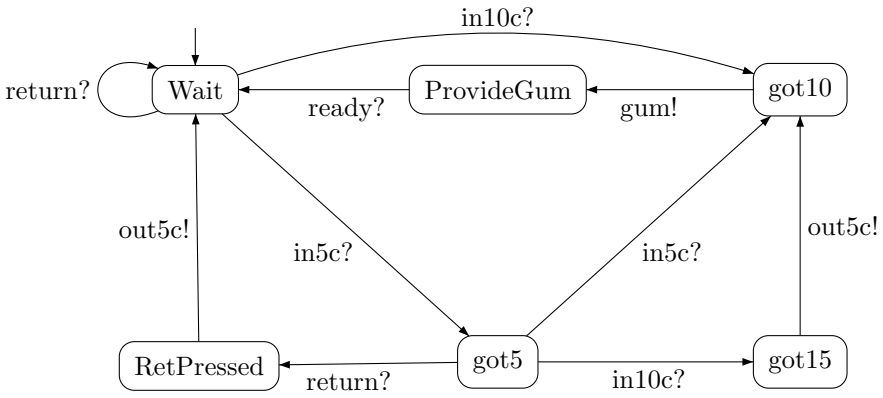
- Σ is a finite set of input symbols,
- Δ is a transition function ($\Delta : S \times \Sigma \to 2^S$),
- $S^0 \subseteq S$ is the set of initial states,
- $F \subseteq S$ is the set of accepting states.

We consider a finite set of communicating processes, each modelled by one dedicated automaton. All symbols in Σ (except ϵ) represent an atomic communication instance. They can also be seen as synchronizations between processes. An *empty transition* is labeled with symbol ϵ and can be taken without any synchronization. Formally this means that if the system consists of automata $A_1, \ldots, A_n$, the intersection of the corresponding languages $L(A_1) \cap \cdots \cap L(A_n)$ form all the system's communication sequences. One constraint is that processes are only allowed to synchronize in pairs. Therefore the intersection of all languages is not an accurate model if the system consists of more than two processes. Section 3.1.2 will present simple means to correct the model concerning this matter.

2.2.2 A Simple Vending Machine

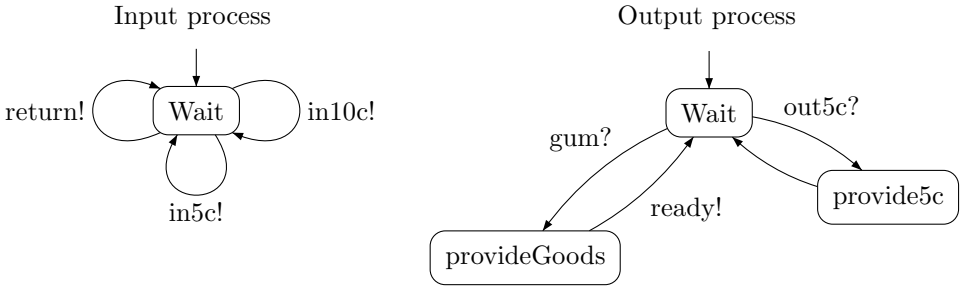
Here is an example of how a simple device can be modeled. Figure 2.1 shows a control unit for a chewing gum vending machine. Figure 2.2 illustrates the dedicated in- and output processes. The system accepts 5 and 10 cent coins. If a 10 cent coin is inserted one chewing gum is delivered immediately. In case a 5 cent coin is inserted, the user has the opportunity either to get the money back by pressing the return key or to get the candy by throwing in a further 5 or 10 cent coin. In the latter case the change is delivered in addition to the goods. The nodes in the diagram represent the

Figure 2.1: Control Process for a Vending Machine



states of the process, edges refer to possible transitions between them. Transitions are *enabled* whenever they can be synchronized on the *channel* that is referenced by the transition label, i.e. there must be another process that has synchronization transitions on the same channel and both must be in a state where they can execute the joint transition. In case a process has multiple enabled transitions, one of them is chosen nondeterministically. The nodes labeled by "Wait" are the initial states, marked by an additional arrow that has no source node.
The processes are not modelled by NFAs so far. On the one hand final states are

Figure 2.2: In-/Output Processes for a Vending Machine



not specified, on the other hand suffixes '!' and '?' are appended to the transition labels. Intuitively, the question mark corresponds to receiving, the exclamation mark to sending. In order for two non-empty transitions in different processes to be enabled, not only the related channel must be identical in both, but also the suffixes of them have to be complementary. The presented graphical description language is used analogously with extensions in several tools for automatic verification. The model can be mapped to a network of NFAs without loosing any information about reachable error states, as I will show later.

2.2.3 Textual Description Language

In order to make graphs readable for tools a textual description is quite useful. The grammar shown in Figure 2.3 generates an appropriate language in this regard. Moreover, the input language is a subset of the *XTA*-format [Beh], one of the input languages for the modelling and analyzing tool UPPAAL [UU]. Following the example of the vending machine, Figure 2.4 shows an XTA representation of a system containing only the output unit.

Ita	→	*ChanList ProcList Globals*
ChanList	→	ϵ | chan *IdList* ;
ProcList	→	*Proc* | *Proc ProcList*
Globals	→	system *IdList* ;
Proc	→	process *Id* { *ProcBody* }
IdList	→	*Id* | *Id* , *IdList*
ProcBody	→	*StateDecls TransDecls*
StateDecls	→	state *IdList* ; init *Id* ; | state *IdList* ; init *Id* ; final *Id* ;
TransDecls	→	trans TransList ;
TransList	→	*Trans* | *Trans* , *TransList*
Trans	→	*Id* − > *Id* { *OpSync* }
OpSync	→	ϵ | sync *Id* ! ; | sync *Id* ? ;
Id	→	*Alpha* | *Id AlphaNum*
Alpha	→	A | ... | Z | a | ... | z
Num	→	0 | ... | 9
AlphaNum	→	*Alpha* | *Num* | _

Figure 2.3: Grammar for the textual description language [BL96]

Figure 2.4: Output Unit in XTA-Format

```
chan gum, out5c, ready;
process OutputUnit{
  state Wait, Provide5c, ProvideGoods;
  init Wait;
  trans Wait -> Provide5c{sync out5c?; },
  Wait -> ProvideGoods{sync gum?; },
  ProvideGoods -> Wait{sync ready!; },
  Provide5c -> Wait{};
}
system OutputUnit;
```

2.2.4 More Sophisticated Models

Because communication sequences are analyzed at a high level, the model described so far is sufficient. However, for real-life applications one will eventually need more instruments to build appropriate models. Global and local variables, constraints on when to enter or exit states or assignments to variables when transitions are taken are means to gain much more expressive models. Time can be another important aspect of reactive systems or protocols. An airbag in a car for example must be activated in time, otherwise it is completely useless. Such systems are called *Real-time Systems*.

2.3 Model Checking

One technique for verifying reactive systems is *model checking*. It has the advantage that it is more precise than simulation or testing. Furthermore, it can be done automatically once the model and the specification are provided. This aspect clearly characterizes an advantage over deductive reasoning. One basic model checking approach is *reachability analysis*. Referring to the vending machine in Section 2.2.2, the question may arise whether the system can reach a situation such that the control process is in state *got*5 while the output process is in state *provideGoods* at the very same time. This would definitely constitute a design flaw. Fortunately, this can not be the case and indeed, the model checker UPPAAL says so as well. Deriving system properties of this kind is also referred to as performing *reachability analysis*.

2.3.1 Reachability Analysis

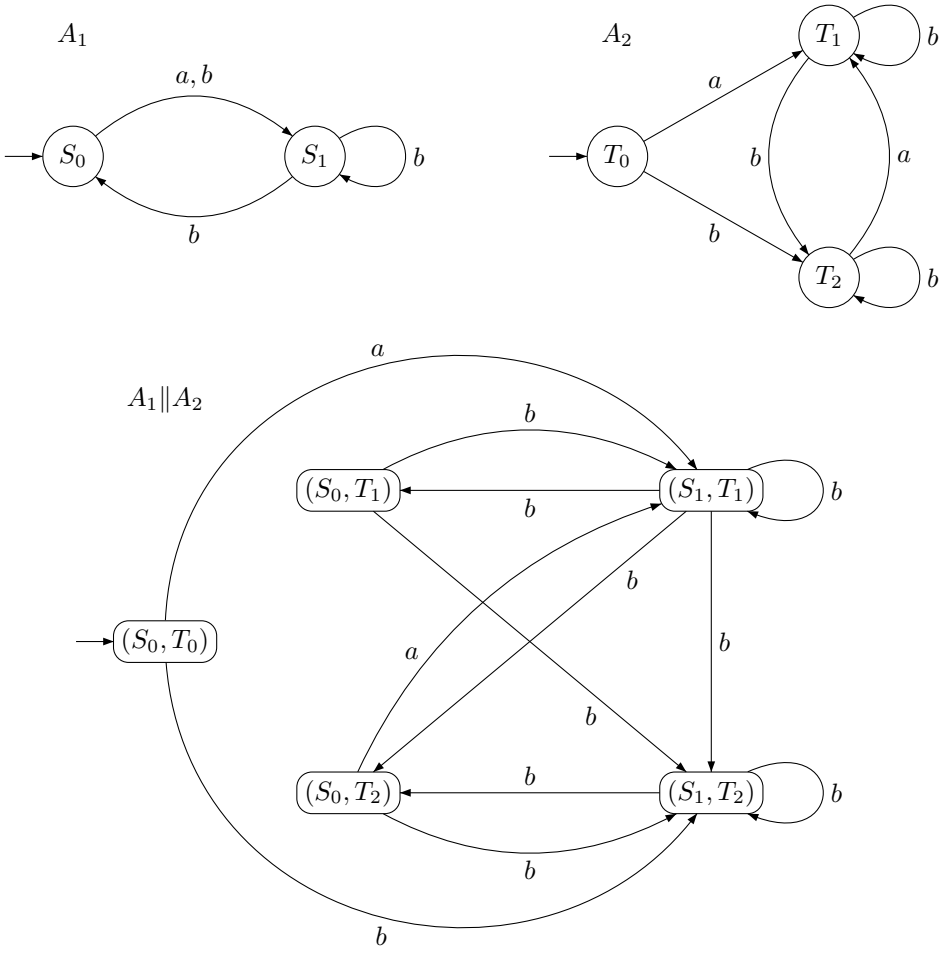
The system's concurrency, i.e. the nondeterminism in its interleavings can easily lead to an exponential state space blow-up when reachability analysis is performed. Building the *product automaton* for instance, a straightforward way to model all execution paths and synchronizations explicitly, has exponential worst case complexity. Let a system consist of automata $A_1 = (\Sigma, S_1, S_1^0, \Delta_1, F_1)$ and $A_2 = (\Sigma, S_2, S_2^0, \Delta_2, F_2)$. The product automaton is then defined as $A = (\Sigma, S, S^0, \Delta, F)$, where:

- $S = S_1 \times S_2$,
- $S^0 = S_1^0 \times S_2^0$,
- for all $s, s' \in S_1, t, t' \in S_2, a \in \Sigma$:
  $((s,t), a, (s',t')) \in \Delta$ iff $(s, a, s') \in \Delta_1$ and $(t, a, t') \in \Delta_2$,

- $F = F_1 \times F_2$.

A is also denoted as $A_1 \| A_2$ and it holds that $L(A_1 \| A_2) = L(A_1) \cap L(A_2)$. The definition can simply be extended to systems that consist of more than two automata. Due to the fact that we restrict synchronizations to be pairwise, some preprocessing of the automata becomes necessary to keep the semantics (Section 3.1.2). Figure 2.5 shows an example of how to build a product automaton. Note, that the unreachable product state (S_1, T_0) has been removed along with all incident edges. As said, the state space of the product automaton grows exponentially with the number of processes. Although there is a lot of room for optimizations when doing reachability analysis, the worst case complexity remains the same. The state explosion problem is actually one of the main sources of research in model checking.

Figure 2.5: Example of a Product Automaton



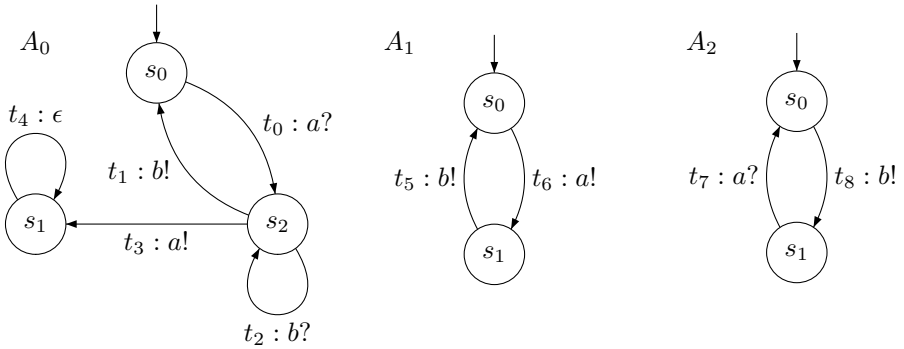
Chapter 3

Abstraction of Communication Sequences

In this chapter I will describe a possible way to avoid the state space explosion when doing reachability analysis. At first there will be some preprocessing to the input automata. The flow and the synchronizations will then be abstracted subsequently by polynomial equations in $\mathbb{Z}[x]$. Thus, the reachability problem is reduced to checking if the system of equations has the zero polynomial as its only solution. We will see that the solution space is an over-approximation. This requires further abstraction refinement. All introduced means will have at most polynomial complexity.

3.1 Model Transformation

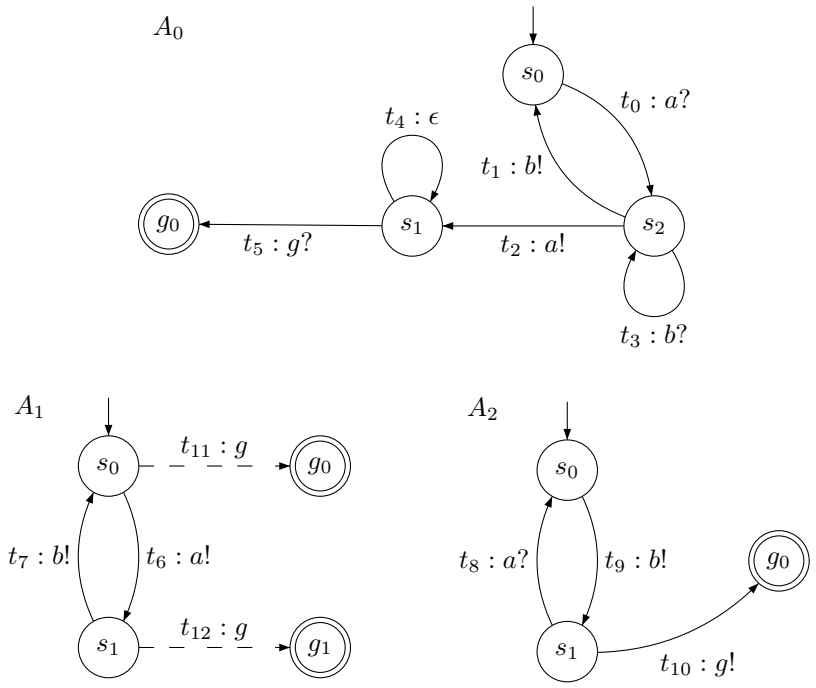
Figure 3.1: Example System



3.1.1 Goal Transitions

Until now, the underlying model is not yet based on NFAs since final states have not been discussed. It is rather set up by a set of finite ω-Automata so far. The capability of accepting infinite words constitutes the difference. The final states that are to be introduced will indirectly specify an *error state* for the reachability

Figure 3.2: Goal Transitions



analysis. Furthermore, final states will restrict all the system's communication sequences to be of finite length. The mapping from ω-automata to NFAs therefore has to preserve the reachability properties (also for infinite sequences).
The idea for the transformation is the following: Imagine we would like to test if state $(A_0.s_1, s, A_2.s_1)$ from Figure 3.1 is reachable for some $s \in A_1$. At first, final states $A_0.g_0$ and $A_2.g_0$ are introduced, visualized by round marks. Both $A_0.s_1$ and $A_2.s_1$ will then be the source of a single, dedicated *goal transition* that yields to $A_0.g_0$ or $A_2.g_0$ respectively, as illustrated in Figure 3.2 (transition labels '$t_i : c$' now specify the name t_i of the corresponding transition in addition to its synchronization channel c). Goal transitions always synchronize on the system's unique *goal channel* g. Now, for a synchronization to be possible on g, the system has to reside in $(A_0.s_1, s, A_2.s_1)$ for some s. Algorithm 1 describes the transformation procedure in detail. With respect to our example, it also introduces $A_1.g_0$ and $A_1.g_1$ along with transitions t_{11} and t_{12}. This is to establish a chance for the system to stop immediately after an error state has been reached. How both t_{11} and t_{12} can join the synchronization between t_5 and t_{10} will be made clear in Section 3.1.2. Regarding the transformed model, the following properties are now equivalent:

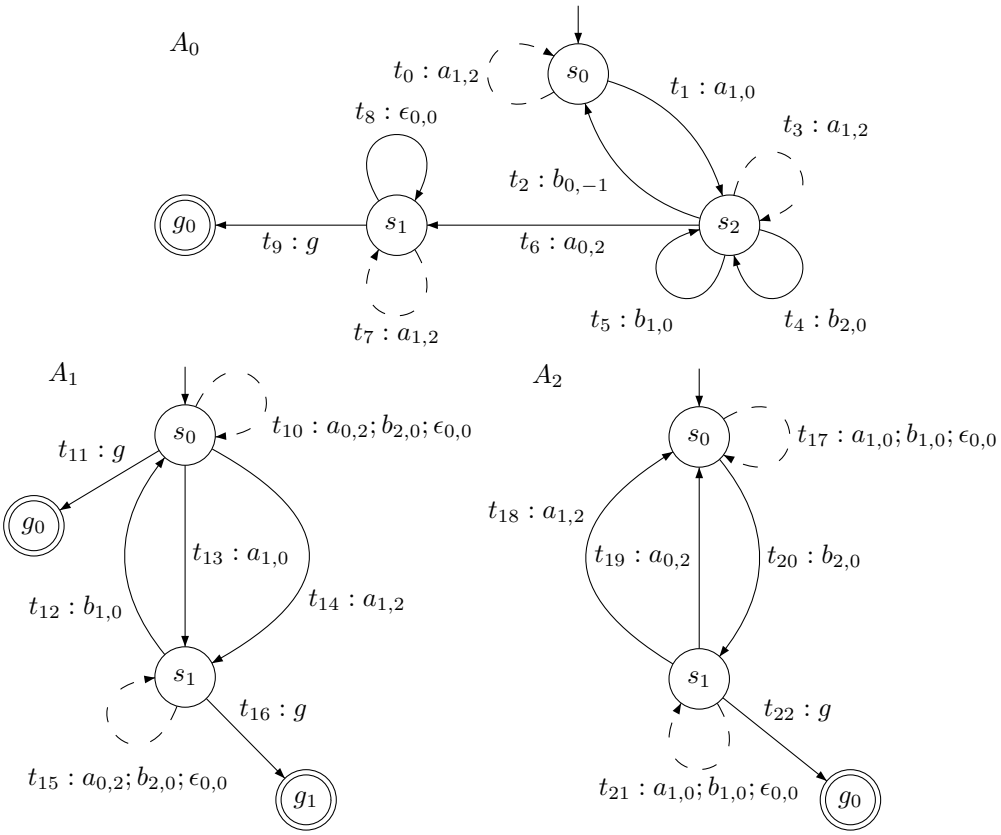
- There is a run in which the specified error state is active at some point.
- There is a **finite** run in which the specified error state is active at some point.

3.1.2 Channel Splitting and Artificial Self Loops

This section will substitute all empty transitions from the input automata on the one hand and duplicate/relabel transitions corresponding to synchronization actions on the other hand. The latter plan will require splitting of channels/symbols

as well as to introduce self loops. The procedure entails the demanded semantics of exclusive pairwise synchronization actions and preserves reachability properties. Furthermore, the channel suffixes '!' and '?' wont be needed anymore, their meaning will also be expressed by the new channels. As a matter of fact, in order to apply the later abstraction techniques, the system must be transformed in such a way that the product automaton's language represents all possible runs to the specified error state. Substitution of empty transitions becomes necessary to keep time consistency throughout the system. This is necessary with regard to abstraction techniques in Section 3.2.2.

Figure 3.3: Unique Synchronization Actions



To make synchronization actions unique with respect to the sending and receiving process, channels are duplicated and renamed accordingly. Figure 3.3 shows the transformed model of our familiar example. An indexed channel $c_{i,j}$ is now reserved for a synchronization in which A_i sends and A_j receives respectively on channel c. All other processes $A_{k \neq i,j}$ get an *artificial self loop* (dashed in example) attached to every non-final state, synchronized on $c_{i,j}$ as well. Remark that for all A_k, artificial self loops are the only transitions that are allowed to synchronize on channel $c_{i,j}$ or $c_{j,i}$. As an example, t_3 from Figure 3.2 has been split into new transitions t_4 and t_5 since A_1 can synchronize with A_0 and A_2. Consequently, A_1 now holds artificial self loops that are synchronized on $b_{2,0}$ or A_2 on $b_{1,0}$ respectively. If for a particular transition (e.g. t_2) no process can be found to synchronize, the corresponding channel index gets a '-1' entry. Transitions that will certainly never be taken in

any possible run are referred to as *dead transitions*.
The described modifications to the model suffice for the system's product automaton to accept a language that accurately describes all runs leading to the error state. However, the later time abstraction demands one little further model alteration. Empty transitions form the problem. Recall that they can be taken without any synchronization, i.e. independently of other processes. We definitely need a system-wide dependency for any system action since we will abstract time globally later on and the processes somehow have to adhere to it. The example shows how empty transitions (i.e. t_8) can easily be transformed. Each process A_i gets one dedicated channel $\epsilon_{i,i}$ for empty transitions to synchronize on. All other processes $A_{j\neq i}$ will again introduce artificial self loops that are synchronized on $\epsilon_{i,i}$ for each non-final state. Thus, the behavior of empty transitions is modelled properly and the requirement of having only system-wide actions is met. The previously defined goal transitions remain untouched. Goal channel g does not require indices since it is unique and is independent of the synchronization's kind with respect to '!' and '?'. The channel suffix can therefore simply be deleted.

3.2 Model Abstraction

Finally, all preprocessing to the underlying model has been performed and we are ready to build up a polynomial system of equations over $\mathbb{Z}[x]$ as an algebraic over-abstraction of the NFA network. The system of equations will be formed in such a way that if its solution space is empty or has the zero polynomial as its only solution, the accepted language of the system's product automaton is empty. This constitutes a proof then for the error state not to be reachable for any run.
As mentioned, real-time properties will not be considered. Therefore it is sufficient to abstract time in a very basic way. Time is global, discrete and whenever a synchronization is processed it is increased by one. The natural numbers are therefore sufficient to abstract time. At first, each transition t_i in the system gets a dedicated variable $e_i \in \mathbb{Z}[x]$. The idea is that for a concrete run, the value of each e_i determines the point(s) in time at which t_i is taken.

$$\begin{aligned} \delta_i(t) &= \begin{array}{ll} 1 & \text{if } t_i \text{ is taken at time } t \\ 0 & \text{otherwise} \end{array} \\ e_i(x) &= \sum_{k=0}^{t} \delta_i(k) \cdot x^k \end{aligned}$$

For example, if we consider a run of the system, in which t_m is taken at time 4 and 11, e_m has the value $x^4 + x^{11}$. Time starts at $t_0 = 0$. One can imagine that in each process a virtual transition carrying the value x^0 leads to the initial state of the corresponding process.
Each single run can now simply be represented by an allocation vector $e \in \mathbb{Z}[x]^n$, where n is the number of transitions in the system. Flow and synchronization equations are the two kinds of constraints that will form our abstraction.

3.2.1 Flow

The basic idea for flow abstraction is that the sum of all incoming transitions of a particular state times x must equal the sum of all outgoing transitions. Recall that the exponent in each term of a polynomial is directly related to the time the corresponding transition is taken. The multiplication of that polynomial by x has the effect that each term's exponent is increased by one. The resulting polynomial

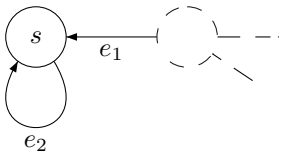
therefore describes a consecutive transition. The following equations form the flow abstraction of our example (Figure 3.3):

$$\begin{array}{lrcl}
A_0: & x \cdot (x^0 + e_0 + e_2) & = & e_0 + e_1 \\
 & x \cdot (e_6 + e_7 + e_8) & = & e_7 + e_8 + e_9 \\
 & x \cdot (e_1 + e_3 + e_4 + e_5) & = & e_2 + e_3 + e_4 + e_5 + e_6 \\
A_1: & x \cdot (x^0 + e_{10} + e_{12}) & = & e_{10} + e_{11} + e_{13} + e_{14} \\
 & x \cdot (e_{13} + e_{14} + e_{15}) & = & e_{12} + e_{15} + e_{16} \\
A_2: & x \cdot (x^0 + e_{17} + e_{18} + e_{19}) & = & e_{17} + e_{20} \\
 & x \cdot (e_{20} + e_{21}) & = & e_{18} + e_{19} + e_{21} + e_{22}
\end{array}$$

This can be expressed equivalently by:

$$M_{Flow} \cdot (x^0, e_0, \ldots, e_{22})^T \quad = \quad 0$$

where M_{Flow} has only entries in $\{0, 1, -x, 1-x\} \subseteq \mathbb{Z}[x]$. Given a network of NFAs, Algorithm 4 constructs M_{Flow} correspondingly. Recall that the introduction of goal transitions guarantees that only finite runs have to be considered for the reachability analysis. Polynomials therefore suffice to represent all erroneous runs. Algorithm 4 might derive equations of the form $x \cdot (e_1 + e_2) = e_2$, describing the flow of the following subgraph:



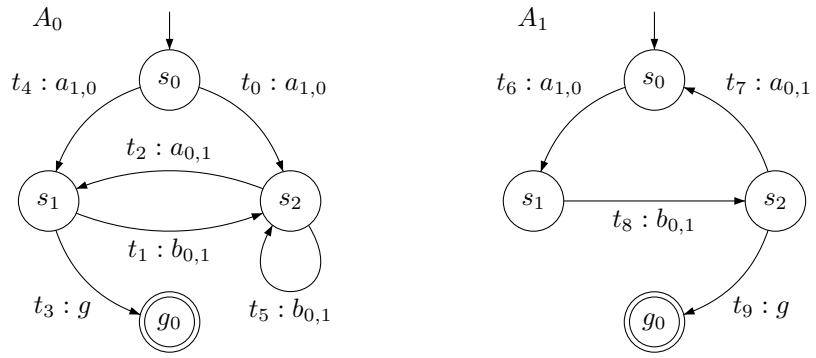
The only solution is $e_1 = e_2 = 0$, i.e. a run in which e_2 is taken cannot be expressed. However, since we assume that goal transitions have been introduced properly, we are not interested in solutions in which e_2 is taken. We know that the process in which the subgraph occurs does have locations that specify an error-state, otherwise there would be an outgoing goal transition from s. Thus, if e_2 is ever taken, we cannot reach an error state anymore and the respective run therefore does not need to be considered.

3.2.2 Synchronization

In case the system consists of only two processes, the abstraction is quite easy since there are no artificial self loops. Synchronization transitions have to be taken at the same time in both processes. Hence, for this particular synchronization action, the values of the corresponding variables both have to include a mutual term x^t where t is the time at which the transitions are taken. Furthermore, the system has been modified in such a way that transitions can only be taken via synchronizations. We therefore get one constraint for each channel $c_{0,1}$ (assume the two processes are named A_0 and A_1) that occurs in the system. Now, we simply equate the sum of all transitions synchronized on $c_{0,1}$ in A_0 with the sum of all transitions synchronized on $c_{0,1}$ in A_1, i.e.:

$$\begin{array}{rcl}
\displaystyle\sum_{k \in Ind(c_{0,1})(A_0)} e_k & = & \displaystyle\sum_{k \in Ind(c_{0,1})(A_2)} e_k \\
Ind & = & \lambda c.A.\{\ k \mid t_k \text{ synchronized on } c \text{ in } A\}
\end{array}$$

Figure 3.4: Two Processes to Synchronize



Synchronizations on goal channel g are handled in the same manner:

$$\begin{aligned} \sum_{k \in GT(A_0)} e_k &= \sum_{k \in GT(A_1)} e_k \\ GT &= \lambda A.\{\ k \mid t_k \text{ is goal transition in } A\ \} \end{aligned}$$

In case dead transitions t_m are located, i.e. $i = -1$ or $j = -1$ for their synchronization channels $c_{i,j}$, e_m can simply be set to zero since we know that it will never be taken. As an example, the network of automata from Figure 3.4 yields the following system of equations:

$$\begin{aligned} e_0 + e_4 &= e_6 \\ e_2 &= e_7 \\ e_1 + e_5 &= e_8 \\ e_3 &= e_9 \end{aligned}$$

$$\Leftrightarrow \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} x^0 \\ e_0 \\ \vdots \\ e_9 \end{pmatrix} = 0$$

However, the abstraction of synchronization events demands a bit more effort if the system consists of more than two processes. The problem is the already stated time consistency. Recall that time is regarded globally. The flow abstraction for each process takes care that time proceeds stepwise for each transition that is taken. The only thing left for the abstraction to include are constraints that force all processes which are not directly involved in a synchronization action to take their proper artificial self loops. To this end, a new variable is introduced for each channel. The variables will be named by the channel they refer to, except that capital letters will be used to distinguish them (Θ will be the variable dedicated to ϵ). The synchronization constraints do now look slightly different:

$$\sum_{k \in Ind(c_{i,j})(i)} e_k = \sum_{k \in Ind(c_{i,j})(j)} e_k = C_{i,j} \qquad (i, j \neq -1)$$

... for each $c_{i,j}$. Again, goal channel g is handled similarly. For all processes A there is one constraint:

$$\sum_{k \in GT(A)} e_k \quad = \quad G$$

The new variables can now help to constrain artificial self loops. For each process A we get one further equation:

$$\begin{aligned} \sum_{k \in Self(A)} e_k &= \sum_{C \in Cha(A)} C \\ Self &= \lambda A.\{\ k \mid t_k \text{ is artificial self loop in } A\} \\ Cha &= \lambda A.\{\ C_{i,j} \mid A \neq A_i, A_j\} \end{aligned}$$

Consider the example from Figure 3.3 again. The following system of equations (see Section A.4 for formal implementation) expresses its synchronization abstraction according to the introduced definitions.

$$e_1 = e_{13} = A_{1,0}$$

$$e_6 = e_{19} = A_{0,2}$$

$$e_{14} = e_{18} = A_{1,2}$$

$$e_5 = e_{12} = B_{1,0}$$

$$e_4 = e_{20} = B_{2,0}$$

$$e_2 = 0$$

$$e_8 = \Theta_{0,0}$$

$$e_9 = e_{22} = G$$

$$e_{11} + e_{16} = G$$

$$e_0 + e_3 + e_7 = A_{1,2}$$

$$e_{10} + e_{15} = A_{0,2} + B_{2,0} + \Theta_{0,0}$$

$$e_{17} + e_{21} = A_{1,0} + B_{1,0} + \Theta_{0,0}$$

3.3 Syzygy Module

The abstraction can now be completely expressed by a matrix M, representing an equation system of the following form:

$$M \cdot u = 0$$

...where M has entries in $\{0, 1, -x, 1 - x\} \subseteq \mathbb{Z}[x]$ and $u \in \mathbb{Z}[x]^n$ is the solution vector:

$$u := (x^0, e_1, \ldots, e_i, C_1, \ldots, C_j, e_{i+1}, \ldots, e_l, e_{l+1}, \ldots, e_{n-j-2}, G)^T$$

It contains all introduced variables plus x^0 in order to describe initial transitions. It is structured as follows:

x^0	The value of initial transitions, i.e. we start at time 0
$e_1 \dots e_i$	One variable for each transition from the input system
$C_1 \dots C_j$	All channels after splitting get dedicated variables
$e_{i+1} \dots e_l$	Artificial self loop variables
$e_{l+1} \dots e_{n-j-2}$	One variable for each goal transition
G	Variable for the dedicated goal channel

Unfortunately, the abstraction so far is not quite handy. An obvious approach now is to check the matrix for possible solutions. Note that the solution space is an over-approximation since polynomials are now allowed to have coefficients in $\mathbb{Z}$. However, expressing the solution space by basis vectors can remove potential redundancy.

The first thing to do is applying Gaussian elimination. This works fine since we deal with univariate polynomials. Forward elimination is formalized by Algorithm 8. It utilizes the greatest common divisor for some calculations to keep Matrix entries as small as possible. Therefore, polynomials have to be regarded as elements in $\mathbb{Q}[x]$ from now on. Forward elimination results in a matrix that is in row echelon form, except that leading coefficients might be different from 1 (note that not every element in $\mathbb{Q}[x]$ has a multiplicative inverse). If the last non-zero row's leading coefficient p is situated at the very last position in its row, i.e. $p \cdot G = 0$, we can stop since G has to be zero. Otherwise backward elimination (Algorithm 9) is applied to get a matrix which is then in reduced row echelon form (again, leading coefficients are allowed to be different from 1). The current representation's degree of redundancy is now directly related to the number of zero rows that the elimination procedure has put forth. Gaussian elimination has cubic straight-line complexity ($O(n^3)$ basic arithmetic operations where $n = MAX(numRows(M), numColumns(M))$). However, greatest common divisor calculations are performed n^2 times at most. Each *gcd* application has polynomial straight-line complexity with respect to length and degree of the input polynomials.

After backward elimination has been applied, column interchange and deletion of zero rows can transform the matrix to have the following form:

$$\begin{matrix} v_1: \\ \vdots \\ v_k: \end{matrix} \begin{pmatrix} p_1 & \dots & 0 & q_{11} & \dots & q_{1m} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & p_k & q_{k1} & \dots & q_{km} \end{pmatrix} =: V$$

The row vectors $v_1, \dots, v_k \in \mathbb{Q}[x]^{n=k+m}$ are referred to as *constraint vectors*. The next step is to find a matrix W that represents a basis of the *syzygy module*. We want W to be of the following form:

$$\begin{matrix} w_1: \\ \vdots \\ w_m: \end{matrix} \begin{pmatrix} r_{11} & \dots & r_{1k} & s_1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ r_{m1} & \dots & r_{mk} & 0 & \dots & s_m \end{pmatrix} =: W$$

Furthermore, all *syzygies* $w_1, \dots, w_m \in \mathbb{Q}[x]^n$ have to be orthogonal to all constraint vectors. Algorithm 10 constructs syzygies taking $O(n^2)$ basic arithmetic operations plus $O(n^2)$ applications of *gcd*. It does not expect its input to be of V's form. It has rather to be a square matrix (zero-rows allowed) such that all leading coefficients are situated on the diagonal.
The syzygies now span up V's syzygy module, i.e. the solution space to our system of equations:

$$S = \{\ u \mid M \cdot u = 0\ \} \quad = \quad \{\lambda_1 w_1^T + \dots + \lambda_m w_m^T \mid \lambda_1, \dots, \lambda_m \in \mathbb{Q}[x]\}$$

3.4 Solution Space Reduction

So far, the solution space has not been reduced, only its representation has been changed. Properties of polynomials with 1-0 coefficients can be used for trying to deduce that variables in u must have the zero polynomial as their only solution. From now on, polynomials that are either the zero polynomial or have at least one coefficient that is smaller than zero will be referred to as *dispensable* polynomials. The following facts can help to reduce the solution space:

Theorem. If a polynomial $p \in \mathbb{Q}[x]$ has a positive root x_0 it is dispensable.

Proof. Let $p(x_0) = a_0 + a_1 \cdot x_0 + \cdots + a_n \cdot x_0^n = 0$, where n is the degree of p and $n > 1$ (there is no polynomial with degree 0 that has a positive root). Let $a_n > 0$
$\Rightarrow \exists a_{i<n} : a_i < 0$
$\Rightarrow p$ dispensable.

□

Theorem. If a linear combination $p = \mu_1 \cdot p_1 + \cdots + \mu_m \cdot p_m$ is dispensable where $p_1, \ldots, p_m \in \mathbb{Q}[x]$, $\mu_1, \ldots, \mu_m \in \mathbb{Q}[x]$ and $\forall \mu_{i \leq n} : \mu_i$ not dispensable
then $\exists p_{i \leq n} : p$ dispensable.

Proof. Obvious: If $\forall p_{i \leq n} : p_i$ not dispensable, then p is dispensable either.

□

Fortunately, the problem of finding dispensable linear combinations can be reduced: Let us assume we can find $\mu_1, \ldots \mu_k \in \mathbb{R}_{\geq 0}$ and an $x \in \mathbb{R}_{>0}$, such that:

$$\begin{aligned} \mu_1 \cdot w_{11}^T(x) + \cdots + \mu_n \cdot w_{1n}^T(x) &= 0 \\ &\vdots \\ \mu_1 \cdot w_{m1}^T(x) + \cdots + \mu_n \cdot w_{mn}^T(x) &= 0 \end{aligned}$$

Taking these coefficients for linear combinations of the single solutions we get:

$$\begin{aligned} & \mu_1 \cdot u_1(x) + \cdots + \mu_n \cdot u_n(x) \\ = \; & \mu_1 \cdot (\lambda_1(x) \cdot w_{11}^T(x) + \cdots + \lambda_m(x) \cdot w_{m1}^T(x)) \; + \cdots + \\ & \mu_n \cdot (\lambda_1(x) \cdot w_{1n}^T(x) + \cdots + \lambda_m(x) \cdot w_{mn}^T(x)) \\ = \; & \lambda_1(x)(\mu_1 \cdot w_{11}^T(x) + \cdots + \mu_n \cdot w_{1n}^T(x)) \; + \cdots + \\ & \lambda_m(x)(\mu_1 \cdot w_{m1}^T(x) + \cdots + \mu_n \cdot w_{mn}^T(x)) \\ = \; & 0 \end{aligned}$$

This yields to the following implication which holds for each $1 \leq l \leq n$:

$$\begin{aligned} & \mu_l > 0 \;\Rightarrow\; u_l(x) = 0 \\ \Leftrightarrow \; & \mu_l > 0 \;\Rightarrow\; u_l \textit{ dispensable} \end{aligned}$$

This allows us to set all u_l to zero for which $\mu_l > 0$ since all solutions that are omitted thereby are dispensable.
Unfortunately, the reduced problem is still quite hard due to the fact that the constraints on $\mu_1, \ldots \mu_n$ and x are non-linear. However, they have a quite simple structure. This forms a subject for further research.

Chapter 4

Conclusion

This thesis shows how a convenient model for communication sequences can be transformed to a set of NFAs in order to abstract the system algebraically. The modifications to the input model as well as the abstraction itself are designed with the objective to derive safety properties without the usual exponential state-space blow-up. As a result, having applied means of at most polynomial complexity, the abstraction is expressed by a basis spanning up a module over $\mathbb{Q}[x]$. This compact representation provides a convenient foundation for further refinement techniques that are needed since the abstraction is still an overapproximation. One approach has been introduced where the refinement problem is reduced to solving non-linear constraints and checking them for satisfiability, respectively. Although the problem is not solved completely yet, the simple constraint structure leaves great hope to be able to reduce the solution space considerable in polynomial complexity.

Bibliography

[Beh] Gerd Behrmann. Uppaal timed automata parser library. `http://www.cs.aau.dk/~behrmann/utap/syntax.html`.

[BET03] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 62–73, New York, NY, USA, 2003. ACM Press.

[BL96] Johan Bengtsson and Fredrik Larsson. *UPPAAL, a Tool for Automatic Verification of Real-Time Systems*. 1996.

[Büc62] Julius R. Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of the 1960 international congress on logic, methodology and philosophy of science*, pages 1–11. Stanford University Press, 1962.

[DdM] Universita di Genova Dipartimento di Matematica. Cocoalib. `http://cocoa.dima.unige.it/cocoalib/`.

[GTWE02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke (Eds.). *Automata Logics, and Infinite Games: A Guide to Current Research*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002.

[Hel05] Keijo Heljanko. Reactive systems. Lecture Slides, `http://www.tcs.hut.fi/Studies/T-79.186/`, Spring 2005.

[Kal87] E. Kaltofen. Single-factor hensel lifting and its application to the straight-line complexity of certain polynomials. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 443–452, New York, NY, USA, 1987. ACM Press.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.

[Mar03] Peter Marwedel. *Embedded System Design*. Kluwer Academic Publishers, 2003.

[oA] Stepfamily Association of America. About communication sequences and patterns. `http://sfhelp.org/pop2/cx_sequences.htm`.

[UU] Uppsala Universitetet and Aalborg University. Uppaal. `http://www.uppaal.com`.

Appendix A

Algorithms

A.1 Definitions and Helper Functions

$\Delta = \Delta_1 \cup \cdots \cup \Delta_n$	All transitions of the system
Σ	All channels of the system
$C = \Sigma \cup \{\epsilon\}$	All channels plus the empty one
$c\_index : C \to P \to P \to int$	Specifies the index of a channel, related to a synchronization between two processes
$channel : \Delta \to C$	Channel on which a transition is synchronized
$curr\_var\_num : int$	Current number of variables, i.e. the length of matrix vectors
$DIR = \{send, receive\}$	Synchronization type
$edges : C \to P \to DIR \to 2^{\Delta}$	Given a channel c, a process P and a synchronization type d, $edges(c)(P)(d)$ provides all transitions in P that are synchronized on c and have synchronization type d
$F = F_1 \cup \cdots \cup F_n$	All goal states of the system
$g\_index : S \to int$	Specifies the index of a goal transition. Every goal transition yields from a unique state.
$gcd : \mathbb{Q}[x] \to \mathbb{Q}[x] \to \mathbb{Q}[x]$	Greatest common divisor of two elements
$getDir : \Delta \to DIR$	Sync. type of a particular transition
$getFreshChannel()$	Delivers a new channel that is not in C yet
$getFreshState()$	Provides a new state that is not in S yet
$getFreshZeroRow : () \to$ MatrixRow*	Instantiates a vector filled with zeros
$goal\_channel : int$	ID of the unique, dedicated goal channel
$in\_out\_trans : S \to 2^{\Delta}$	All self loops of a state (not artificial)
$in\_trans : S \to 2^{\Delta}$	All incoming transitions of a state
$lc\_index : int \to int$	Provides the position of the leading coefficient in a particular row if there is one, 0 otherwise
MatrixRow	Data type representing a vector over $\mathbb{Q}[x]$
M:MatrixRow* Array	Matrix representing the equation system
n	Number of processes in the system
$needsLoop : P \to bool$	Not every process needs artificial self loops. For instance if a process is involved in every possible synchronization, it does not need any.

$num\_vars : int$	Number of abstraction variables
$numColumns : Matrix \rightarrow int$	Number of columns
$numRows : Matrix \rightarrow int$	Number of rows
$out\_trans : S \rightarrow 2^{\Delta}$	All outgoing transitions of a state
$owner : \Delta \rightarrow P$	Returns the process the transition belongs to
$P = \{P_1, \ldots, P_n\}$	All processes of the system
$P' = P \cup \{\perp\}$	$\perp$ describes the process that does not exist.
RingElement	Data type representing $\mathbb{Q}[x]$
$S = S_1 \cup \cdots \cup S_n$	All local states of the system
$S^0 = S_1^0 \cup \cdots \cup S_n^0$	All initial states of the system
$s\_index : S \rightarrow int$	Specifies the index of a particular artificial self loop. Self loops are attached to a unique state.
$syncs : C \rightarrow 2^{(P' \times P')}$	All possible synchronizations on a specific channel
SYZ:MatrixRow* Array	Matrix in which the row vectors form the syzygy module
$t\_index : \Delta \rightarrow P' \rightarrow int$	Specifies the index of a transition, synchronized on a particular process

All means that will be introduced in the following sections have been actually implemented in C++. CoCoALib[DdM] turned out to be a useful tool for representing ring elements in $\mathbb{Q}[x]$ and basic functions on them. For parsing the textual description language (XTA-format), FLEX and BISON have been used.

A.2 Goal Transitions

Algorithm 1 shows how to insert goal transitions. It is obsolete since the following sections will deal with goal transitions in a different context as well. However, the procedure illustrates how they are handled basically. Its complexity is $O(|S|)$.

A.3 Preprocessing

The very first step to do for deriving equations is building an appropriate data structure that describes all possible synchronizations in the system in a compact way. A convenient instrument is function $syncs : C \rightarrow 2^{(P' \times P')}$, set up by Algorithm 2. For each channel it specifies a list of process pairs. Each pair describes a possible synchronization on that specific channel. The first position of the pair relates to the process that sends, the second one is the receiving process. The channel ϵ, describing an empty transition, is handled differently in such a way that it is mapped to all processes that include empty transitions. Consequently, the first and second position of the pair always hold the same process in that case. Another exception are dead transitions. Here, the corresponding field of the pair will hold the symbol $\perp$. The complexity is $O(|C| \cdot n^2)$. Recall the system from Figure 3.1. A corresponding image of *syncs* would look like this:

a: $\{(P_0, P_2), (P_1, P_0), (P_1, P_2)\}$
b: $\{(P_1, P_0), (P_2, P_0), (P_0, \bot)\}$
ϵ: $\{(P_0, P_0)\}$

After that, we can assign a unique identifier to each variable that will occur in the resulting equation system. Each identifier directly corresponds to the index of the particular variable in the final matrix. Algorithm 3 sets up these identifiers in $O(n^2 \cdot (|\Delta| + |C|) + |S|)$ which is also the upper bound on num_vars.

A.4 Building the System of Equations

There are basically four kinds of equations: Algorithm 4 abstracts the flow in $O(|S| \cdot (n \cdot |\Delta| \cdot num\_vars))$. Subsequently, all equations related to synchronization are derived. There are equations on split channels including dead and empty transitions (Algorithm 5, $O(n^2 \cdot |C| \cdot |\Delta| \cdot num\_vars)$), on artificial self loops (Algorithm 6, $O(n \cdot num\_vars)$) and finally on goal transitions (Algorithm 7, $O(n \cdot num\_vars)$).

Algorithm 1 Adding Goal Transitions

Require: $S_1 \cap \cdots \cap S_n = \varnothing$, $\Delta_1 \cap \cdots \cap \Delta_n = \varnothing$
  g $\leftarrow$ $getFreshChannel()$;
  $\Sigma \leftarrow \Sigma \cup \{\text{g}\}$;
  s' $\leftarrow$ $getFreshState()$;
  $S \leftarrow S \cup \{\text{s'}\}$;
  for all i=1 to n **do**
    if $F_i = \varnothing$ **then**
      for all $s \in S_i$ **do**
        $\Delta \leftarrow \Delta \cup (s,\text{g},\text{s'})$;
      end for
    else
      for all $s \in F_i$ **do**
        $\Delta \leftarrow \Delta \cup (s,\text{g},\text{s'})$;
      end for
    end if
  end for

Algorithm 2 Synchronization Abstraction

```
bool sending_found;
bool receiving_found;
bool watch_for_send;
bool watch_for_rec;
for all c ∈ C do
  for all P_i ∈ P do
    if c = ε and edges(ε)(P_i)(send) ≠ ∅ then
      syncs(ε) ← syncs(ε) ∪ {(P_i, P_i)};
      continue;
    end if
    watch_for_rec ← edges(c)(P_i)(send) ≠ ∅;
    receiving_found ← false;
    watch_for_send ← edges(c)(P_i)(receive) ≠ ∅;
    sending_found ← false;
    for all P_j ∈ P do
      if i = j then
        continue;
      end if
      if edges(c)(P_j)(send) ≠ ∅ then
        sending_found ← true;
      end if
      if edges(c)(P_j)(receive) ≠ ∅ then
        receiving_found ← true;
      end if
      if watch_for_rec and edges(c)(P_j)(receive) ≠ ∅ then
        syncs(c) ← syncs(c) ∪ {(P_i, P_j)};
      end if
    end for
    if watch_for_send and not sending_found then
      syncs(c) ← syncs(c) ∪ {(⊥, P_i)};
    end if
    if watch_for_rec and not receiving_found then
      syncs(c) ← syncs(c) ∪ {(P_i, ⊥)};
    end if
  end for
end for
```

Algorithm 3 Variable Indices

Require: $\Delta_1 \cap \cdots \cap \Delta_n = \varnothing$, $S_1 \cap \cdots \cap S_n = \varnothing$, $P_1 \cap \cdots \cap P_n = \varnothing$, $F_1 \cap \cdots \cap F_n = \varnothing$, $\forall i : |F_i| < 2$

```
curr_var_num ← 1;
{Indices of transitions...}
for all t ∈ Δ do
  if channel(t) = ϵ then
    t_index(t)(owner(t)) ← curr_var_num++;
  else
    for all (p1, p2) ∈ syncs(channel(t)) do
      if getDir(t) = send then
        if p1 = owner(t) then
          t_index(t)(p2) ← curr_var_num++;
        end if
      else
        if p2 = owner(t) then
          t_index(t)(p1) ← curr_var_num++;
        end if
      end if
    end for
  end if
end for
{Indices of channels...}
for all c ∈ C do
  for all (p1, p2) ∈ syncs(c) do
    if p1 ≠⊥ and p2 ≠⊥ then
      c_index(c)(p1)(p2) ← curr_var_num++;
    end if
  end for
end for
{Indices of artificial self loops...}
for all i = 1 to n do
  if needsLoop(P_i) then
    for all s ∈ S_i do
      s_index(s) ← curr_var_num++;
    end for
  end if
end for
{Indices of goal transitions...}
for all i = 1 to n do
  if F_i = ∅ then
    for all s ∈ S_i do
      g_index(s) ← curr_var_num++;
    end for
  else
    for all s ∈ F_i do
      g_index(s) ← curr_var_num++;
    end for
  end if
end for
goal_channel ← curr_var_num++;
```

Algorithm 4 Flow Equations

Require: $\forall t \in \Delta, p \in P : t\_index(t)(p) = 0$ if not defined differently by Algorithm 3

```
Matrix M;
MatrixRow* row;
for all i = 1 to n do
  for all s ∈ S_i do
    if (s ∈ S^0 or in_trans(s) ≠ ∅) and
    (out_trans(s) ≠ ∅ or s ∈ F) then
      row ← new MatrixRow;
      if s ∈ S^0 then
        row[0]← −x;
      end if
      if s ∈ F or F_i = ∅ then
        row[g_index(s)]←1;
      end if
      if needsLoop(P_i) then
        row[s_index(s)]← 1 − x;
      end if
      for all t ∈ in_trans(s) do
        for all p ∈ P' do
          if t_index(t)(p) ≠ 0 then
            row[t_index(t)(p)]← −x;
          end if
        end for
      end for
      for all t ∈ out_trans(s) do
        for all p ∈ P' do
          if t_index(t)(p) ≠ 0 then
            row[t_index(t)(p)]← 1;
          end if
        end for
      end for
      for all t ∈ in_out_trans(s) do
        for all p ∈ P' do
          if t_index(t)(p) ≠ 0 then
            row[t_index(t)(p)]← 1 − x;
          end if
        end for
      end for
      M.addRow(row);
    end if
  end for
end for
```

Algorithm 5 Equations on Synchronization Pairs, Empty and Dead Transitions

Require: $\forall c \in C, (p, p') \in syncs(c) : \neg(p = p' = \perp)$

```
Matrix M;
MatrixRow* row;
for all c ∈ C do
  for all (p, p') ∈ syncs(c) do
    {locating some dead transitions}
    if p = ⊥ then
      for all t ∈ edges(c)(p')(receive) do
        row ← new MatrixRow;
        row[t_index(t)(⊥)] ← 1;
        M.addRow(row);
      end for
      continue;
    else if p' = ⊥ then
      for all t ∈ edges(c)(p)(send) do
        row ← new MatrixRow;
        row[t_index(t)(⊥)] ← 1;
        M.addRow(row);
      end for
      continue;
    end if
    {locating empty transitions}
    if p = p' then
      row ← new MatrixRow;
      row[c_index(p)(p')] ← −1;
      for all t ∈ edges(ϵ)(p)(send) do
        row[t_index(t)(p)] ← 1;
      end for
      M.addRow(row);
      continue;
    end if
    {two equations for each (p, p') ∈ syncs(c) : p ≠ p' ∧ p ≠⊥ ∧p' ≠⊥}
    row ← new MatrixRow; {first equation}
    row[c_index(p)(p')] ← −1;
    for all t ∈ edges(c)(p)(send) do
      row[t_index(t)(p')] ← 1;
    end for
    M.addRow(row);
    row ← new MatrixRow; {second equation}
    row[c_index(p)(p')] ← −1;
    for all t ∈ edges(c)(p')(receive) do
      row[t_index(t)(p)] ← 1;
    end for
    M.addRow(row);
  end for
end for
```

Algorithm 6 Equations on Self Loops

```
Matrix M;
MatrixRow* row;
for all i = 1 to n do
  if not needsLoop(P_i) then
    continue;
  end if
  row ← new MatrixRow;
  for all s ∈ S_i do
    row[s_index(s)]← 1;
  end for
  for all c ∈ C do
    for all (p, p') ∈ syncs(c) do
      if p =⊥ or p' =⊥ then
        continue;
      end if
      if p ≠ P_i and p' ≠ P_i then
        row[c_index(p)(p')]← −1;
      end if
    end for
  end for
  M.addRow(row);
end for
```

Algorithm 7 Equations on Goal Transitions

```
Matrix M;
MatrixRow* row;
for all i = 1 to n do
  row ← new MatrixRow;
  row[goal_channel]← −1;
  if F_i ≠ ∅ then
    for all s ∈ F_i do
      row[g_index(s)]← 1;
    end for
  else
    for all s ∈ S_i do
      row[g_index(s)]← 1;
    end for
  end if
  M.addRow(row);
end for
```

Algorithm 8 Gaussian Forward Elimination

```
RingElement factor1;
RingElement factor2;
RingElement curr_gcd;
MatrixRow* temp;
int row_pos = 1;
int pivot_pos;
for i=1 to numColumns(M) do
  pivot_pos ← -1;
  for r = row_pos to numRows(M) do
    if M[r][i] ≠ 0 then
      pivot_pos ← (pivot_pos < 0)?r:pivot_pos;
      if M[r][i] = 1 then
        temp ← M[pivot_pos];
        M[pivot_pos] ← M[r];
        M[r] ← temp;
        break;
      end if
    end if
  end for
  if pivot_pos < 0 then
    continue;
  end if
  for j=pivot_pos+1 to numRows(M) do
    if M[j][i]==0 then
      continue;
    else
      curr_gcd ← gcd(M[j][i],M[pivot_pos][i]);
      factor1 ← M[j][i] / curr_gcd;
      factor2 ← M[pivot_pos][i] / curr_gcd;
      for k=i to numColumns(M) do
        M[j][k] ← M[j][k] * factor2;
        M[j][k] ← M[j][k] - M[pivot_pos][k] * factor1;
      end for
    end if
  end for
  temp ← M[row_pos];
  M[row_pos] ← M[pivot_pos];
  M[pivot_pos] ← temp;
  row_pos++;
end for
```

Algorithm 9 Gaussian Backward Elimination

Require: $numRows$(M) > 1, Gaussian forward elimination has been applied

```
RingElement f1;
RingElement f2;
RingElement curr_gcd;
for i=numRows(M) down to 2 do
  if lc_index(i) = 0 then
    continue;
  end if
  for j=i-1 down to 1 do
    if M[j][lc_index(i)]=0 then
      continue;
    end if
    curr_gcd ← gcd(M[i][lc_index(i)],M[j][lc_index(i)]);
    f1 ← M[i][lc_index(i)] / curr_gcd;
    f2 ← M[j][lc_index(i)] / curr_gcd;
    for k=numColumns(M) down to lc_index(j) do
      M[j][k] ← M[j][k] * f1 - M[i][k] * f2;
    end for
  end for
end for
```

Algorithm 10 Syzygy Module Extraction

Require: Gaussian forward and backward elimination has been applied, $numRows$(M) = $numColumns$(M), leading coefficients on the diagonal

```
RingElement p;
RingElement q;
MatrixRow* new_row;
for i=1 to numRows(M) do
  if lc_index(i)=0 then
    new_row ← getFreshZeroRow();
    p ← 1;
    for j=1 to numRows(M) do
      if lc_index(j) ≠0 then
        q ← M[j][j] / gcd(M[j][j],M[j][i]);
        p ← p * q / gcd(p,q);
      end if
    end for
    new_row[i] ← p;
    for j=1 to numRows(M) do
      if lc_index(j) ≠0 then
        new_row[j] ← -p * M[j][i] / M[j][j];
      end if
    end for
    SYZ.addRow(new_row);
  end if
end for
```