Saarland University

Faculty of Natural Sciences and Technology I
Department of Computer Science

Bounded Synthesis of Distributed Architectures

Bachelor's Thesis

Carolyn Guthoff

Reviewers

Prof. Bernd Finkbeiner
Dr. Swen Jacobs

submitted

September 15th, 2015

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

(Datum / Date) (Unterschrift / Signature)

Abstract

Synthesis derives an implementation that satisfies a certain specification. Contrary to verification, synthesis can be even more time consuming and very complex. A few years ago bounded synthesis was introduced. This approach bounds a system parameter, i.e. its size, and the search space is limited to all implementations within this bound.

We will briefly explain the general approach of bounded synthesis for both monolithic systems and distributed architectures. Afterwards we will look at an input format for architectures. We will compare results of three different SMT solvers: CVC4, Yices and Z3 and look at the difference between resulting transition systems in the mealy or moore format. We will also look at the differences between using consistency or weak symmetry constraints when solving a constraint system of a specification for a distributed architecture and look at results for different types of architectures.

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

In computer science, there are two ways to give a formal guarantee of correctness for a model of an implementation. One is verification and one is synthesis. Verification takes a given implementation and checks whether or not this implementation satisfies a certain specification. Synthesis on the other hand takes a specification and derives an implementation that satisfies the given specification. The advantage of the latter approach is that we do not need to first write code or derive a model and then check whether it satisfies our specification, but rather we let a computer do all the work. However, because synthesis is very complex it may need long computation time.

In 2012 Finkbeiner and Schewe [10] introduced a new approach to synthesis: Bounded Synthesis. Bounded synthesis restricts a system parameter, i.e. its size, and limits the search space to all implementations within this bound. The theoretical approach included ways to do this for monolithic systems and for distributed architectures. Khalimov et al. [11] introduced a tool to do bounded synthesis for monolithic systems. However, so far no tool has been introduced to handle bounded synthesis of distributed architectures. This makes it quite complicated to test bounded synthesis of distributed architectures on a large scale, because it takes a lot of time to write the instances to test it by hand.

The goal of this thesis is to experiment with and evaluate the behavior of bounded synthesis of distributed architectures. Distributed architectures can be found in a lot of different everyday systems like cars or airplanes where we want multiple parts to work together. However these parts may not necessarily need the same input to work, but we still want one specification for the entire system. This is were distributed architectures come into play, because they make it possible for different system processes to get different inputs and grant things independently.

In Chapter 2, we start out with introducing background information we need for bounded synthesis and then continue in Chapter 3 with introducing bounded synthesis for monolithic systems. We explain the approach in detail and then extend it to bounded synthesis of distributed architectures in Chapter 4. For bounded synthesis of distributed architectures, we need an input format for architectures, which will be introduced in Chapter 5. Chapter 6 sheds light on the implementation which is an extension of the tool *Party* by Khalimov et al. [11]. And finally, we take a look at different experiments that include comparing different architectures for three specifications with varying numbers of clients, comparing solvers, output formats and different sets of constraints.

CHAPTER 2

BACKGROUND

We start out with an introduction to the linear-time temporal logic (LTL), get to know transitions systems and look into automata. In the next chapter, we are going to see the formal approach of bounded synthesis for monolithic systems, which will be followed by the expansion to bounded synthesis for distributed systems.

Most of this chapter is based on the paper 'Bounded synthesis' by Finkbeiner and Schewe [10].

2.1 LTL

In this thesis we rely on linear-time temporal logic (LTL) as our specification language and logic.

2.1.1 Syntax

Let Π be a set of atomic propositions. The syntax of LTL is defined over the following grammar:

$$\varphi, \psi ::= true \mid p \mid \neg\varphi \mid \varphi \vee \psi \mid \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \psi$$

where $p \in \Pi$ is an atomic proposition.

$\neg$ is a unary prefix operator, called negation.

$\vee$ is a binary operator, called disjunction.

$\bigcirc$ is a basic temporal modality that is called *Next* operator.

$\mathcal{U}$ is another basic temporal modality that takes two arguments and is called *Until* operator.

While some of the well-known boolean connectives, like conjunction $\wedge$, implication $\rightarrow$ and equivalence $\leftrightarrow$, are not part of LTL's syntax, they can easily be derived:

$$\begin{aligned} \varphi \wedge \psi &\stackrel{\text{def}}{=} \neg(\neg\varphi \vee \neg\psi) \\ \varphi \rightarrow \psi &\stackrel{\text{def}}{=} \neg\varphi \vee \psi \\ \varphi \leftrightarrow \psi &\stackrel{\text{def}}{=} (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi) \end{aligned}$$

The precedence order is as follows.

$$\{\neg, \bigcirc\} > \{\mathcal{U}\} > \{\wedge\} > \{\vee, \rightarrow, \leftrightarrow\}$$

We can additionally derive two temporal modalities using the basic temporal modalities *Next* and *Until*, the *Eventually* operator $\diamond$ and the *Globally* operator $\square$, which are defined as follows:

$$\begin{aligned} \diamond\varphi &\stackrel{\text{def}}{=} true\,\mathcal{U}\,\varphi \\ \square\varphi &\stackrel{\text{def}}{=} \neg\diamond\neg\varphi \end{aligned}$$

The intuitive meaning of $\diamond\varphi$ is, that φ will be eventually true in the future. $\square\varphi$ is satisfied, if there is no moment in the future where $\neg\varphi$ holds. This is equivalent to the statement that φ holds from now on forever.

2.1.2 Semantics

In general, LTL formulas describe properties over infinite words, so that a path either fulfills a LTL formula or it does not.

Definition 1. (Semantics of LTL formulas [10])
Let $\sigma \in \omega \rightarrow 2^{\Pi}$ be an infinite word and $i \in \omega$ a natural number. A sequence $\sigma \in \omega \rightarrow 2^{\Pi}$ is a *model* of an LTL formula φ, denoted by $\sigma \models \varphi$, if, and only if, $\sigma, 0 \models \varphi$. A formula φ holds on the word σ from position i, if $\sigma, i \models \varphi$. The semantics are defined as follows:

$\sigma, i \models p :\Leftrightarrow p \in \sigma(i)$
for atomic propositions $p \in \Pi$. p holds on the word σ at position i, if p is in σ at position i.

$\sigma, i \models \neg\varphi :\Leftrightarrow \sigma, i \not\models \varphi$
where φ is an LTL formula. $\neg\varphi$ is $true$ at position i, if φ is $false$ at time instant i in the model σ.

$\sigma, i \models \varphi \vee \psi :\Leftrightarrow \sigma, i \models \varphi$ **or** $\sigma, i \models \psi$
where φ and ψ are LTL formulas. $\varphi \vee \psi$ is $true$ at position i , if φ is $true$ at position i or ψ is $true$ at position i.

$\sigma, i \models \bigcirc\varphi :\Leftrightarrow \sigma, i+1 \models \varphi$
where φ is LTL formula. $\bigcirc\varphi$ is $true$ at position i, if φ is $true$ at the position $i+1$.

$\sigma, i \models \varphi\,\mathcal{U}\,\psi :\Leftrightarrow \exists n \geq i.\sigma, n \models \psi$ **and** $\forall m \in \{i, ..., n-1\}.\sigma, m \models \varphi$
where φ and ψ are LTL formulas. $\varphi\mathcal{U}\psi$ is $true$ at position i if, and only if, there exists an n greater or equal to i for which ψ is $true$ at time position i and for all $m \in \{i, ..., n-1\}$, ψ is $true$ at position m.

Figure 2.1 shows the intuitive meanings of some LTL formulas.

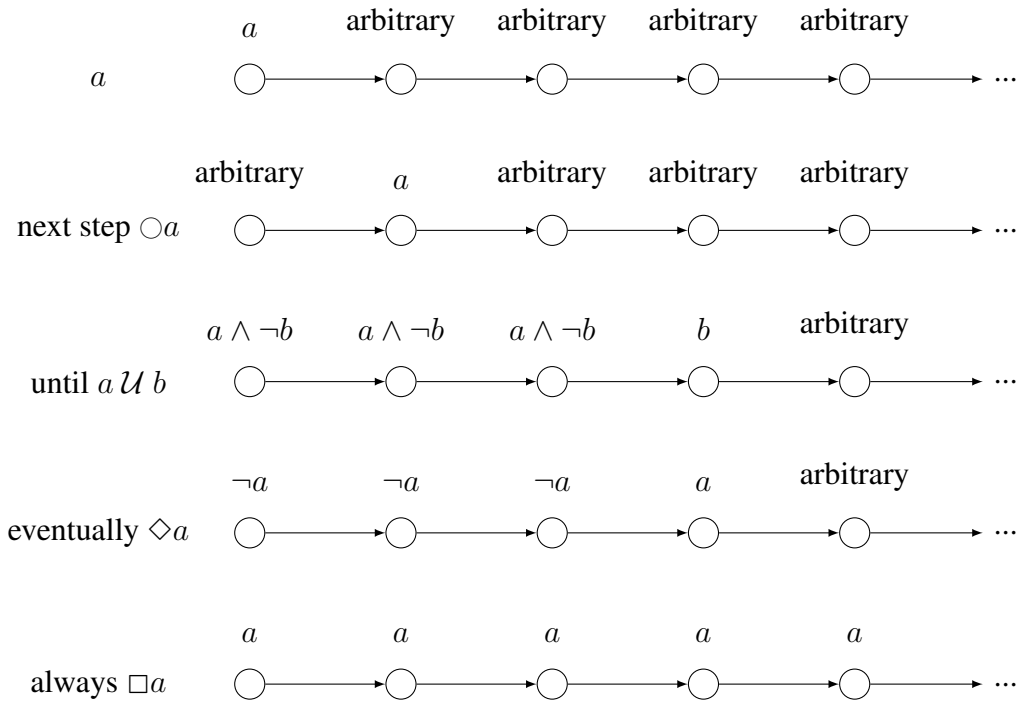


Figure 2.1: Intuitive Semantics of the Introduced Temporal Modalities [2] on Page 233

2.2 Transition Systems

Whenever we want to do synthesis, we want to find an implementation. A way to give such an implementation is as a transition system.

Definition 2. (Transition System)
Let Σ be a set of labels and Υ a finite set of directions. A Σ-labeled Υ-transition system is a tuple $\mathcal{T} = (T, t_0, \tau, o)$ where

- T is a set of states
- $t_0 \in T$ is the initial state
- $\tau : T \times \Upsilon \to T$ is the transition function that maps the pair of a state and a direction (t, υ) to a state
- $o : T \to \Sigma$ is the labeling function that maps a state to a label

$\mathcal{T}$ is finite-state if, and only if, T is finite.

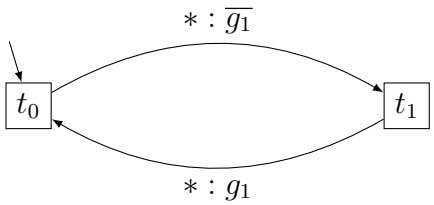
For a given set $\Pi = I \dot{\cup} O$ of atomic propositions, which is the disjoint union of a set I of boolean input variables and a set O of boolean output variables, our goal is to find 2^{Π}-labeled 2^{I}-transition systems that describe implementations. We assume that the initial input of a transition system is some fixed set $i_0 \subseteq I$.

2.2.1 Mealy and Moore Transition Systems

A transition system can be given as either a Mealy or a Moore format. Usually a mealy transition system can be smaller than a moore transition system, because while for mealy the output depends on both the current state and the input, the output for moore only depends on the input. The difference of mealy and moore transition systems to mealy and moore automata, which have a set of inputs and a set of outputs instead of a set of directions and a set of labels, is that the automata can ignore input, while the transition systems have to have a succesor for every input direction.

Mealy Transition System A Mealy transition system shows both the input and the output in the transition which is taken.

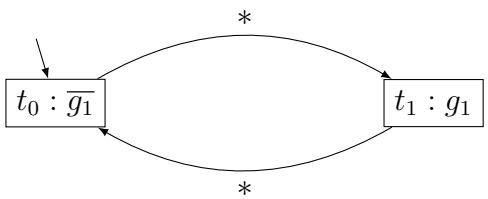
Example 1. (Mealy Transition System)



$*$ denotes any input. In this example, if we get any input and are in state t_0, we get $\overline{g_1}$ as output and end up in state t_1. The same applies to the other direction, only we get g_1 as output.

Moore Transition System A Moore transition system shows the inputs on the transitions and the outputs in the states.

Example 2. (Moore Transition System)



This example does exactly the same thing as the mealy automaton above. The difference is that here the output is mentioned in the state, not on the transition.

2.2.2 Satisfiability

We take a 2^{Π}-labeled 2^{I}-transition system $\mathcal{T} = (T, t_0, \tau, o)$ and an LTL formula φ. Furthermore we take a sequence $\mu : \omega \to T$ that maps a natural number to a state. Last, we take a successor relation

$$\forall i \in \omega.\ \exists \upsilon \in \Upsilon.\ \mu(i+1) \ = \ \tau(\mu(i),\ \upsilon)$$

such that for all numbers i in ω there exists a direction in the set of directions for which it holds that the sequence at moment $i + 1$ equals the successor of the sequence at i in direction υ.

Definition 3. (Satisfiability)
$\mathcal{T}$ satisfies φ if for all sequences μ that start in the initial state $\mu(0) = t_0$ and adhere to the successor relation, the sequence $\sigma_\mu : i \to o(\mu(i))$ that maps a position i to the label of μ at position i is a model of φ.

2.2.3 Input Preservation

A 2^{Π}-labeled 2^{I}-transition system is input preserving, if the input of a state is accurately represented in the output.

Definition 4. (Input Preservation)
If, for all states $t \in T$ and inputs $i \subseteq I$, it holds that $o(\tau(t, i)) \cap I = i$, so the label of the successor of t in direction i intersected with the set of boolean input variables I equals the input i. For the initial state t_0 we additionally require $o(t_0) \cap I = i_0$, so the label of the initial state t_0 intersected with the set of boolean input variables I equals the initial input i_0.

In our case, the set of directions Υ equals the set of interpretations of the boolean input variables 2^I, $\Upsilon = 2^I$, and the set of labels Σ equals the set of atomic propositions Π, $\Sigma = \Pi$. If we would not have the requirement of input preservation, the set of labels Σ would equal the set of boolean output variables of the processes O_{P^-}, $\Sigma = O_{P^-}$. The resulting transition system would in most cases be a lot smaller, however it would also be harder to read.

2.3 Tree Automata

The approach we are using for bounded synthesis translates a specification into a non-deterministic Büchi automaton and then into a universal co-Büchi automaton. To understand these two types of automata, we first introduce a more generalized type of automaton, the alternating parity tree automaton.

Intuitively, an alternating parity tree automaton is an automaton, where the transition function outputs not a single successor state, but rather a combination of states depending on a given direction. In addition, alternating parity tree automata also assign a color to each state which is used to define different accepting conditions.

Definition 5. (Alternating Parity Tree Automaton)
An alternating parity tree automaton is a tuple $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ where

- Σ is a finite set of labels
- Υ is a finite set of directions
- Q is a finite set of states
- $q_0 \in Q$ is a designated initial state
- $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$ is the transition function that gets a state and an input letter as input and has a positive boolean combination of pairs of states and directions as output. See an explanation of positive boolean combination below this definition.
- $\alpha : Q \to C \subset \mathbb{N}$ is a coloring function.

A boolean combination is a formula that is built from variables, conjunctions $\wedge$, disjunctions $\vee$, negations $\neg$, $true$ and $false$. A positive boolean combination does not include negations $\neg$.

Example 3. (Positive Boolean Combination)

$$\begin{aligned}\delta(1, r_1r_2g_1g_2) &= (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge (\bot, r_1r_2) \\ &= (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge false\end{aligned}$$

This is the example of a positive boolean combination where we look at the transition of state 1 in direction $r_1r_2g_1g_2$.

We now have the definition of an alternating parity tree automaton. The way we look at the automaton, it runs on Σ-labeled Υ-transition systems. Whether an automaton accepts a transition system or not is defined in terms of run graphs.

2.3.1 Run Graphs

Definition 6. (Run Graph)
A run graph $\mathcal{G}$ of a parity automaton $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ on a Σ-labeled Υ-transition systems $\mathcal{T} = (T, t_0, \tau, o)$ is a minimal directed graph $\mathcal{G} = (G, E)$ that satisfies the following constraints:

- The vertices $G \subseteq Q \times T$ form a subset of the product of Q and T.
- The pair of initial states $(q_0, t_0) \in G$ is a vertex of $\mathcal{G}$.
- For each vertex $(q, t) \in G$, the set $\{(q', \upsilon) \in Q \times \Upsilon | ((q, t), (q', \tau(t, \upsilon))) \in E\}$ satisfies $\delta(q, o(t))$.

In other words, all elements in G are pairs of states of the automaton and the transition system. The pair of initial states (q_0, t_0) of the automaton and the transition system has to be an element

of G. And finally, each element $(q, t) \in G$ has to fulfill the condition, that its successors satisfy the transition function δ of the automaton when δ is given the state of the automaton q and the label of the direction $o(t)$ as input.

A run graph is now accepting, if every infinite path $g_0 g_1 g_2 ... \in G^\omega$ in the run graph satisfies the parity condition. The parity condition requires that the highest number occuring infinitely often in the sequence $\alpha_0 \alpha_1 \alpha_2 \in \mathbb{N}$ with $\alpha_i = \alpha(q_i)$ and $g_i = (q_i, t_i)$ is even. An automaton accepts a transition system, if the transition system has an accepting run graph corresponding to the automaton.

An example of a run graph can be found at the end of this chapter on Page 11 in Example 6.

Language $\mathcal{L}(\mathcal{A})$ The language $\mathcal{L}(\mathcal{A})$ accepted by an automaton $\mathcal{A}$ is the set of transition systems accepted by $\mathcal{A}$. An automaton is empty if, and only if, its language is empty.

Acceptance Game The acceptance condition can also be modelled as a two-player game. First, player *Accept* gets a pair $(q, t) \in Q \times \Upsilon$ and chooses a set of atoms satisfying $\delta(q, o(t))$. Player *Reject* then chooses one of these atoms, which is executed. The transition system is now accepted, if player *Accept* has a strategy that enforces a path which satisfies the parity condition.

So far we have talked about alternating parity tree automata and run graphs. However there are different kinds of automata we will be using and we are taking a look at them now.

2.3.2 Types of Automata

Non-deterministic Automata A alternating automaton is called non-deterministic, if the image of $\delta(q, t)$ only includes such formulas, that, when rewritten in a disjunctive normal form, contain in every disjunct at most one element of $Q \times \{\upsilon\}$ for each $\upsilon \in \Upsilon$.

Emptiness Game The Emptiness Game is a variation of the Acceptance Game on Page 8. Its goal is to check the emptiness of a non-deterministic automaton. Unlike in the acceptance game, player *Accept* now additionally chooses the label from Σ. If player *Reject* wins the emptiness game, then the non-deterministic automaton is empty.

Universal Automata A universal automaton is another special form of an alternating automaton. The transition function of a universal automaton maps only to positive boolean combinations that contain no disjunctions $\vee$.

Deterministic Automata An automaton that is both universal and non-deterministic is called deterministic.

2.3.3 Büchi Automata

A Büchi automaton is a special parity automaton. A parity automaton is called Büchi automaton, if the image of α is contained in $\{1, 2\}$.

Definition 7. (Büchi Automaton)
A Büchi automaton is an automaton $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, F)$ where

- Σ is a finite set of labels
- Υ is a finite set of directions
- Q is a finite set of states
- $q_0 \in Q$ is a designated initial state
- $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$ is the transition function that gets a state and an input letter as input and has a positive boolean combination of pairs of states and directions as output
- $F \subseteq Q$ is the set of accepting states or states with the higher color

A run graph of a Büchi automaton is accepting, if, on every infinite path, there are infinitely many visits to states in F.

2.3.4 Co-Büchi Automata

A co-Büchi automaton is a special parity automaton. A parity automaton is called co-Büchi automaton, if the image of α is contained in $\{0, 1\}$.

Definition 8. (co-Büchi Automaton)
A co-Büchi automaton is an automaton $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, F)$ where

- Σ is a finite set of labels
- Υ is a finite set of directions
- Q is a finite set of states
- $q_0 \in Q$ is a designated initial state
- $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$ is the transition function that gets a state and an input letter as input and has a positive boolean combination of pairs of states and directions as output
- $F \subseteq Q$ is the set of rejecting states or states with the higher color

A run graph of a co-Büchi automaton is accepting, if, on every infinite path, there are only finitely many visits to states in F.

An example of a universal co-Büchi automaton can be found later in this chapter on Page 10 in Example 4 and in a longer form in the Appendix on Page 62.

2.3.5 Safety Automata

A safety automaton is a special parity automaton. A parity automaton is called safety automaton, if the image of α is contained in $\{0\}$.

Definition 9. (Safety Automaton)
A safety automaton is an automaton $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta)$ where

- Σ is a finite set of labels
- Υ is a finite set of directions
- Q is a finite set of states
- $q_0 \in Q$ is a designated initial state
- $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$ is the transition function that gets a state and an input letter as input and has a positive boolean combination of pairs of states and directions as output

Every run graph of a safety automaton is accepting.

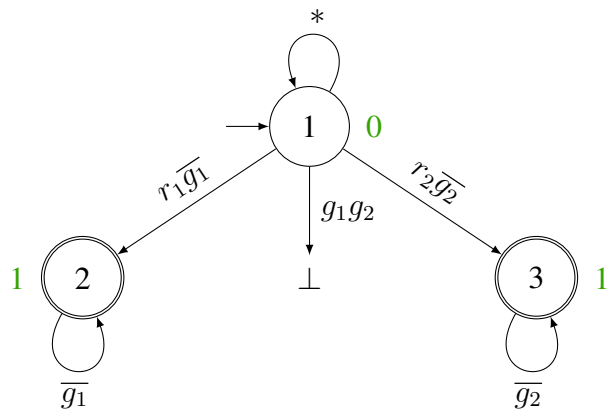
2.3.6 Examples

The following examples all refer to the co-Büchi automaton on Page 62. For simplicity reasons, we show a shorter example of this automaton here. The automaton describes the negated specifiction of

$$\varphi = \square(r_1 \to \diamond g_1) \wedge \square(r_2 \to \diamond g_2) \wedge \square(\neg g_1 \vee \neg g_2)$$

An exact explanation of how we get from this specification to the automaton can be found in the next Chapter.

Example 4. (Universal co-Büchi Automaton)

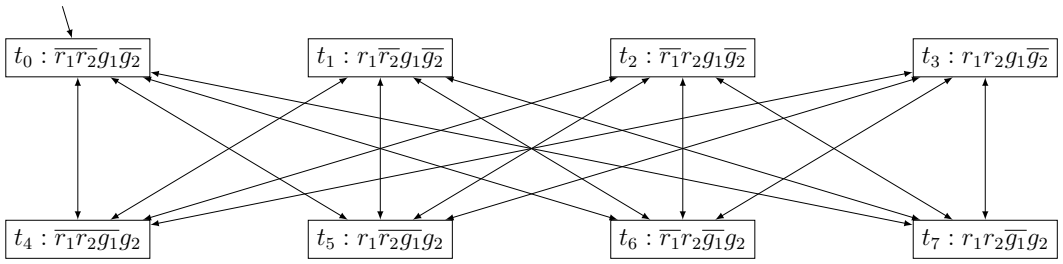


$\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, F)$
$\Sigma = \{r_1, r_2, g_1, g_2\}$
$\Upsilon = \{r_1, r_2\}$
$Q = \{1, 2, 3, \bot\}$
$q_0 = \{1\}$
$F = \{2, 3\}$

The green numbers next to the states are the colors of the states.

In order to look at a run graph, we need a transition system. The following is a transition system, that we will see at the end of the next chapter on Page 23.

Example 5. (Transition System)



We now look at the run graph of the automaton given in Example 4 on the transition system of Example 5.

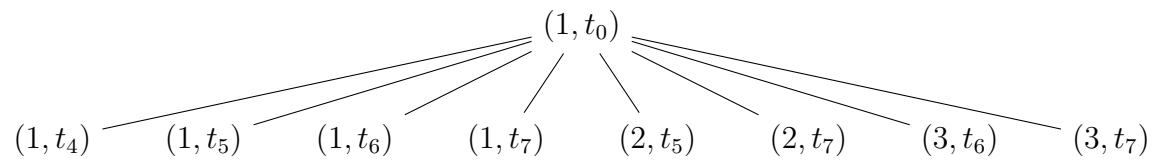
Example 6. (Run Graph)
For simplicity reasons, we start with the pair of initial states as vertex:

$$(1, t_0)$$

We could also start with any other combination of states of the automaton and the transition system and go from there. We now look at what states we can reach from t_0 in the transition system, which are the following:

$$t_4, t_5, t_6, t_7$$

The next step is to check in which states in the automaton we could end up in if we would go in either of the states in the transition system. So for example with t_4 we could stay in state 1, but with t_5 we could stay in state 1, but we could also go to state 2. The following is then the beginning of the run graph:



So far, we fulfill two of the three conditions: the vertices are a subset of the product of Q and T and the pair of initial states is a vertex, because we started with it. That leaves us to check whether the third condition is fulfilled. Let us recall the third condition:

For each vertex $(q,t) \in G$, the set $\{(q',\upsilon) \in Q \times \Upsilon | ((q,t),(q',\tau(t,\upsilon))) \in E\}$ satisfies $\delta(q,o(t))$.

We now have to look that the set of successors of $(1,t_0)$ satisfies $\delta(1,o(t_0))$.

$$\{(1,r_1r_2),(1,r_1\overline{r_2}),(1,\overline{r_1}r_2),(1,\overline{r_1r_2}),(2,r_1r_2),(2,r_1\overline{r_2}),(3,r_1r_2),(3,\overline{r_1}r_2)\} \stackrel{?}{\models} \delta(1,o(t_0))$$

First we can switch out $o(t_0)$ for the label of t_0.

$$\{(1,r_1r_2),(1,r_1\overline{r_2}),(1,\overline{r_1}r_2),(1,\overline{r_1r_2}),(2,r_1r_2),(2,r_1\overline{r_2}),(3,r_1r_2),(3,\overline{r_1}r_2)\} \stackrel{?}{\models} \delta(1,\overline{r_1r_2}g_1\overline{g_2})$$

Now we can check whether the set of pairs of states satisfies the transition function by looking at the transition of state 1 in direction $\overline{r_1r_2}g_1\overline{g_2}$ in the automaton. The transition looks like this:

$$\delta(1,\overline{r_1r_2}g_1\overline{g_2}) = (1,r_1r_2) \wedge (1,r_1\overline{r_2}) \wedge (1,\overline{r_1}r_2) \wedge (1,\overline{r_1r_2})$$

Because all successor states are in the set of successor states for which we want to make sure that is satisfies the transition function, we know that it acctually satisfies the transition function.

$$\{(1,r_1r_2),(1,r_1\overline{r_2}),(1,\overline{r_1}r_2),(1,\overline{r_1r_2}),(2,r_1r_2),(2,r_1\overline{r_2}),(3,r_1r_2),(3,\overline{r_1}r_2)\} \models \delta(1,\overline{r_1r_2}g_1\overline{g_2})$$

Doing this iteratively for all vertices yields the final run graph.

If a path leads to $false$, the run graph stops in that path. The run graph is only accepting for a universal co-Büchi automaton, if the highest color occuring infinitely often is a 0. States marked with the color 0 are not in the set rejecting states and thus do not threaten the acceptance. However, states colored with a 1 can only occur finitely often because otherwise the automaton will reject based on the acceptance condition.

CHAPTER 3

BOUNDED SYNTHESIS OF MONOLITHIC SYSTEMS

In this chapter we look at the approach of Bounded Synthesis of Monolithic Systems as introduced by Finkbeiner and Schewe [10].

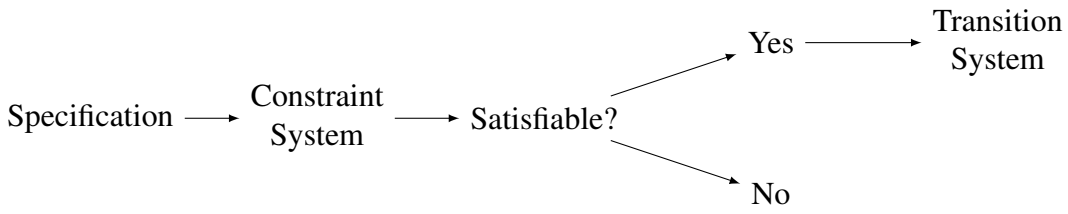


Figure 3.1: Short Overview of Bounded Synthesis for Monolithic Systems

Figure 3.1 gives a short overview of what we are going to see in this chapter. We are going to introduce a monolithic architecture and then transform a specification to a constraint system that can be solved in order to find a implementation for our synthesis problem.

3.1 The Architecture

The architecture used in monolithic systems is a single-process architecture, also known as simple arbiter, with an environment process env and one system process p. The outputs of the environment are the inputs of the system process, $O_{env} = I_p$.

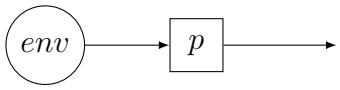


Figure 3.2: Monolithic Architecture

We will defer a formal definition of architectures to Page 24.

The following approach does not explicitly take the architecture into account because it is fully known, meaning every system process gets all environment outputs as input.

3.2 The Synthesis Problem

The Synthesis Problem is the input of the bounded synthesis approach. It includes all information necessary to use the following approach and get a usable result.

Definition 10. (The Synthesis Problem)
The *Synthesis Problem* is a tuple (I, O, i_o, φ) where

- I is a set of *boolean input variables*
- O is a set of *boolean output variables*
- i_0 is the *fixed initial input*, otherwise known as root direction
- φ is an *LTL formula* over the set $\Pi = I \dot{\cup} O$ of the disjoint union over I and O

The synthesis problem specifies the problem we want to solve. Its solution is the answer to the question whether the specification φ is realizable or not. We call φ (finite-state) realizable, if there exists an input preserving 2^{Π}-labeled 2^{I}-transition system which satisfies φ.

The following is an example that will accompany us throughout this and in a slightly different form throughout the next chapter.

Example 7. (The Synthesis Problem)
We consider a synthesis problem (I, O, i_o, φ) of a simple arbiter with the following variable assignments:

- Input variables $I = r_1, r_2$
- Output variables $O = g_1, g_2$
- Fixed initial input $i_0 = \emptyset$
- Specification $\varphi = \square(r_1 \rightarrow \diamond g_1) \wedge \square(r_2 \rightarrow \diamond g_2) \wedge \square(\neg g_1 \vee \neg g_2)$

We have sets of boolean input and output variables with requests and grants, no initial input and a specification that says, that every request should be eventuall followed by a grant. The specification also includes a mutual exclusion property that states, that the grants can never be given at the same time.

3.3 Universal Co-Büchi Automaton

The first thing we have to do is translate the LTL specification to a universal co-Büchi automaton. An overview can be found in Figure 3.3. We first negate the specification, translate it to a non-deterministic Büchi automaton and then simulate the non-deterministic Büchi automaton along each path with a universal co-Büchi automaton. Given the following theorem, we

can then say that if each path in the universal co-Büchi automaton is co-Büchi accepting, the specification φ must hold along every path.

Theorem 1. *[13] Given an LTL formula φ, we can construct a universal co-Büchi automaton $\mathcal{U}_\varphi$ with $2^{O(|\varphi|)}$ states that accepts a transition system $\mathcal{T}$ if, and only if, $\mathcal{T}$ satisfies φ.*

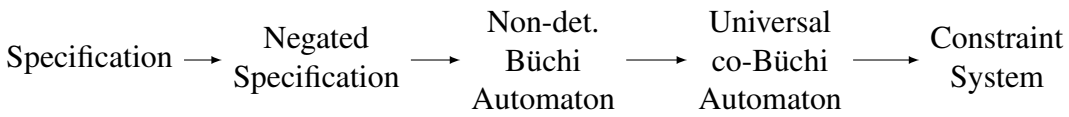


Figure 3.3: First Steps of Bounded Synthesis for Monolithic Architectures

3.3.1 Negating the Specification

Our first step will be to negate the specification.

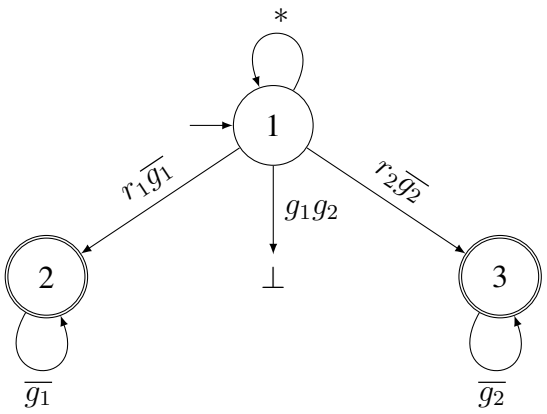
Example 8. (Negating the Specification)

$\neg\varphi = \neg(\square(r_1 \rightarrow \diamond g_1) \wedge \square(r_2 \rightarrow \diamond g_2) \wedge \square(\neg g_1 \vee \neg g_2))$

3.3.2 Non-deterministic Büchi Automaton $\mathcal{A}_\varphi$

Our next step will be to translate the negated specification to a non-deterministic Büchi automaton $\mathcal{A}_\varphi$. For this we can use a tool that can translate a LTL formula to a non-deterministic Büchi automaton like LTL3BA [1].

Example 9. (Non-deterministic Büchi Automaton)
$\mathcal{A}_\varphi = (\Sigma, \Upsilon, Q, q_0, \delta, F)$
$\Sigma = \{r_1, r_2, g_1, g_2\}$
$\Upsilon = \{r_1, r_2\}$
$Q = \{1, 2, 3, \bot\}$
$q_0 = \{1\}$
$\delta : Q \times \Sigma \rightarrow \mathbb{B}^+(Q \times \Upsilon)$
$F = \{2, 3\}$



This is the non-deterministic Büchi automaton, corresponding to the specification in Example 8 that we get as output of LTL3BA.

3.3.3 Universal Co-Büchi Automaton $\mathcal{U}_\varphi$

Since we want to build a constraint system, we need a universal co-Büchi automaton instead of a non-deterministic Büchi automaton. Luckily, it is fairly easy to translate a non-deterministic Büchi automaton to a universal co-Büchi automaton. The transition function and the acceptance condition change. Instead of disjunctions $\vee$, the transition function now uses conjunctions $\wedge$ to connect positive boolean combinations atoms that are pairs of states and directions. Accepting states are only visited finitely often. The actual automaton still looks the same though. For a better understanding see Subsection 2.3 in the last Chapter, starting on Page 6.

Example 10. (Universal Co-Büchi Automaton)

$$\mathcal{U}_\varphi = (\Sigma, \Upsilon, Q, q_0, \delta, F)$$

The automaton looks exactly the same as in example 9. A description of the entire automaton can be found in the appendix on page 62.

3.4 Annotated Transition Systems

We have a universal co-Büchi automaton $\mathcal{U}_\varphi$ and thus know that every transition system has a unique run graph. See Subsection 2.3.1 on Page 7 for further information about run graphs. The annotation function for labeled transition systems

$$\lambda : Q \times T \to \{\_\} \cup \mathbb{N}$$

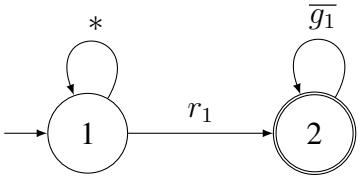
assigns a pair (q, t) of states of the automaton and the transition system to either a blank sign or a natural number. The natural number indicates the maximal number of rejecting states that occur on some path to (q, t) in the run graph. This ensures that critical paths, which are paths that include rejecting states, are only finitely long.

Transition systems that have an annotation function that only assigns natural numbers to the

vertices of the run graph thus have an upper bound on the number of rejecting states visited. We call such annotations valid. Transition systems with a valid annotation are accepted by the universal co-Büchi automaton.

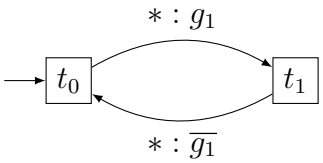
Example 11. (Annotation Function)

Let us first take a look at the universal co-Büchi automaton $\mathcal{U}_\psi$ that describes the specification $\psi = \square(r_1 \rightarrow \diamond g_1)$. So every request r_1 is eventually followed by a grant g_1.

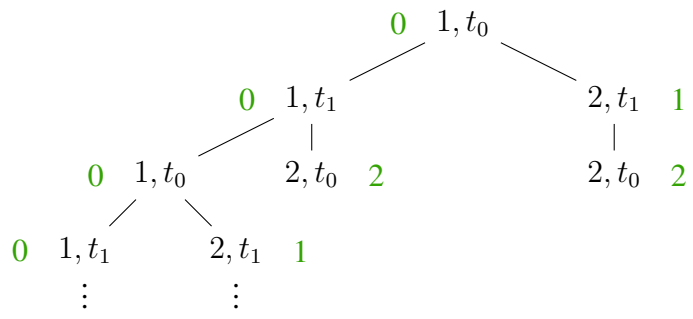


The specification describes a liveness property. If we get a request r_1 we eventually have to follow it up with a grant g_1. If we do not, we will stay forever in state 2. Since we want the automaton to accept, we need the annotation function to assign a natural number to each combination of a state of the automaton and a state of the transition system. If the automaton would however not accept, the annotation function should never be able to find a natural number.

Let us say we have the following transition system $\mathcal{T}$.



The run graph $\mathcal{G}$ of the transition system $\mathcal{T}$ and the universal co-Büchi automaton $\mathcal{U}_\psi$ looks as follows.



The green colors on the left and right of the states represent the natural numbers which the annotation function assigned to these vertices. We do not have to care about the states that are labeled with a 0, because state 1 is not a rejecting state. Because we are looking at a universal automaton, every path in the run graph has to be accepting. Another thing is, that

we look at all paths at the same time. And if we now again look at $\mathcal{G}$, we see that whenever we are in a vertex labeled with $(2, t_0)$ or $(2, t_1)$, the path stops with this vertex or with the next vertex. We can conclude, that the rejecting states are only visited finitely often. Note that the vertex $(2, t_0)$ is always labeled with a 2 because the same vertex is always labeled with the same number, and the highest number the vertex $(2, t_0)$ can be labeled with is a 2.

3.5 The Constraint System

Our goal now is to find a transition system that is accepted by a universal co-Büchi automaton. Therefore, we represent the unknown transition system and its annotation by uninterpreted functions in a constraint system.

3.5.1 Encoding the Transition Function

We start out with encoding the transition function of the transition system. To make things a little easier we recall the definition of a transition system:

$$\mathcal{T} = (T, t_0, \tau, o)$$

And we recall the corresponding transition function:

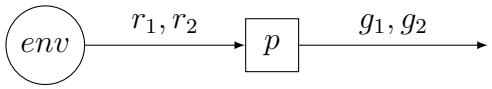
$$\tau : T \times \Upsilon \to T$$

We introduce a unary function symbol τ_υ for every output of the environment $\upsilon \subseteq O_{env}$.

$$\tau_\upsilon(t) = \tau(t, \upsilon)$$

Intuitively, τ_υ maps a state t of the transition system to its υ-successor, meaning to the successor in direction υ. Basically it is another way to write $\tau(t, \upsilon)$. Since the transition function is encoded as a function, it has to have a successor for every direction.

Example 12. (Encoding the Transition Function)
Let us take a look at our architecture:



This architecture consists of two processes $P = \{env, p\}$, one environment process env and one system process p. We can also see that the outputs of the environment are the inputs of the system process, which also equals the set of inputs we get from the synthesis problem:

$$O_{env} = \{r_1, r_2\} = I_p = I$$

The directions are a subset of the output of the environment:

$$\upsilon \subseteq \{r_1 r_2, r_1 \overline{r_2}, \overline{r_1} r_2, \overline{r_1 r_2}\}$$

3.5.2 Encoding the Labeling Function

Next, we want to encode the labeling function of the transition system:

$$o : T \to \Sigma$$

We first need to introduce V, which is the the set of all boolean system variables and since we look at simple arbiter, V equals the disjoint union of the set of inputs and the set of outputs:

$$V = I \dot{\cup} O = \Pi$$

We use a unary predicate symbol a for every $a \in V$. We can define a as a function from a state $t \in T$ to a boolean.

$$a : T \to \mathbb{B}$$

Intuitively, a maps a state t of the transition system to $true$ if, and only if, it is part of the label $o(t)$.

$$a(t) = true \Leftrightarrow a \in o(t)$$

So a shows, whether a symbol is part of a label or not. We introduce one function for every variable in V.

Example 13. (Encoding the Labeling Function)
Let us say we have the following set of boolean system variables

$$V = \{r_1, r_2, g_1, g_2\}$$

and the following state of the transition system

$$t_0 = \overline{r_1 r_2} g_1 \overline{g_2}$$

If we now want to check, whether r_1 is part of the label of t_0, we can do this as follows.

$$r_1(t_0) = true \Leftrightarrow r_1 \in o(t_0)$$

However, when exchanging $o(t_0)$ with the actual label, we can see that r_1 is not part of the label, so r_1 of t_0 maps to $false$.

$$r_1 \notin \overline{r_1 r_2} g_1 \overline{g_2} \Rightarrow r_1(t_0) = false$$

On the other hand, if we want to check, whether $\overline{r_1}$ is part of the label of t_0, the labeling of the variable may look like this.

$$\overline{r_1}(t_0) = true \Leftrightarrow \overline{r_1} \in o(t_0)$$

When exchanging $o(t_0)$ with the actual label, we see that it maps to $true$.

$$\overline{r_1} \in \overline{r_1 r_2} g_1 \overline{g_2} \Rightarrow \overline{r_1}(t_0) = true$$

3.5.3 Input Preservation

With the knowledge of how to encode the transition function and the labeling function, we can derive our first constraint: input preservation. Input preservation basically states that the label of a state t in the transition system must accurately reflect its input. For example if we are in state t_0 and we get $r_1 r_2$ as input, we want the input $r_1 r_2$ to be reflected in the label of the state which we will reach from t_0 with input $r_1 r_2$.

For input preservation, we add a constraint for each output of the environment and each direction υ. We have the choice between the following two:

$$\begin{array}{ll} \text{constraint } \forall t. a(\tau_\upsilon(t)) & \text{if } a \in \upsilon \\ \text{constraint } \forall t. \neg a(\tau_\upsilon(t)) & \text{if } a \notin \upsilon \end{array}$$

We pick the first constraint if the output is in the direction and the second one, if it is not.

Example 14. (Input Preservation)
Looking back at Example 12, we know that the environment outputs and the directions look as follows:

$$O_{env} = \{r_1, r_2\}$$

$$\upsilon \subseteq \{r_1 r_2, r_1 \overline{r_2}, \overline{r_1} r_2, \overline{r_1 r_2}\}$$

Let us take a look at $a = r_1$. If we have the direction $\upsilon = r_1 r_2$, then $r_1 \in \upsilon$ and thus we add the first constraint:

$$\forall t. r_1(\tau_{r_1 r_2}(t))$$

However, if we have the direction $\upsilon = \overline{r_1} r_2$, then $r_1 \notin \upsilon$ and thus we add the second constraint:

$$\forall t. \neg r_1(\tau_{\overline{r}_1 r_2}(t))$$

When doing this for all outputs of the environment and for all directions, we get the following constraint for our constraint system:

$$\forall t. r_1(\tau_{r_1 r_2}(t)) \wedge r_2(\tau_{r_1 r_2}(t)) \wedge r_1(\tau_{r_1 \overline{r}_2}(t)) \wedge \neg r_2(\tau_{r_1 \overline{r}_2}(t)) \wedge \neg r_1(\tau_{\overline{r}_1 r_2}(t)) \wedge r_2(\tau_{\overline{r}_1 r_2}(t)) \wedge \neg r_1(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \neg r_2(\tau_{\overline{r}_1 \overline{r}_2}(t))$$

3.5.4 Encoding the Annotation Function

Next, we have to encode the annotation function:

$$\lambda : Q \times T \to \{\_\} \cup \mathbb{N}$$

We first introduce the unary predicate symbol $\lambda_q^{\mathbb{B}}$ that maps a state t of the transition system to $true$ if, and only if, $\lambda(q, t)$ is a natural number:

$$\lambda_q^{\mathbb{B}}(t) = true \Leftrightarrow \lambda(q, t) \in \mathbb{N}$$

We also introduce the unary function symbol $\lambda_q^{\#}$ that maps a state t of the transition system to either a natural number if $\lambda(q, t)$ is a natural number or to unconstrained if it is not.

$$\lambda_q^{\#}(t) = \begin{cases} \lambda(q, t) & \text{if } \lambda(q, t) \in \mathbb{N} \\ \text{unconstrained} & \text{if } \lambda(q, t) = \_ \end{cases}$$

We only need $\lambda_q^{\#}(t)$ to encode liveness properties like the one you can see in Example 11 on Page 17.

An example for the encoding of an annotation function in the constraint system can be found in the next subsection on Page 21 in Example 15.

3.5.5 Encoding the Transitions

We now have encodings for the transition and the labeling functions of the transition system as well as an encoding for the annotation function. With this knowledge we can formalize that an annotation of a transition system is valid by the following first-order constraints:

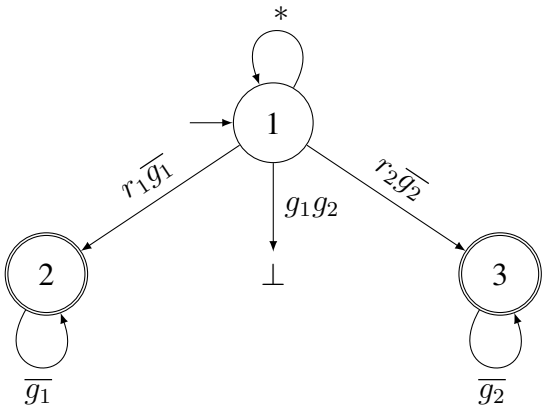
$$\forall t. \lambda_q^{\mathbb{B}}(t) \wedge (q', \upsilon) \in \delta(q, \overrightarrow{a}(t)) \rightarrow \lambda_{q'}^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_{q'}^{\#}(\tau_\upsilon(t)) \triangleright_{q'} \lambda_q^{\#}(t)$$

$\overrightarrow{a}(t)$ represents the label $o(t)$, $(q', \upsilon) \in \delta(q, \overrightarrow{a}(t))$ will be replaced with the corresponding propositional formula and $\triangleright_{q'} \equiv >$ if $q' \in F$ or $\triangleright_{q'} \equiv \geq$ otherwise.

The constraint says that if $\lambda(q, t)$ is $true$ and the propositional formula corresponding to the transition that we want to formalize is a successor of q in direction $\overrightarrow{a}(t)$, then $\lambda_{q'}^{\mathbb{B}}(\tau_\upsilon(t))$ is $true$ and $\lambda_{q'}^{\#}(\tau_\upsilon(t))$ is greater or greater equal than $\lambda_q^{\#}(t)$, depending on whether q' is an accepting state or not.

The part $\lambda_{q'}^{\#}(\tau_\upsilon(t)) \triangleright_{q'} \lambda_q^{\#}(t)$ is only needed, if q has a loop and is a rejecting state, $q \in F$.

Example 15. (Encoding the Transitions)
Let us take a look at our universal co-Büchi automaton that represents the specification:



We want to encode the transition from state 1 of the automaton to state 2. We have 4 cases, one for each direction, which leads to this constraint:

$$\begin{aligned}
\forall t. \lambda_1^{\mathbb{B}}(t) \wedge r_1(t) \quad &\rightarrow \quad \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_1^{\#}(t) \\
&\wedge \quad \lambda_2^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_1^{\#}(t) \\
&\wedge \quad \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#}(t) \\
&\wedge \quad \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}(t)
\end{aligned}$$

For the full constraint system we have to encode every transition.

3.5.6 Initialization

Finally, we have to encode the initialization. If we look back at our synthesis problem, we remember the root direction i_0, which we have to encode. Moreover, we require that the pair of initial states will be labeled by a natural number. Usually the initial state of the transition system is labeled with 0.

Example 16. (Initialization)
In our example, the root direction is: $i_0 = \emptyset$. This means that our initial inputs have to be $false$. We can encode this and the requirement that the pair of initial states should be labeled by a natural number with the following constraint:

$$\lambda_1^{\mathbb{B}}(0) \wedge \neg r_1(0) \wedge \neg r_2(0)$$

$\lambda_1^{\mathbb{B}}(0)$ represents the initial state, while $\neg r_1(0)$ and $\neg r_2(0)$ represent the input direction in the transition system.

3.5.7 The Final Constraint System

Now we know everything we need in order to built the constraint system for any specification for a simple arbiter.

Example 17. (The Final Constraint System)
The constraint system for our example specification looks as follows:

1. $\forall t. r_1(\tau_{r_1 r_2}(t)) \wedge r_2(\tau_{r_1 r_2}(t)) \wedge r_1(\tau_{r_1 \bar{r}_2}(t)) \wedge \neg r_2(\tau_{r_1 \bar{r}_2}(t)) \wedge \neg r_1(\tau_{\bar{r}_1 r_2}(t)) \wedge r_2(\tau_{\bar{r}_1 r_2}(t)) \wedge \neg r_1(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \neg r_2(\tau_{\bar{r}_1 \bar{r}_2}(t))$
2. $\lambda_1^{\mathbb{B}}(0) \wedge \neg r_1(0) \wedge \neg r_2(0)$
3. $\forall t. \lambda_1^{\mathbb{B}}(t) \rightarrow \lambda_1^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \geq \lambda_1^{\#} \wedge \lambda_1^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 \bar{r}_2}(t)) \geq \lambda_1^{\#}$
   $\wedge\ \lambda_1^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{\bar{r}_1 r_2}(t)) \geq \lambda_1^{\#} \wedge \lambda_1^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 r_2}(t)) \geq \lambda_1^{\#}$
4. $\forall t. \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(t) \vee \neg g_2(t)$
5. $\forall t. \lambda_1^{\mathbb{B}}(t) \wedge r_1(t) \wedge \neg g_1(t) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) > \lambda_1^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \bar{r}_2}(t)) > \lambda_1^{\#}$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\bar{r}_1 r_2}(t)) > \lambda_1^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}$
6. $\forall t. \lambda_1^{\mathbb{B}}(t) \wedge r_2(t) \wedge \neg g_2(t) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) > \lambda_1^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \bar{r}_2}(t)) > \lambda_1^{\#}$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\bar{r}_1 r_2}(t)) > \lambda_1^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}$
7. $\forall t. \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(t) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) > \lambda_2^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \bar{r}_2}(t)) > \lambda_2^{\#}$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\bar{r}_1 r_2}(t)) > \lambda_2^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_2^{\#}$
8. $\forall t. \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(t) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) > \lambda_3^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \bar{r}_2}(t)) > \lambda_3^{\#}$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\bar{r}_1 r_2}(t)) > \lambda_3^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_3^{\#}$

Constraint 1 is the input preservation, constraint 2 is the initialization and constraints 3-10 encode transitions of the universal co-Büchi automaton.

3.6 Interpretation

The last step is solving the constraint system. For this task, SMT solvers like Z3, CVC4 and Yices can be used. See Page 37 and following for an introduction to these SMT solvers.

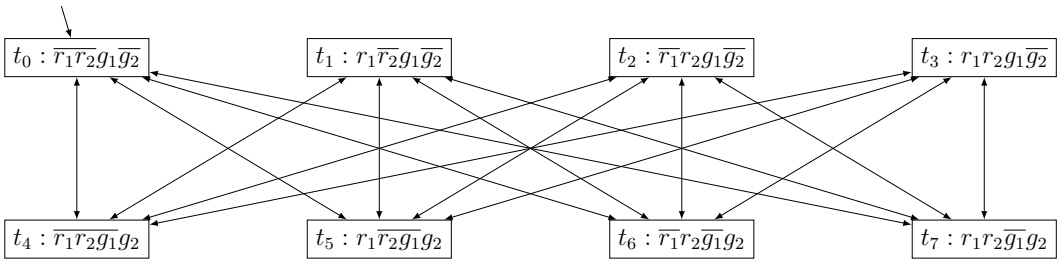
SMT stands for satisfiability modulo theories. We use boolean satisfiability (SAT), data types, uninterpreted functions and universal quantification in our constraint system to solve bounded synthesis. The model of the constraint system is then the interpretation of the functions and variables such that everything is satisfied.

Of course the constraint system has to first be rewritten to a format readable to SMT solvers. Party [11] can do that job for us. It is a tool for monolithic and parameterized synthesis which lets a user input a specification with inputs and outputs and figures out if the specification is satisfiable or not. See Page 37 for more information about Party.

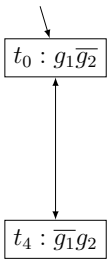
3.7 The Result

The result of an SMT solver is the outcome of the satisfiability check. Fortunately, if a constraint system is satisfiable, most SMT solvers can give a model which can be presented as a transition system.

Example 18. (The Result)
The transition system for our synthesis problem looks like this:



If we remove input preservation, the transition system looks like this:



Now that we have our transition system we can check with a run graph, whether the universal co-Büchi automaton really accepts the transition system or not. The beginning of the corresponding run graph can be found at the end of the previous Chapter on Page 11.

CHAPTER 4

BOUNDED SYNTHESIS OF DISTRIBUTED ARCHITECTURES

In the last chapter, we saw bounded synthesis for monolithic systems. We now look at bounded synthesis for distributed architectures, which means that we have one more parameter, namely the architecture to look at. This chapter is based on the paper "Bounded synthesis" by Finkbeiner and Schewe [10].

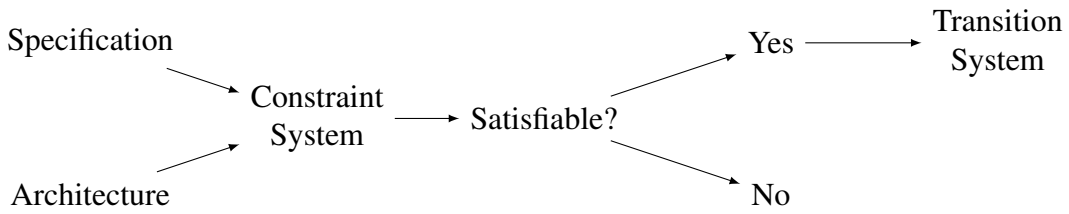


Figure 4.1: Short Overview of Bounded Synthesis for Distributed Architectures

Instead of only having the specification as input, we now also get an architecture as input.

4.1 The Architecture

Architectures used for bounded synthesis of distributed architectures need more components in the definition than the simple arbiter we introduced in the last chapter, because we do not look at architectures with only one system process anymore.

Definition 11. (Architecture)
An architecture is a tuple $A = (P, env, V, I, O)$ where

- P is a set of system processes and the designated environment process, $P^- = P \setminus \{env\}$ is the set of system processes
- env is the designated environment process
- V is the set of boolean system variables with $V = \bigcup_{p \in O} O_p$

- I assigns a set of input variables to each system process, $I = \{I_p \subseteq V | p \in P^-\}$
- O assigns a set of output variables to each process, $O = \{O_p \subseteq V | p \in P\}$

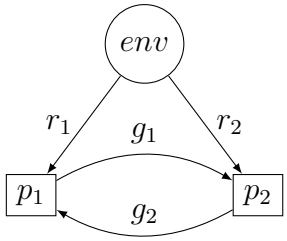
The same variable may occur in multiple sets in I to indicate broadcasting. All sets in O are however disjoint.

An architecture is said to be *fully informed* if, for all system processes, the inputs of a system process equals the output of the environment.

$$\forall p \in P^-.I_p = O_{env}$$

In Chapter 3 we already saw a monolithic architecture in Figure 3.1. The following example shows a distributed architecture.

Example 19. (Distributed Architecture)



$A = (P, env, V, I, O)$

$P = \{env, p_1, p_2\}$ $\quad P^- = \{p_1, p_2\}$

$V = \{r_1, r_2, g_1, g_2\}$

$I = \{r_1, r_2, g_1, g_2\}$

$I_{p_1} = \{r_1, g_2\}$ $\quad I_{p_2} = \{r_2, g_1\}$

$O = \{r_1, r_2, g_1, g_2\}$ $\quad O_{env} = \{r_1, r_2\}$

$O_{p_1} = \{g_1\}$ $\quad O_{p_2} = \{g_2\}$

The architecture is not fully informed because neither p_1 nor p_2 get all outputs of the environment O_{env} as inputs: $O_{env} \neq I_{p_1} \neq I_{p_2}$

4.2 The Synthesis Problem

We have to specify the problem we want to solve. The synthesis problem for bounded synthesis of monolithic architectures, which was specified in Definition 10 on Page 14, needed a set of boolean input variables, a set of boolean output variables, a fixed initial input and a specification. The set of boolean input variables and the set of boolean output variables did nothing else, than describing the architecture. So now we only have to replace those two sets with the architecture that we get as input.

Definition 12. (The Synthesis Problem)
The synthesis problem for distributed architectures is a tuple (φ, A, i_0) where

- φ is the specification
- A is the architecture
- i_0 is the fixed initial input with $i_0 \subseteq O_{env}$

Example 20. (The Synthesis Problem)
The following is an example of a synthesis problem. We will see it throughout the rest of this chapter.

$$(\varphi, A, \emptyset)$$

where

$$\varphi = \square(r_1 \rightarrow \diamond g_1) \wedge \square(r_2 \rightarrow \diamond g_2) \wedge \square(\neg g_1 \vee \neg g_2)$$

and A is the architecture from Example 19.

4.3 The Transition System

Our goal for bounded synthesis of distributed architectures is to find distributed implementations. Each system process $p \in P^-$ is implemented as a 2^{O_p}-labeled 2^{I_p}-transition system $\mathcal{T}_p = (T_p, t_0^p, \tau_p, o_p)$. The only constriction of the environment process env is the initial input to the system processes, $i_0 \subseteq O_{env}$.

Definition 13. (Transition System of a Distributed Architecture)
Let $P^- = \{p_1, ..., p_k\}$. We are looking for the composition of process implementations which is a 2^V-labeled $2^{O_{env}}$-transition system $\mathcal{T}_A = (T, t_0, \tau, o)$ where

- T is a set of states
- t_0 is the initial state
- τ is the transition function
- o is the labeling function

The following subsections will introduce this construction in detail and give examples about what this transition system looks like.

The Set of States The set of states T is formed by the product of the states of the process transition systems and the environment output.

$$T = T_{p_1} \times ... \times T_{p_k} \times 2^{O_{env}}$$

We will see an example in a second after introducing the initial state.

The Initial State The initial state t_0 is a tuple consisting of the initial states of the process transition systems and the initial environment output e_0.

$$t_0 = (t_0^{p_1}, ..., t_0^{p_k}, e_0)$$

Example 21. (The Initial State)
Example 20 has two system processes, $P^- = \{p_1, p_2\}$, and our environment process has the following outputs: $O_{env} = \{r_1, r_2\}$. From Example 20 we also know that the transition system has no input direction, so we will take $\overline{r_1 r_2}$ as our root direction. Combining this information we get the following start state:

$$t_0 = (t_0^{p_1}, t_0^{p_2}, \overline{r_1 r_2})$$

t_0 is a tuple of the start states of the system process transition systems and the root direction of the synthesis problem.

4.3.1 The Labeling Function

The labeling function labels each state with the union of the labels of the process transition system states and the environment output.

$$o(s_{p_1}, ..., s_{p_k}, e) = o(s_{p_1}) \cup .... \cup o(s_{p_k}) \cup e$$

Example 22. (The Labeling Function)
To see the labeling function in action, we look at the previous Example 21:

$$t_0 = (t_0^{p_1}, t_0^{p_2}, \overline{r_1 r_2})$$

Labeling t_0 now looks as follows:

$$\begin{aligned} o(t_0^{p_1}, t_0^{p_2}, \overline{r_1 r_2}) &= o(t_0^{p_1}) \cup o(t_0^{p_2}) \cup \overline{r_1 r_2} \\ &= \overline{g_1} \cup g_2 \cup \overline{r_1 r_2} \\ &= \overline{g_1} g_2 \overline{r_1 r_2} \end{aligned}$$

4.3.2 The Transition Function

The transition function updates for each process $p \in P^-$ the state of the process transition system according to the transition function τ_p using the visible part of the state label as input and also updates the output value of the state with the new output from the environment.

$$\tau((s_{p_1}, ..., s_{p_k}, e), e') = (\tau_{p_1}(s_{p_1}, l \cap I_{p_1}), ..., (\tau_{p_k}(s_{p_k}, l \cap I_{p_k}), e')$$

where $l = o(s_{p_1}, ..., s_{p_k}, e)$.

Example 23. (The Transition Function)
Consider the state $t_0 = (t_0^{p_1}, t_0^{p_2}, \overline{r_1 r_2})$ from which we want to go into direction $\overline{r_1} r_2$:

$$\tau((t_0^{p_1}, t_0^{p_2}, \overline{r_1 r_2}), \overline{r_1} r_2)$$

We know what the corresponding l is from the previous Example 22:

$$l = o(t_0^{p_1}, t_0^{p_2}, \overline{r_1 r_2}) = \overline{g_1} g_2 \overline{r_1 r_2}$$

Looking back at our architecture, we have the following inputs:

$$I_{p_1} = \{r_1, g_2\} \qquad I_{p_2} = \{r_2, g_1\}$$

We now have to intersect l with the input sets:

$$l \cap I_{p_1} = g_2 \overline{r_1} \qquad l \cap I_{p_2} = \overline{g_2 r_2}$$

Next, we can trace the transition back to the transitions of the system process transition systems:

$$\begin{aligned} \tau((t_0^{p_1}, t_0^{p_2}, \overline{r_1 r_2}), \overline{r_1 r_2}) &= (\tau_{p_1}(t_0^{p_1}, l \cap I_{p_1}), \tau_{p_2}(t_0^{p_2}, l \cap I_{p_1}), \overline{r_1} r_2) \\ &= (\tau_{p_1}(t_0^{p_1}, g_2 \overline{r_1}), \tau_{p_2}(t_0^{p_2}, \overline{g_2 r_2}), \overline{r_1} r_2) \\ &= (t_1^{p_1}, t_1^{p_2}, \overline{r_1} r_2) \end{aligned}$$

Overall, if we go from state $(t_0^{p_1}, t_0^{p_2}, \overline{r_1 r_2})$ into direction $\overline{r_1} r_2$, we end up in state $(t_1^{p_1}, t_1^{p_2}, \overline{r_1} r_2)$.

4.4 The Constraint System

For bounded synthesis of distributed systems, our goal is to find a family of transition systems

$$\{\mathcal{T}_p = (T_p, t_o^p, \tau_p, o_p) | p \in P^-\}$$

such that their composition is a transition system that satisfies a given specification.

In order to do this, we have to adapt the constraint system we constructed for monolithic synthesis.

4.4.1 Decomposing the State Space

In our first step we decompose the global state space of the combined transition system. To this end, we introduce a unary function symbol d_p that maps a state of the product state space to its p-component.

Example 24. (Decomposition of the Global State Space)

$$\begin{aligned} d_{p_1}(t_0) &= d_{p_1}(t_0^{p_1}, t_0^{p_2}, \overline{r_1 r_2}) \\ &= t_0^{p_1} \end{aligned}$$

d_{p_1} maps t_0 to the system process transition system state of p_1 in order to make sure, that the calling function can only access the part it has rights to access.

If we would not restrict the access, processes could access hidden variables they should neither be able to see nor change.

To better understand the reason for this, we first look at the constraint system we derived in Chapter 3 and then decompose the global state space by restricting all variables that are not global.

Example 25. (Decomposing the Global State Space in the Constraint System)
Looking back at Example 17 on Page 22, we see this constraint system.

1. $\forall t. r_1(\tau_{r_1 r_2}(t)) \wedge r_2(\tau_{r_1 r_2}(t)) \wedge r_1(\tau_{r_1 \bar{r}_2}(t)) \wedge \neg r_2(\tau_{r_1 \bar{r}_2}(t)) \wedge \neg r_1(\tau_{\bar{r}_1 r_2}(t)) \wedge r_2(\tau_{\bar{r}_1 r_2}(t)) \wedge \neg r_1(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \neg r_2(\tau_{\bar{r}_1 \bar{r}_2}(t))$
2. $\lambda_1^{\mathbb{B}}(0) \wedge \neg r_1(0) \wedge \neg r_2(0)$
3. $\forall t. \lambda_1^{\mathbb{B}}(t) \rightarrow \lambda_1^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \geq \lambda_1^{\#} \wedge \lambda_1^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 \bar{r}_2}(t)) \geq \lambda_1^{\#}$
   $\wedge\ \lambda_1^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{\bar{r}_1 r_2}(t)) \geq \lambda_1^{\#} \wedge \lambda_1^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 r_2}(t)) \geq \lambda_1^{\#}$
4. $\forall t. \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(t) \vee \neg g_2(t)$
5. $\forall t. \lambda_1^{\mathbb{B}}(t) \wedge r_1(t) \wedge \neg g_1(t) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) > \lambda_1^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \bar{r}_2}(t)) > \lambda_1^{\#}$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\bar{r}_1 r_2}(t)) > \lambda_1^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}$
6. $\forall t. \lambda_1^{\mathbb{B}}(t) \wedge r_2(t) \wedge \neg g_2(t) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) > \lambda_1^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \bar{r}_2}(t)) > \lambda_1^{\#}$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\bar{r}_1 r_2}(t)) > \lambda_1^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}$
7. $\forall t. \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(t) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) > \lambda_2^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \bar{r}_2}(t)) > \lambda_2^{\#}$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\bar{r}_1 r_2}(t)) > \lambda_2^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_2^{\#}$
8. $\forall t. \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(t) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) > \lambda_3^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \bar{r}_2}(t)) > \lambda_3^{\#}$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\bar{r}_1 r_2}(t)) > \lambda_3^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_3^{\#}$

We have to decompose all variables which are not global. In our example those are g_1 and g_2. In the constraint system above, they are marked in green and blue respectively. As one can see, we only have to change constraints 4 to 8.

4. $\forall t. \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(d_{p_1}(t)) \vee \neg g_2(d_{p_2}(t))$
5. $\forall t. \lambda_1^{\mathbb{B}}(t) \wedge r_1(t) \wedge \neg g_1(d_{p_1}(t)) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) > \lambda_1^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \bar{r}_2}(t)) > \lambda_1^{\#}$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\bar{r}_1 r_2}(t)) > \lambda_1^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}$
6. $\forall t. \lambda_1^{\mathbb{B}}(t) \wedge r_2(t) \wedge \neg g_2(d_{p_2}(t)) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) > \lambda_1^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \bar{r}_2}(t)) > \lambda_1^{\#}$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\bar{r}_1 r_2}(t)) > \lambda_1^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}$
7. $\forall t. \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(d_{p_1}(t)) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) > \lambda_2^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \bar{r}_2}(t)) > \lambda_2^{\#}$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\bar{r}_1 r_2}(t)) > \lambda_2^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_2^{\#}$
8. $\forall t. \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(d_{p_2}(t)) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\bar{r}_1 \bar{r}_2}(t)) > \lambda_3^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 \bar{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \bar{r}_2}(t)) > \lambda_3^{\#}$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\bar{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\bar{r}_1 r_2}(t)) > \lambda_3^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_3^{\#}$

4.4.2 Consistency and Weak Symmetry

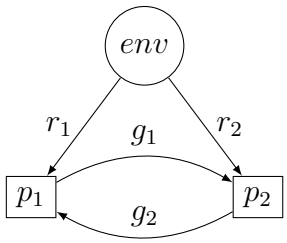
Consistency We not only have to make sure that a process can only access certain variables, we also have to make sure that a process acts consistently on any two histories it cannot distinguish. The update of state t in the combined transition system may thus only depend on the state of the process transition system and the input visible to the process.

We now have to add two sets of constraints.

1. $\forall t.d_p(\tau_\upsilon(t)) = d_p(\tau_{\upsilon'}(t))$ for all decisions $\upsilon, \upsilon' \subseteq O_{env}$ of the environment that are indistinguishable for p (i.e., $\upsilon \cap I_p = \upsilon' \cap I_p$) [10].

What this says is, that if we have two transitions and if the part the process can see goes into the same direction for both transitions, then it will not distinguish between the other parts that it should not be able to see.

Example 26. (Consistency - Part I)



Looking at this architecture, we have to make sure that p_1 cannot distinguish between the environment outputs r_2 and $\overline{r_2}$ and vice versa for p_2 and the environment outputs r_1 and $\overline{r_1}$.

9. $\forall t.d_1(\tau_{r_1 r_2}(t)) = d_1(\tau_{r_1 \bar{r}_2}(t)) \wedge d_1(\tau_{\bar{r}_1 r_2}(t)) = d_1(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge d_2(\tau_{r_1 r_2}(t)) = d_2(\tau_{\bar{r}_1 r_2}(t)) \wedge d_2(\tau_{r_1 \bar{r}_2}(t)) = d_2(\tau_{\bar{r}_1 \bar{r}_2}(t))$

Now we can take a look at the second constraint.

2. $\forall t, u.d_p(t) = d_p(u) \wedge \bigwedge_{a \in I_p \setminus O_{env}} (a(d_{p_a}(t)) \leftrightarrow a(d_{p_a}(u))) \rightarrow d_p(\tau_\upsilon(t) = d_p(\tau_\upsilon(u))$
   for all decisions $\upsilon \subseteq O_{env} \cup I_p$ (picking one representative for each class of environment decisions that p cannot distinguish). $p_a \in P^-$ denotes the process controlling the output variable $a \in O_{p_a}$ [10].

We also have to consider the case where we have two states of the combined transition system, but we are essentially working on the same system process transition system states and we get the same inputs that are no environment outputs. Then the successor states of the system process transition system states have to be the same.

Example 27. (Consistency - Part II)
The architecture we are looking at is the same one as in the previous example.

10. $\forall t, u. d_{p_1}(t) = d_{p_1}(u) \wedge (g_2(d_{p_2}(t)) \leftrightarrow g_2(d_{p_2}(u))) \rightarrow d_{p_1}(\tau_{r_1 r_2}(t)) = d_{p_1}(\tau_{r_1 r_2}(u)) \wedge d_{p_1}(\tau_{\bar{r}_1 r_2}(t)) = d_{p_1}(\tau_{\bar{r}_1 r_2}(u))$
11. $\forall t, u. d_{p_2}(t) = d_{p_2}(u) \wedge (g_1(d_{p_1}(t)) \leftrightarrow g_1(d_{p_1}(u))) \rightarrow d_{p_2}(\tau_{r_1 r_2}(t)) = d_{p_2}(\tau_{r_1 r_2}(u)) \wedge d_{p_2}(\tau_{r_1 \bar{r}_2}(t)) = d_{p_2}(\tau_{r_1 \bar{r}_2}(u))$

Let us take a look at constraint 10. We look at 2 states t and u. Their process p_1 parts are equal. If we now get g_2 as input, than the successor states of t and u have to be the same for all input directions.

Weak Symmetry Weak symmetry constraints can replace the consistency constraints. Compared to consistency, weak symmetry has one advantage: while consistency only looks at the transition function for the composition of the transition systems, weak symmetry also includes the system process transition functions. That means that when using weak symmetry, one can also derive the transition systems for the system processes.

$$\forall t. d_p(\tau_\upsilon(t)) = \tau_p(d_p(t); \mathsf{input}_p(\overrightarrow{\sigma}(t), \upsilon))$$

Example 28. (Weak Symmetry)
We are still looking at the same architecture from Example 26. The constraints we added in Example 26 can now be replaced by the following constraints.

9. (a) $\forall t.\ d_{p_1}(\tau_{r_1 r_2}(t)) = d_{p_1}(\tau_{r_1 \bar{r}_2}(t)) = \tau_{p_1}(d_{p_1}(t); \top, g_2(d_{p_2}(t)))$
$\wedge\ d_{p_1}(\tau_{\bar{r}_1 r_2}(t)) = d_{p_1}(\tau_{\bar{r}_1 \bar{r}_2}(t)) = \tau_{p_1}(d_{p_1}(t); \bot, g_2(d_{p_2}(t)))$

   (b) $\forall t.\ d_{p_2}(\tau_{r_1 r_2}(t)) = d_{p_2}(\tau_{\bar{r}_1 r_2}(t)) = \tau_{p_2}(d_{p_2}(t); \top, g_1(d_{p_1}(t)))$
$\wedge\ d_{p_2}(\tau_{r_1 \bar{r}_2}(t)) = d_{p_2}(\tau_{\bar{r}_1 \bar{r}_2}(t)) = \tau_{p_2}(d_{p_2}(t); \bot, g_1(d_{p_1}(t)))$

If we take a look at constraint 9 (a), we see that it does not matter if r_2 is granted or not. The output only depends on whether r_1 and g_2 were granted or not.

4.5 The Interpretation

As mentioned in the last chapter, SMT solvers like Z3, CVC4 or Yices can solve this constraint system when rewritten into an appropriate format. See Page 37 and following for an introduction to these SMT solvers.

4.6 The Result

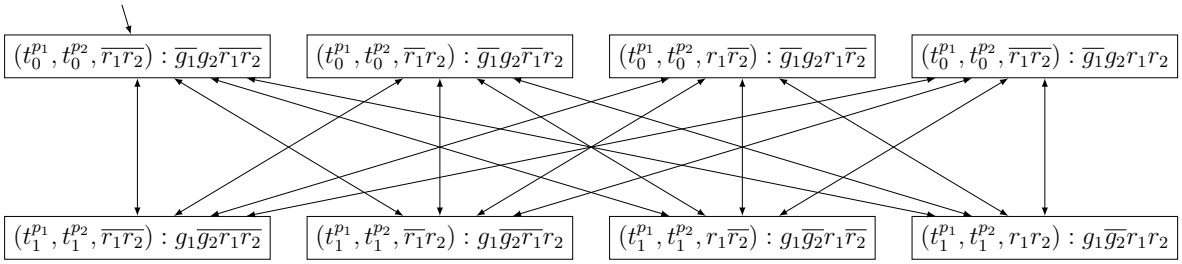
This constraint system approach will give as results a transition system for each system process and a combined transition system which is the composition of the system process transition systems, if, and only if, the specification is realizable for the given architecture.

Example 29. (The Result)
For our synthesis problem from example 20 we get back a family of two transition systems:

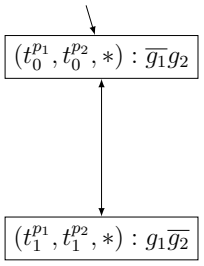
$$\mathcal{T}_{p_1} = (T_{p_1}, t_o^{p_1}, \tau_{p_1}, o_{p_1}) \qquad \mathcal{T}_{p_2} = (T_{p_2}, t_o^{p_2}, \tau_{p_2}, o_{p_2})$$

The composition of $\mathcal{T}_{p_1}$ and $\mathcal{T}_{p_2}$ looks as follows:

$$\mathcal{T}_A = (T, t_0, \tau, o)$$



If we do not use input preservation, the transition system is slightly smaller:



CHAPTER 5

INPUT FORMAT FOR DISTRIBUTED ARCHITECTURES

Looking back, we have so far been over background information needed to understand bounded synthesis and we saw bounded synthesis for monolithic systems and for distributed architectures. However thus far, we only know about the theoretical approach. To implement bounded synthesis for distributed architectures, we need an input format for the architecture.

5.1 Architectures

In Chapter 3 we introduced architectures on Page 24. Because we will be talking about architectures for the remainder of this chapter, we look at the definition and the example again:

Definition 14. (Architecture)
An architecture is a tuple $A = (P, env, V, I, O)$ where

- P is a set of system processes and the designated environment process, $P^- = P$ env is the set of system processes
- env is the designated environment process
- V is the set of boolean system variables with $V = \bigcup_{p \in P} O_P$
- I assigns a set of input variables to each system process, $I = \{I_P \subseteq V | p \in P^-\}$
- O assigns a set of output variables to each process, $O = \{O_P \subseteq V | p \in P\}$

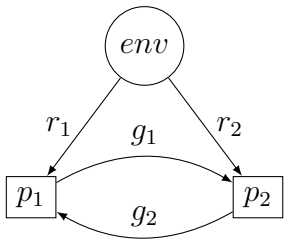
The same variable may occur in multiple sets in I to indicate broadcasting. All sets in O are however disjoint.

The following example is a distributed architecture.

```
1 [PROCESSES] processes
2 [INPUT] [process] inputs
3 [OUTPUT] [process] outputs
4 [BOUND] [process] bounds
```

Figure 5.1: Modeling the Architecture as an Input Format

Example 30. (Distributed Architecture)



$$A = (P, env, V, I, O)$$

$$P = \{env, p_1, p_2\} \qquad P^- = \{p_1, p_2\}$$

$$V = \{r_1, r_2, g_1, g_2\}$$

$$I = \{r_1, r_2, g_1, g_2\}$$

$$I_{p_1} = \{r_1, g_2\} \qquad I_{p_2} = \{r_2, g_1\}$$

$$O = \{r_1, r_2, g_1, g_2\} \qquad O_{env} = \{r_1, r_2\}$$

$$O_{p_1} = \{g_1\} \qquad O_{p_2} = \{g_2\}$$

The architecture is not fully known because neither p_1 nor p_2 get all outputs of the environment O_{env} as inputs: $O_{env} \neq I_{p_1} \neq I_{p_2}$

Using the definition of architectures, we can represent any architecture that comes to mind. However, if we want to use such an architecture as input for a program, it remains unclear which input format to use. The following subsection proposes an input format for architectures.

5.2 The Format

In the last section we recalled architectures. We saw the definition and an example. We now want to derive an input format for architectures, that is easily understandable and writable.

We have to model the processes, the inputs and outputs. Since the set of boolean system variables is the union of all process outputs, we do not explicitly have to model them. Moreover, we want to model the bounds for the size of the system process transition systems. Bounding the size of the composition of the transition systems can be done in the tool we are are using the input format with, which we will see in Chapter 6.

```
1 [P] processes
2 [I] [process] inputs
3 [O] [process] outputs
4 [B] [process] bounds
```

Figure 5.2: Modeling the Architecture as an Input Format with Abbreviations

Instead of formally defining the input format, we give the specification in form of actual code in Figure 5.1 and Figure 5.2. We explain the meaning of each part in the following. Figure 5.1 shows the input format and Figure 5.2 shows the input format using abbreviations. The order of the lines does not matter.

[PROCESSES] or [P] With the keywords 'PROCESSES' or 'P' in square brackets, one can model all processes of the architecture, except the environment process, which will always be named 'env' and does not have to be mentioned in the list. The processes can be named anything, however one should not use the keyword 'env' to describe a process other than the environment process.

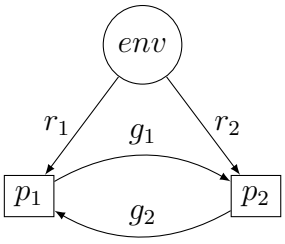
[INPUTS] or [I] The keywords 'INPUT' or 'I' in square brackets can describe the inputs for one process. The process is named in square brackets with a space in between after the keyword and then all inputs for that one process are mentioned. All inputs are separated by one space. The environment process has no inputs.

[OUTPUTS] or [O] The keywords 'OUTPUT' or 'O' in square brackets can be used the same way 'INPUT' is used for inputs. The process is again named in square brackets with a space in between after the keyword and then all outputs for that one process are mentioned. Again, all outputs are separated by one space. All outputs of the environments need to be modeled, even though the environment process was not mentioned in the list of processes.

[BOUNDS] or [B] The keywords 'BOUND' or 'B' in square brackets are used the same way as 'INPUT' or 'OUTPUT'. The process is named in square brackets with a space in between after the keyword and then a bound is given. The bound can be any number greater than 0.

The following example models the architecture seen in Example 30 earlier in this chapter.

Example 31. (The Format)




```
[PROCESSES] p1 p2
[OUTPUT] [env] r1 r2
[INPUT] [p1] r1 g2
[OUTPUT] [p1] g1
[BOUND] [p1] 4
[INPUT] [p2] r2 g1
[OUTPUT] [p2] g2
[BOUND] [p2] 4
```

The system process are p_1 and p_1. The outputs of the environment are r_1 and r_2 and the inputs of system process p_1 are r_1 and g_2 and so on. Both system processes p_1 and p_2 have a bound of 4.

The input format is easy to understand and easy to use.

CHAPTER 6

IMPLEMENTATION

This chapter is about the implementation of bounded synthesis for distributed architectures, which was realized in the tool PartyPlus. PartyPlus is an extension to the tool Party [11]. Unlike Party it however does not only use Z3 as a SMT Solver, but also CVC4 and Yices. Party and PartyPlus along with all three solvers will be introduced and explained in more detail below.

6.1 Party

Party is a tool that was first introduced in the paper "PARTY: Parameterized Synthesis of Token Rings" by Ayrat Khalimov, Swen Jabobs, and Roderick Bloem from the Graz University of Technology in Austria [11]. Luckily, Party cannot only do parameterized synthesis, but also bounded synthesis for monolithic systems. An overview of the control flow can be found in Figure 6.1.

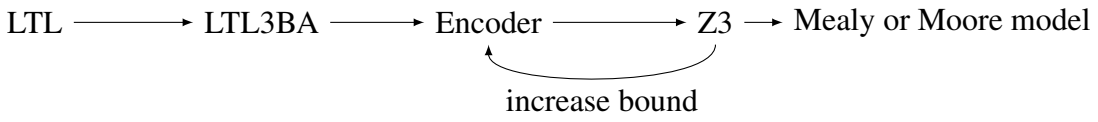


Figure 6.1: Overview of the Control Flow of Party

Party gets as input an LTL specification in a format that is very similar to that of Acacia+ [7]. It then uses LTL3BA [1], a tool that can build Büchi automata when given a specification, to build the universal co-Büchi automaton from which it derives the constraint system with the encoder. The interesting part here is that the number of states used is increased in each round until either a model is found or the tool times out. The model can be outputted as a transition system in either the moore or mealy format.

Party does not get a fixed initial input as part of the input, because that is only needed if we look at input preserving transition systems. Party, however, outputs transition systems which are not input preserving, because they are in general smaller than input preserving transition systems and thus easier to compute.

6.2 Solvers

PartyPlus has the option to use one of three SMT solvers, CVC4, Yices and Z3, to solve the constraint system. In our case, the job of an SMT solver is to take our constraint system and

interpret all uninterpreted functions. This means that the SMT solver is supposed to find an interpretation or model for us.

6.2.1 CVC4

CVC4 is a collaboration project between the New York University and the University of Iowa. The main contributers are Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, and Tim King from the New York University and Andrew Reynolds, and Cesare Tinelli from the University of Iowa. [4] CVC4 supports the SMT-Lib 2.0 [5] and extended it with an option to use data types which we use. More details on the SMT instances we derive and use can be found starting on Page 40.

6.2.2 Yices

Yices was first introduced back in 2006 by Bruno Dutertre and Leonardo de Moura at SRI International. [9] Since then it has been continuously updated. Even though Yices also supports the SMT-Lib 2.0, it did not extend it with data types. However, Yices has its own input format *Yices* that supports data types but is slightly different from the SMT-Lib 2.0 when it comes to defining functions, data types and using universal quantifiers.

6.2.3 Z3

Z3 is a SMT solver developed at Microsoft Research by Leonardo de Moura and Nikolaj Bjørner. [8] It supports the SMT-Lib 2.0 and data types and luckily it accepts the same instances that CVC4 does.

6.3 PartyPlus

PartyPlus is based on Party. The extension makes it possible to solve bounded synthesis for distributed architectures. An overview of the control flow can be found in Figure 6.3.

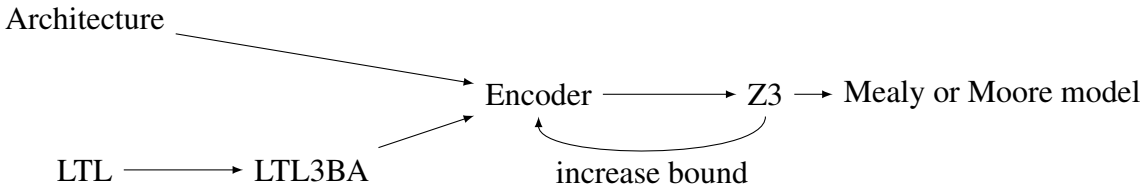


Figure 6.2: Overview of the Control Flow of PartyPlus

Unlike Party, PartyPlus does not only get a specification as input, but also an architecture. LTL3BA builds once again an automaton from the specification. We again start building the constraint system based on the automaton. However, now the architecture is used to make certain alterations to the constraint system that the encoder builds. The theoretical background was introduced in Chapters 3 and 4. The encoder first builds a constraint system for a 1 state transition systems. If the constraint system is not satisfiable, meaning we cannot find a transition system of size 1, the encoder builds a constraint system for a transition system of the size 2. This process is done iteratively until we either find a solution, or the tool times out.

6.3.1 The Input

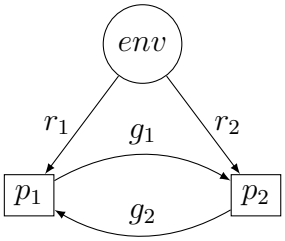
PartyPlus gets two files as input. One has the file extension '.arc' and one has the file extension '.ltl'.

.arc The file with the extension '.arc' includes the architecture in the input format that was introduced in the last chapter on Page 33.

Example 32. (.arc)

```
[PROCESSES] p1 p2
[OUTPUT] [env] r1 r2
[INPUT] [p1] r1 g2
[OUTPUT] [p1] g1
[BOUND] [p1] 4
[INPUT] [p2] r2 g1
[OUTPUT] [p2] g2
[BOUND] [p2] 4
```

This is an example of what could be written in the architecture file. The code represents the following architecture.



We have seen the same example already in the last chapter on Page 31.

.ltl The file with the extension '.ltl' includes the specification. The input format is the same that Party also uses, which is similar to that of Acacia+ [7].

Example 33. (.ltl)
Let us take a look at a specification we have seen numerous times already, the request response specification.

$$\varphi = \square(r_1 \rightarrow \diamond g_1) \wedge \square(r_2 \rightarrow \diamond g_2) \wedge \square(\neg g_1 \vee \neg g_2)$$

The representation would look as follows in the LTL file.

```
G((r1=1) -> (F(g1=1))) * G((r2=1) -> (F(g2=1))) * G(!((g1=1) * (g2=1)));
```

Always remember the semicolon at the end of the line.

6.3.2 The Options

PartyPlus has several command line options to choose from. Not only can one pick one of the 3 different solvers, but also whether a moore or a mealy model is outputted. The user can also choose between consistency and weak symmetry constraints during computation. The difference is explained in Chapter 4 on Page 30. It is possible to keep the instances created for the solvers. An upper bound can be given on the size of the transition system or it is possible to let the tool check for a process implementation of a certain size. If a transition system is found, one can choose to save it in a dot format file.

Figure 6.3 shows all mutually exclusive options.

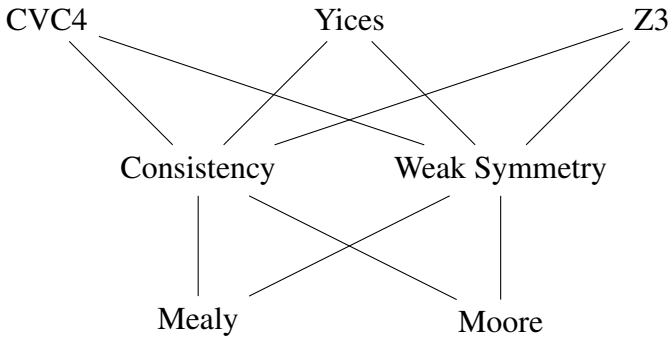


Figure 6.3: Configurations of PartyPlus

6.3.3 Control Flow

Most of the control flow is based on Party [11]. As soon as the tool has the input files and all picked options, the specification is rewritten to a format LTL3BA can read. Because LTL3BA does not deal with the release operator $\mathcal{R}$, it is replaced by an equivalent structure if present in a formula. LTL3BA builds us the universal co-Büchi automaton that we need to continue.

We use an encoder to built the constraint system. This is done by extending the constraint system iteratively until the solver either times out or finds a model.

CVC4 and Z3 both accept an extension of the SMT-Lib 2.0 that allows data types. Even though Yices accepts the SMT-Lib 2.0, it does not accept the data types extension. Yices however has its own input language called *Yices* which accepts the use of data types. Explanations by example can be found below for both an instance for CVC4 and Z3, and Yices.

Example 34. (SMT Instance for CVC4 and Z3)
We want to create a SMT instance of Examples 32 and 33 for CVC4 or Z3 with weak symmetry constraints and we want a moore transition system as output. The constraint system for this looks as follows.

1. $\forall t. r_1(\tau_{r_1 r_2}(t)) \wedge r_2(\tau_{r_1 r_2}(t)) \wedge r_1(\tau_{r_1 \bar{r}_2}(t)) \wedge \neg r_2(\tau_{r_1 \bar{r}_2}(t)) \wedge \neg r_1(\tau_{\bar{r}_1 r_2}(t)) \wedge r_2(\tau_{\bar{r}_1 r_2}(t)) \wedge \neg r_1(\tau_{\bar{r}_1 \bar{r}_2}(t)) \wedge \neg r_2(\tau_{\bar{r}_1 \bar{r}_2}(t))$

2. $\lambda_1^{\mathbb{B}}(0) \wedge \neg r_1(0) \wedge \neg r_2(0)$
3. $\forall t. \lambda_1^{\mathbb{B}}(t) \rightarrow \lambda_1^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) \geq \lambda_1^{\#} \wedge \lambda_1^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1\overline{r}_2}(t)) \geq \lambda_1^{\#}$
   $\wedge\ \lambda_1^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{\overline{r}_1 r_2}(t)) \geq \lambda_1^{\#} \wedge \lambda_1^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 r_2}(t)) \geq \lambda_1^{\#}$
4. $\forall t. \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(d_{p_1}(t)) \vee \neg g_2(d_{p_2}(t))$
5. $\forall t. \lambda_1^{\mathbb{B}}(t) \wedge r_1(t) \wedge \neg g_1(d_{p_1}(t)) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_1^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_1^{\#}$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}$
6. $\forall t. \lambda_1^{\mathbb{B}}(t) \wedge r_2(t) \wedge \neg g_2(d_{p_2}(t)) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_1^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_1^{\#}$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}$
7. $\forall t. \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(d_{p_1}(t)) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_2^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_2^{\#}$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_2^{\#} \wedge \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_2^{\#}$
8. $\forall t. \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(d_{p_2}(t)) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_3^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_3^{\#}$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_3^{\#} \wedge \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_3^{\#}$
9. (a) $\forall t.\ d_{p_1}(\tau_{r_1 r_2}(t)) = d_{p_1}(\tau_{r_1\overline{r}_2}(t)) = \tau_{p_1}(d_{p_1}(t); \top, g_2(d_{p_2}(t)))$
   $\wedge\ d_{p_1}(\tau_{\overline{r}_1 r_2}(t)) = d_{p_1}(\tau_{\overline{r}_1\overline{r}_2}(t)) = \tau_{p_1}(d_{p_1}(t); \bot, g_2(d_{p_2}(t)))$

   (b) $\forall t.\ d_{p_2}(\tau_{r_1 r_2}(t)) = d_{p_2}(\tau_{\overline{r}_1 r_2}(t)) = \tau_{p_2}(d_{p_2}(t); \top, g_1(d_{p_1}(t)))$
   $\wedge\ d_{p_2}(\tau_{r_1\overline{r}_2}(t)) = d_{p_2}(\tau_{\overline{r}_1\overline{r}_2}(t)) = \tau_{p_2}(d_{p_2}(t); \bot, g_1(d_{p_1}(t)))$

The timeout and the option to show statistics can be set in the commandline call for both CVC4 and Z3.

The following is the equivalent SMT instance. The states of the transition system are enrolled and we frequently summarize over the values of r_1 and r_2. Further explanation can be found below.

```
(declare-datatypes () ((T (t0) (t1) (t2) (t3) (t4) (t5) (t6) (t7) (t8) (t9) (t10) (t11) (t12) (t13) (t14) (t15) (t16) (
    t17) (t18) (t19) (t20) (t21) (t22) (t23) (t24) (t25) (t26) (t27) (t28) (t29) (t30) (t31) (t32) (t33) (t34) (t35)
    (t36) (t37) (t38) (t39) (t40) (t41) (t42) (t43) (t44) (t45) (t46) (t47) (t48) (t49) (t50) (t51) (t52) (t53) (t54)
     (t55) (t56) (t57) (t58) (t59) (t60) (t61) (t62) (t63) (t64) (t65) (t66) (t67) (t68) (t69) (t70) (t71) (t72) (t73
    ) (t74) (t75) (t76) (t77) (t78) (t79) (t80) (t81) (t82) (t83) (t84) (t85) (t86) (t87) (t88) (t89) (t90) (t91) (
    t92) (t93) (t94) (t95) (t96) (t97) (t98) (t99) (t100) (t101) (t102) (t103) (t104) (t105) (t106) (t107) (t108) (
    t109) (t110) (t111) (t112) (t113) (t114) (t115) (t116) (t117) (t118) (t119) (t120) (t121) (t122) (t123) (t124) (
    t125) (t126) (t127))))
(declare-datatypes () ((A2 (a20) (a21) (a22) (a23) (a24) (a25) (a26) (a27) (a28) (a29) (a210) (a211))))
(declare-datatypes () ((A1 (a10) (a11) (a12) (a13) (a14) (a15) (a16) (a17) (a18) (a19) (a110) (a111))))

(declare-fun tau (Bool Bool T) T)
(declare-fun tau2 (Bool Bool A2) A2)
(declare-fun tau1 (Bool Bool A1) A1)
(declare-fun g2 (A2) Bool)
(declare-fun g1 (A1) Bool)
(declare-fun d2 (T) A2)
(declare-fun d1 (T) A1)

(declare-datatypes () ((Q (q_accept_all) (q_accept_S2) (q_accept_S3) (q_T0_init))))
(declare-fun laB (Q T) Bool)
(declare-fun laC (Q T) Int)

(assert (laB q_T0_init t0))
(assert (forall ((?r2_0 Bool) (?r1_0 Bool)) (=> (and (laB q_accept_all t0)  )   false  )))
(assert (forall ((?r2_0 Bool) (?r1_0 Bool)) (=> (and (laB q_accept_S2 t0)  (not (g1 (d1 t0)))  )  (and (laB q_accept_S2
     (tau ?r1_0 ?r2_0 t0)) (> (laC q_accept_S2 (tau ?r1_0 ?r2_0 t0)) (laC q_accept_S2 t0))) )))
(assert (forall ((?r2_0 Bool) (?r1_0 Bool)) (=> (and (laB q_accept_S3 t0)  (not (g2 (d2 t0)))  )  (and (laB q_accept_S3
     (tau ?r1_0 ?r2_0 t0)) (> (laC q_accept_S3 (tau ?r1_0 ?r2_0 t0)) (laC q_accept_S3 t0))) )))
(assert (forall ((?r2_0 Bool) (?r1_0 Bool)) (=> (and (laB q_T0_init t0) (and (g1 (d1 t0)) (not (g2 (d2 t0)))) )  (laB
    q_T0_init (tau ?r1_0 ?r2_0 t0)) )))
(assert (forall ((?r2_0 Bool)) (=> (and (laB q_T0_init t0)  (not (g1 (d1 t0)))  )  (laB q_accept_S2 (tau true ?r2_0 t0)
    ) )))
(assert (forall ((?r2_0 Bool) (?r1_0 Bool)) (=> (and (laB q_T0_init t0)  (not (g1 (d1 t0)))  )  (laB q_T0_init (tau ?
    r1_0 ?r2_0 t0)) )))
(assert (forall ((?r2_0 Bool) (?r1_0 Bool)) (=> (and (laB q_T0_init t0) (and (g1 (d1 t0)) (g2 (d2 t0))) )   false  )))
(assert (forall ((?r1_0 Bool)) (=> (and (laB q_T0_init t0)  (not (g2 (d2 t0)))  )  (laB q_accept_S3 (tau ?r1_0 true t0)
    ) )))
(assert (and (= (d2 (tau true true t0)) (tau2 (g1 (d1 t0)) true (d2 t0))) (= (d2 (tau true false t0)) (tau2 (g1 (d1 t0)
    ) false (d2 t0))) (= (d2 (tau false true t0)) (tau2 (g1 (d1 t0)) true (d2 t0))) (= (d2 (tau false false t0)) (
    tau2 (g1 (d1 t0)) false (d2 t0)))))
(assert (and (= (d1 (tau true true t0)) (tau1 (g2 (d2 t0)) true (d1 t0))) (= (d1 (tau true false t0)) (tau1 (g2 (d2 t0)
    ) true (d1 t0))) (= (d1 (tau false true t0)) (tau1 (g2 (d2 t0)) false (d1 t0))) (= (d1 (tau false false t0)) (
    tau1 (g2 (d2 t0)) false (d1 t0)))))
; encoded state t0

(assert (laB q_T0_init t0))
(assert (forall ((?r2_0 Bool) (?r1_0 Bool)) (=> (and (laB q_accept_all t1)  )   false  )))
```

```
32 (assert (forall ((?r2_0 Bool) (?r1_0 Bool)) (=> (and (laB q_accept_S2 t1)  (not (g1 (d1 t1)))  )  (and (laB q_accept_S2
       (tau ?r1_0 ?r2_0 t1)) (> (laC q_accept_S2 (tau ?r1_0 ?r2_0 t1)) (laC q_accept_S2 t1))) )))
33 (assert (forall ((?r2_0 Bool) (?r1_0 Bool)) (=> (and (laB q_accept_S3 t1)  (not (g2 (d2 t1)))  )  (and (laB q_accept_S3
       (tau ?r1_0 ?r2_0 t1)) (> (laC q_accept_S3 (tau ?r1_0 ?r2_0 t1)) (laC q_accept_S3 t1))) )))
34 (assert (forall ((?r2_0 Bool) (?r1_0 Bool)) (=> (and (laB q_T0_init t1) (and (g1 (d1 t1)) (not (g2 (d2 t1)))) )  (laB
      q_T0_init (tau ?r1_0 ?r2_0 t1)) )))
35 (assert (forall ((?r2_0 Bool)) (=> (and (laB q_T0_init t1)  (not (g1 (d1 t1)))  )  (laB q_accept_S2 (tau true ?r2_0 t1)
      ) )))
36 (assert (forall ((?r2_0 Bool) (?r1_0 Bool)) (=> (and (laB q_T0_init t1)  (not (g1 (d1 t1)))  )  (laB q_T0_init (tau ?
      r1_0 ?r2_0 t1)) )))
37 (assert (forall ((?r2_0 Bool) (?r1_0 Bool)) (=> (and (laB q_T0_init t1) (and (g1 (d1 t1)) (g2 (d2 t1))) )   false  )))
38 (assert (forall ((?r1_0 Bool)) (=> (and (laB q_T0_init t1)  (not (g2 (d2 t1)))  )  (laB q_accept_S3 (tau ?r1_0 true t1)
      ) )))
39 (assert (and (= (d2 (tau true true t1)) (tau2 (g1 (d1 t1)) true (d2 t1))) (= (d2 (tau true false t1)) (tau2 (g1 (d1 t1)
      ) false (d2 t1))) (= (d2 (tau false true t1)) (tau2 (g1 (d1 t1)) true (d2 t1))) (= (d2 (tau false false t1)) (
      tau2 (g1 (d1 t1)) false (d2 t1)))))
40 (assert (and (= (d1 (tau true true t1)) (tau1 (g2 (d2 t1)) true (d1 t1))) (= (d1 (tau true false t1)) (tau1 (g2 (d2 t1)
      ) true (d1 t1))) (= (d1 (tau false true t1)) (tau1 (g2 (d2 t1)) false (d1 t1))) (= (d1 (tau false false t1)) (
      tau1 (g2 (d2 t1)) false (d1 t1)))))
41 ; encoded state t1
42
43 (assert (forall ((?r1_0 Bool) (?r2_0 Bool)) (or (= (tau ?r1_0 ?r2_0 t0) t0) (= (tau ?r1_0 ?r2_0 t0) t1))))
44 (assert (forall ((?r1_0 Bool) (?r2_0 Bool)) (or (= (tau ?r1_0 ?r2_0 t1) t0) (= (tau ?r1_0 ?r2_0 t1) t1))))
45
46 (check-sat)
47
48 (get-value ((tau2 true true (d2 t0))))
49 (get-value ((tau2 true true (d2 t1))))
50 (get-value ((tau2 true false (d2 t0))))
51 (get-value ((tau2 true false (d2 t1))))
52 (get-value ((tau2 false true (d2 t0))))
53 (get-value ((tau2 false true (d2 t1))))
54 (get-value ((tau2 false false (d2 t0))))
55 (get-value ((tau2 false false (d2 t1))))
56 (get-value ((tau1 true true (d1 t0))))
57 (get-value ((tau1 true true (d1 t1))))
58 (get-value ((tau1 true false (d1 t0))))
59 (get-value ((tau1 true false (d1 t1))))
60 (get-value ((tau1 false true (d1 t0))))
61 (get-value ((tau1 false true (d1 t1))))
62 (get-value ((tau1 false false (d1 t0))))
63 (get-value ((tau1 false false (d1 t1))))
64 (get-value ((tau true true t0)))
65 (get-value ((tau true true t1)))
66 (get-value ((tau true false t0)))
67 (get-value ((tau true false t1)))
68 (get-value ((tau false true t0)))
69 (get-value ((tau false true t1)))
70 (get-value ((tau false false t0)))
71 (get-value ((tau false false t1)))
72 (get-value ((g1 (d1 t0))))
73 (get-value ((g1 (d1 t1))))
74 (get-value ((g2 (d2 t0))))
75 (get-value ((g2 (d2 t1))))
76 (get-value ((d1 t0)))
77 (get-value ((d1 t1)))
78 (get-value ((d2 t0)))
79 (get-value ((d2 t1)))
80 (exit)
```

Lines 1 - 3

We are choosing to encode the states of the transition system as elements of a new datatype `T`. This datatype has a size of 128 states. The size can however be bound with the options bound or size. While using the option bound tests all constraint systems up to this bound, the option size only tests the constraint system that looks for a model with exactly that many transition system states.

Since we have two system processes p_1 and p_2 in our architecture, and both have their own transition systems, their states are encoded in lines 2 and 3 as the new datatypes `A1` and `A2` respectively. The bound given in the implementation for the size of system process transition systems is 12. This number can however be changed by using the bound option for the system processes in the architecture input format.

Lines 5 - 7

Here the transition functions for our three transition systems are encoded. `tau` is

the transition function for the composition of the system process transition systems and `tau1` and `tau2` are the transition functions for the system process transition systems.

Lines 8 - 9

`g1` and `g2` are the outputs of the system processes, represented as uninterpreted functions. Since we want a moore model as output, they do not get any directions as input, only a state of the transition systems. However, since only process `p1` can grant `g1`, `g1` only gets access to a state of the system process transition system of `p1`. The same applies for process `p2` an grant `g2`.

Lines 10 - 11

`d1` and `d2` are the help functions that make sure that `g1` and `g2` can only access the parts of a in `T`, they are allowed to work with.

Line 13

The states of the universal co-Büchi automaton are also represented in a datatype.

Lines 14 - 15

The unary predicate symbol $\lambda_q^{\mathbb{B}}$ is introduced as `laB` and the unary function symbol $\lambda_q^{\#}$ as `laC`.

Lines 17 - 28

The first state of the transition system, `t0`, is encoded. Line 17 defines the initial state or constraint 2. Lines 18 - 25 encode transitions of the automaton. Let us take a look at Line 22. It roughly translates to

$$\forall r_2.\lambda_0^{\mathbb{B}}(t_0) \wedge \neg g_1(d_{p_1}(t_0)) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{true\ ?r_2}(t_0))$$

We cannot find any such a transition in the constraint system above. First, let us enroll the universal quantor over r_2.

$$\lambda_0^{\mathbb{B}}(t_0) \wedge \neg g_1(d_{p_1}(t_0)) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{true\ true}(t_0)) \wedge \lambda_0^{\mathbb{B}}(t_0) \wedge \neg g_1(d_{p_1}(t_0)) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{true\ false}(t_0))$$

We can start to see a certain resemblance with constraint 5. $\lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}$ cannot be found here because the approach used differentiates between cycles and other transitions. We do not look at a cycle transition so we do not care about this part of the constraint.

Lines 26 and 27 represent the weak symmetry constraints 9 (a) and 9 (b).

Line 28 is a simple comment.

Lines 30 - 41

State `t1` is defined in the same fashion as `t0` above.

Lines 43 - 44

In Line 1 we introduced the data type for the compositional transition system. We

want to make sure that only those states of the transition system are used in the model, that were actually constrained in the instance.

Line 46

The solver is told to do its job and solve the constraint system.

Lines 48 - 79

Since we want to get a model as output, we do have to know what that model should look like. If the constraint system is satisfiable, lines 48 - 79 tell us exactly, what a model could look like. Here is an example model.

```
1  sat
2  (((tau2 true true (d2 t0)) a20))
3  (((tau2 true true (d2 t1)) a20))
4  (((tau2 true false (d2 t0)) a20))
5  (((tau2 true false (d2 t1)) a20))
6  (((tau2 false true (d2 t0)) a20))
7  (((tau2 false true (d2 t1)) a21))
8  (((tau2 false false (d2 t0)) a20))
9  (((tau2 false false (d2 t1)) a21))
10 (((tau1 true true (d1 t0)) a10))
11 (((tau1 true true (d1 t1)) a11))
12 (((tau1 true false (d1 t0)) a10))
13 (((tau1 true false (d1 t1)) a11))
14 (((tau1 false true (d1 t0)) a10))
15 (((tau1 false true (d1 t1)) a10))
16 (((tau1 false false (d1 t0)) a10))
17 (((tau1 false false (d1 t1)) a10))
18 (((tau true true t0) t1))
19 (((tau true true t1) t0))
20 (((tau true false t0) t1))
21 (((tau true false t1) t0))
22 (((tau false true t0) t1))
23 (((tau false true t1) t0))
24 (((tau false false t0) t1))
25 (((tau false false t1) t0))
26 (((g1 (d1 t0)) true))
27 (((g1 (d1 t1)) false))
28 (((g2 (d2 t0)) false))
29 (((g2 (d2 t1)) true))
30 (((d1 t0) a11))
31 (((d1 t1) a10))
32 (((d2 t0) a21))
33 (((d2 t1) a20))
```

Line 80

This line tells the solver to exit.

Constraint 1 is not encoded because constraint 1 represents input preservation and we do not use input preservation in our implementation.

We have now seen an example instance for CVC4 and Z3. The same instance for Yices differs only slightly.

Example 35. (Yices Instance)

```
1  (set-timeout 1800)
2
3  (define-type T (scalar t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14 t15 t16 t17 t18 t19 t20 t21 t22 t23 t24 t25
       t26 t27 t28 t29 t30 t31 t32 t33 t34 t35 t36 t37 t38 t39 t40 t41 t42 t43 t44 t45 t46 t47 t48 t49 t50 t51 t52 t53
       t54 t55 t56 t57 t58 t59 t60 t61 t62 t63 t64 t65 t66 t67 t68 t69 t70 t71 t72 t73 t74 t75 t76 t77 t78 t79 t80 t81
       t82 t83 t84 t85 t86 t87 t88 t89 t90 t91 t92 t93 t94 t95 t96 t97 t98 t99 t100 t101 t102 t103 t104 t105 t106 t107
       t108 t109 t110 t111 t112 t113 t114 t115 t116 t117 t118 t119 t120 t121 t122 t123 t124 t125 t126 t127))
4  (define-type A1 (scalar a10 a11 a12 a13 a14 a15 a16 a17 a18 a19 a110 a111))
5  (define-type A2 (scalar a20 a21 a22 a23 a24 a25 a26 a27 a28 a29 a210 a211))
6
7  (define tau::(-> bool bool T T))
8  (define tau2::(-> bool bool A2 A2))
9  (define tau1::(-> bool bool A1 A1))
10 (define g2::(-> A2 bool))
11 (define g1::(-> A1 bool))
12 (define d2::(-> T A2))
```

```
(define d1::(-> T A1))

(define-type Q (scalar q_accept_all q_accept_S2 q_T0_init q_accept_S3))
(define laB::(-> Q T bool))
(define laC::(-> Q T int))

(assert (laB q_T0_init t0))
(assert (=> (and (laB q_accept_all t0) ) false ))
(assert (=> (and (laB q_accept_S2 t0) (not (g1 (d1 t0))) ) (and (laB q_accept_S2 (tau true true t0)) (> (laC
      q_accept_S2 (tau true true t0)) (laC q_accept_S2 t0))) ))
(assert (=> (and (laB q_accept_S2 t0) (not (g1 (d1 t0))) ) (and (laB q_accept_S2 (tau true false t0)) (> (laC
      q_accept_S2 (tau true false t0)) (laC q_accept_S2 t0))) ))
(assert (=> (and (laB q_accept_S2 t0) (not (g1 (d1 t0))) ) (and (laB q_accept_S2 (tau false true t0)) (> (laC
      q_accept_S2 (tau false true t0)) (laC q_accept_S2 t0))) ))
(assert (=> (and (laB q_accept_S2 t0) (not (g1 (d1 t0))) ) (and (laB q_accept_S2 (tau false false t0)) (> (laC
      q_accept_S2 (tau false false t0)) (laC q_accept_S2 t0))) ))
(assert (=> (and (laB q_T0_init t0) (and (not (g2 (d2 t0))) (g1 (d1 t0))) ) (laB q_T0_init (tau true true t0)) ))
(assert (=> (and (laB q_T0_init t0) (and (not (g2 (d2 t0))) (g1 (d1 t0))) ) (laB q_T0_init (tau true false t0)) ))
(assert (=> (and (laB q_T0_init t0) (and (not (g2 (d2 t0))) (g1 (d1 t0))) ) (laB q_T0_init (tau false true t0)) ))
(assert (=> (and (laB q_T0_init t0) (and (not (g2 (d2 t0))) (g1 (d1 t0))) ) (laB q_T0_init (tau false false t0)) ))
(assert (=> (and (laB q_T0_init t0) (and (g2 (d2 t0)) (g1 (d1 t0))) ) false ))
(assert (=> (and (laB q_T0_init t0) (not (g1 (d1 t0))) ) (laB q_T0_init (tau true true t0)) ))
(assert (=> (and (laB q_T0_init t0) (not (g1 (d1 t0))) ) (laB q_T0_init (tau true false t0)) ))
(assert (=> (and (laB q_T0_init t0) (not (g1 (d1 t0))) ) (laB q_T0_init (tau false true t0)) ))
(assert (=> (and (laB q_T0_init t0) (not (g1 (d1 t0))) ) (laB q_T0_init (tau false false t0)) ))
(assert (=> (and (laB q_T0_init t0) (not (g2 (d2 t0))) ) (laB q_accept_S3 (tau true true t0)) ))
(assert (=> (and (laB q_T0_init t0) (not (g2 (d2 t0))) ) (laB q_accept_S3 (tau false true t0)) ))
(assert (=> (and (laB q_T0_init t0) (not (g1 (d1 t0))) ) (laB q_accept_S2 (tau true true t0)) ))
(assert (=> (and (laB q_T0_init t0) (not (g1 (d1 t0))) ) (laB q_accept_S2 (tau true false t0)) ))
(assert (=> (and (laB q_accept_S3 t0) (not (g2 (d2 t0))) ) (and (laB q_accept_S3 (tau true true t0)) (> (laC
      q_accept_S3 (tau true true t0)) (laC q_accept_S3 t0))) ))
(assert (=> (and (laB q_accept_S3 t0) (not (g2 (d2 t0))) ) (and (laB q_accept_S3 (tau true false t0)) (> (laC
      q_accept_S3 (tau true false t0)) (laC q_accept_S3 t0))) ))
(assert (=> (and (laB q_accept_S3 t0) (not (g2 (d2 t0))) ) (and (laB q_accept_S3 (tau false true t0)) (> (laC
      q_accept_S3 (tau false true t0)) (laC q_accept_S3 t0))) ))
(assert (=> (and (laB q_accept_S3 t0) (not (g2 (d2 t0))) ) (and (laB q_accept_S3 (tau false false t0)) (> (laC
      q_accept_S3 (tau false false t0)) (laC q_accept_S3 t0))) ))
(assert (and (= (d2 (tau true true t0)) (tau2 (g1 (d1 t0)) true (d2 t0))) (= (d2 (tau true false t0)) (tau2 (g1 (d1 t0)
      ) false (d2 t0))) (= (d2 (tau false true t0)) (tau2 (g1 (d1 t0)) true (d2 t0))) (= (d2 (tau false false t0)) (
      tau2 (g1 (d1 t0)) false (d2 t0)))))
(assert (and (= (d1 (tau true true t0)) (tau1 (g2 (d2 t0)) true (d1 t0))) (= (d1 (tau true false t0)) (tau1 (g2 (d2 t0)
      ) true (d1 t0))) (= (d1 (tau false true t0)) (tau1 (g2 (d2 t0)) false (d1 t0))) (= (d1 (tau false false t0)) (
      tau1 (g2 (d2 t0)) false (d1 t0)))))
; encoded state t0

(assert (laB q_T0_init t0))
(assert (=> (and (laB q_accept_all t1) ) false ))
(assert (=> (and (laB q_accept_S2 t1) (not (g1 (d1 t1))) ) (and (laB q_accept_S2 (tau true true t1)) (> (laC
      q_accept_S2 (tau true true t1)) (laC q_accept_S2 t1))) ))
(assert (=> (and (laB q_accept_S2 t1) (not (g1 (d1 t1))) ) (and (laB q_accept_S2 (tau true false t1)) (> (laC
      q_accept_S2 (tau true false t1)) (laC q_accept_S2 t1))) ))
(assert (=> (and (laB q_accept_S2 t1) (not (g1 (d1 t1))) ) (and (laB q_accept_S2 (tau false true t1)) (> (laC
      q_accept_S2 (tau false true t1)) (laC q_accept_S2 t1))) ))
(assert (=> (and (laB q_accept_S2 t1) (not (g1 (d1 t1))) ) (and (laB q_accept_S2 (tau false false t1)) (> (laC
      q_accept_S2 (tau false false t1)) (laC q_accept_S2 t1))) ))
(assert (=> (and (laB q_T0_init t1) (and (not (g2 (d2 t1))) (g1 (d1 t1))) ) (laB q_T0_init (tau true true t1)) ))
(assert (=> (and (laB q_T0_init t1) (and (not (g2 (d2 t1))) (g1 (d1 t1))) ) (laB q_T0_init (tau true false t1)) ))
(assert (=> (and (laB q_T0_init t1) (and (not (g2 (d2 t1))) (g1 (d1 t1))) ) (laB q_T0_init (tau false true t1)) ))
(assert (=> (and (laB q_T0_init t1) (and (not (g2 (d2 t1))) (g1 (d1 t1))) ) (laB q_T0_init (tau false false t1)) ))
(assert (=> (and (laB q_T0_init t1) (and (g2 (d2 t1)) (g1 (d1 t1))) ) false ))
(assert (=> (and (laB q_T0_init t1) (not (g1 (d1 t1))) ) (laB q_T0_init (tau true true t1)) ))
(assert (=> (and (laB q_T0_init t1) (not (g1 (d1 t1))) ) (laB q_T0_init (tau true false t1)) ))
(assert (=> (and (laB q_T0_init t1) (not (g1 (d1 t1))) ) (laB q_T0_init (tau false true t1)) ))
(assert (=> (and (laB q_T0_init t1) (not (g1 (d1 t1))) ) (laB q_T0_init (tau false false t1)) ))
(assert (=> (and (laB q_T0_init t1) (not (g2 (d2 t1))) ) (laB q_accept_S3 (tau true true t1)) ))
(assert (=> (and (laB q_T0_init t1) (not (g2 (d2 t1))) ) (laB q_accept_S3 (tau false true t1)) ))
(assert (=> (and (laB q_T0_init t1) (not (g1 (d1 t1))) ) (laB q_accept_S2 (tau true true t1)) ))
(assert (=> (and (laB q_T0_init t1) (not (g1 (d1 t1))) ) (laB q_accept_S2 (tau true false t1)) ))
(assert (=> (and (laB q_accept_S3 t1) (not (g2 (d2 t1))) ) (and (laB q_accept_S3 (tau true true t1)) (> (laC
      q_accept_S3 (tau true true t1)) (laC q_accept_S3 t1))) ))
(assert (=> (and (laB q_accept_S3 t1) (not (g2 (d2 t1))) ) (and (laB q_accept_S3 (tau true false t1)) (> (laC
      q_accept_S3 (tau true false t1)) (laC q_accept_S3 t1))) ))
(assert (=> (and (laB q_accept_S3 t1) (not (g2 (d2 t1))) ) (and (laB q_accept_S3 (tau false true t1)) (> (laC
      q_accept_S3 (tau false true t1)) (laC q_accept_S3 t1))) ))
(assert (=> (and (laB q_accept_S3 t1) (not (g2 (d2 t1))) ) (and (laB q_accept_S3 (tau false false t1)) (> (laC
      q_accept_S3 (tau false false t1)) (laC q_accept_S3 t1))) ))
(assert (and (= (d2 (tau true true t1)) (tau2 (g1 (d1 t1)) true (d2 t1))) (= (d2 (tau true false t1)) (tau2 (g1 (d1 t1)
      ) false (d2 t1))) (= (d2 (tau false true t1)) (tau2 (g1 (d1 t1)) true (d2 t1))) (= (d2 (tau false false t1)) (
      tau2 (g1 (d1 t1)) false (d2 t1)))))
(assert (and (= (d1 (tau true true t1)) (tau1 (g2 (d2 t1)) true (d1 t1))) (= (d1 (tau true false t1)) (tau1 (g2 (d2 t1)
      ) true (d1 t1))) (= (d1 (tau false true t1)) (tau1 (g2 (d2 t1)) false (d1 t1))) (= (d1 (tau false false t1)) (
      tau1 (g2 (d2 t1)) false (d1 t1)))))
; encoded state t1

(assert (or (= (tau true true t0) t0) (= (tau true true t0) t1)))
(assert (or (= (tau true false t0) t0) (= (tau true false t0) t1)))
(assert (or (= (tau false true t0) t0) (= (tau false true t0) t1)))
(assert (or (= (tau false false t0) t0) (= (tau false false t0) t1)))
(assert (or (= (tau true true t1) t0) (= (tau true true t1) t1)))
(assert (or (= (tau true false t1) t0) (= (tau true false t1) t1)))
(assert (or (= (tau false true t1) t0) (= (tau false true t1) t1)))
(assert (or (= (tau false false t1) t0) (= (tau false false t1) t1)))

(check)
(echo "(((tau true true t0)) ")(eval (tau true true t0))
```

```
84 (echo "(((tau true true t1)) ")(eval (tau true true t1))
85 (echo "(((tau true false t0)) ")(eval (tau true false t0))
86 (echo "(((tau true false t1)) ")(eval (tau true false t1))
87 (echo "(((tau false true t0)) ")(eval (tau false true t0))
88 (echo "(((tau false true t1)) ")(eval (tau false true t1))
89 (echo "(((tau false false t0)) ")(eval (tau false false t0))
90 (echo "(((tau false false t1)) ")(eval (tau false false t1))
91 (echo "(((tau1 true true (d1 t0))) ")(eval (tau1 true true (d1 t0)))
92 (echo "(((tau1 true true (d1 t1))) ")(eval (tau1 true true (d1 t1)))
93 (echo "(((tau1 true false (d1 t0))) ")(eval (tau1 true false (d1 t0)))
94 (echo "(((tau1 true false (d1 t1))) ")(eval (tau1 true false (d1 t1)))
95 (echo "(((tau1 false true (d1 t0))) ")(eval (tau1 false true (d1 t0)))
96 (echo "(((tau1 false true (d1 t1))) ")(eval (tau1 false true (d1 t1)))
97 (echo "(((tau1 false false (d1 t0))) ")(eval (tau1 false false (d1 t0)))
98 (echo "(((tau1 false false (d1 t1))) ")(eval (tau1 false false (d1 t1)))
99 (echo "(((tau2 true true (d2 t0))) ")(eval (tau2 true true (d2 t0)))
100 (echo "(((tau2 true true (d2 t1))) ")(eval (tau2 true true (d2 t1)))
101 (echo "(((tau2 true false (d2 t0))) ")(eval (tau2 true false (d2 t0)))
102 (echo "(((tau2 true false (d2 t1))) ")(eval (tau2 true false (d2 t1)))
103 (echo "(((tau2 false true (d2 t0))) ")(eval (tau2 false true (d2 t0)))
104 (echo "(((tau2 false true (d2 t1))) ")(eval (tau2 false true (d2 t1)))
105 (echo "(((tau2 false false (d2 t0))) ")(eval (tau2 false false (d2 t0)))
106 (echo "(((tau2 false false (d2 t1))) ")(eval (tau2 false false (d2 t1)))
107 (echo "(((g2 (d2 t0))) ")(eval (g2 (d2 t0)))
108 (echo "(((g2 (d2 t1))) ")(eval (g2 (d2 t1)))
109 (echo "(((g1 (d1 t0))) ")(eval (g1 (d1 t0)))
110 (echo "(((g1 (d1 t1))) ")(eval (g1 (d1 t1)))
111 (echo "(((d2 t0)) ")(eval (d2 t0))
112 (echo "(((d2 t1)) ")(eval (d2 t1))
113 (echo "(((d1 t0)) ")(eval (d1 t0))
114 (echo "(((d1 t1)) ")(eval (d1 t1))
115
116 (show-stats)
117 (exit)
```

There are only a few things that differ between the SMT instance and the Yices instance.

Lines 1, 116
When using CVC4 and Z3, things like setting a time out or getting statistics can be added in the command line when doing the solver call. For the Yices input format however, we have to add these things into the file that includes our constraint system.

Lines 3 - 13
The data types and functions are defined here. The Yices input format uses a slightly different semantic than the SMT-Lib 2.0 to do that.

Lines 19 - 44
`t0` is encoded. The difference to the SMT instance is that we have to enroll the universal quantifiers `r1` and `r2` because the Yices input format does not support universal quantifiers.

Lines 46 - 71
`t1` is encoded in the same fashion as `t0` above.

Lines 73 - 80
Since we want to make sure that we only use those transition system states constrained in the constrained system, we define that.

Line 82
We again tell the solver to do its job and solve the constraint system.

Line 83 - 114
Another difference to the SMT instance can be found here. The way we get the solver to tell us what the model looks like differs from the SMT instance. In order to get a similar output to that of the SMT instance, we first echo the transition we want to see the result too and than evaluate the transition to get the output.

Line 117
We tell the solver to exit.

6.3.4 The Output

If the solver finds a model, we have to output this model somehow. Party has this function that takes the solver output and builds a dot graph with appropriate labeling. The dot graph is either outputted in the terminal or one can choose to save the dot graph to a file. Figure 6.4 shows the dot graph for the instance shown in Example 35.

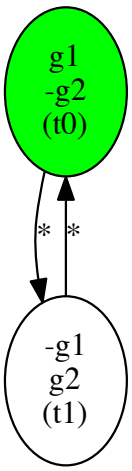


Figure 6.4: Dot Graph for Response Request Yices, Moore, Weak Symmetry

CHAPTER 7

EXPERIMENTS AND EVALUATIONS

In the past chapters we went over all preliminaries to start testing. This chapter deals with the experiment setup, the experiments and the results. All three parts are explained in more detail below.

7.1 Setup

We used a Xeon E3 3.6Ghz 32GB RAM Singlecore for testing. All tests were done with EDACC [3]. EDACC is a framework that helps with the execution and testing of algorithms.

We used an overall timeout of 3600 seconds and a solver call timeout of 1800 seconds. We also ran a few experiments were we set the timeout to 7200 seconds. If results are shown with the timeout being 7200 seconds instead of 3600 seconds, it will be explicitly mentioned.

CVC4 We used CVC4 in version 1.4 and used the option `--finite-model-find`.

Z3 We used the Z3 version 4.3.1 and had to turn off ematching.

Yices Yices was used in version 2.3.1 and we used the standard setting. We did not use the SMT-Lib 2.0 with Yices, but rather Yices' own input format. However, the code should be compatible with any later version of Yices that supports the Yices input language.

7.1.1 Tool Changes for Testing Purposes

We had to change a few minor things in the tool for running the experiments.

Input For simplicity reasons during testing, we had to combine the two input files with the extensions '.arc' and '.ltl' to one file with the extension '.al'. The file consisted of the architecture, then a break line starting with '=' and then the specification.

Example 36. (.al)
The following are Examples 32 and 33 with a breakline seperating them.

```
1 [P] p1 p2
2 [O] [env] r1 r2
```

```
3 [I] [p1] r1 g2
4 [O] [p1] g1
5 [I] [p2] r2 g1
6 [O] [p2] g2
7 ================================================
8
9 G((r1=1) -> (F(g1=1))) * G((r2=1) -> (F(g2=1))) * G(!((g1=1) * (g2=1)));
```

Output The times shown in the following results do not include the building time for the dot graph.

7.2 Architectures

We tested quite a few different architectures. We are going to show two architectures with two system processes here. More can be found in the appendix on Page 65.

7.2.1 Architectures with Two System Processes

Figure 7.1 shows a fully known architecture with two system processes. Both system processes, p_1 and p_2, get both requests, r_1 and r_2, as input and system process p_1 outputs grant g_1 and gets grant g_2 as input, while system process p_2 outputs grant g_2 and gets grant g_1 as input.

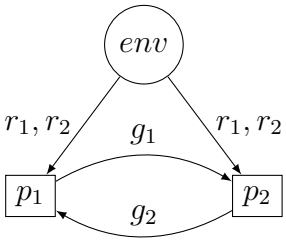


Figure 7.1: Architecture 2a

Figure 7.2 is not a fully known architecture because the sets of inputs for the system processes p_1 and p_2 are not equal to the set of outputs of the environment O_{env}. The only difference to the architecture shown in Figure 7.1 is that system process p_1 only gets request r_1 as input from the environment and system process p_2 only gets request r_2 as input from the environment.

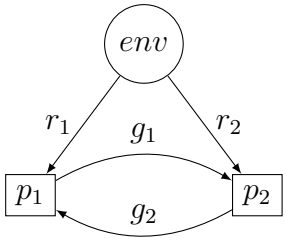


Figure 7.2: Architecture 2b

7.3 LTL Specifications

We used three specifications for testing. We tested the request response specification in different sizes. Furthermore, we tested a full arbiter [12] and a Pnueli arbiter [6], both also in different sizes. Below are more details about the specifications.

7.3.1 Request Response

Figure 7.3 shows the generalized request response specification [6]. The specification says that

$$\bigwedge_{i=0,..,n} \Box(r_i \rightarrow \Diamond g_i) \wedge \bigwedge_{\substack{i,j=0,..,n \\ \text{where } i \neq j}} \Box(\neg g_i \vee \neg g_j)$$

Figure 7.3: Request Response Specification

a request should eventually be followed by a grant and no two grants should be given at the same time (mutual exclusion).

We have already seen this specification in size 2 throughout this thesis.

$$\Box(r_1 \rightarrow \Diamond g_1) \wedge \Box(r_2 \rightarrow \Diamond g_2) \wedge \Box(\neg g_1 \vee \neg g_2)$$

7.3.2 Pnueli Arbiter

The Pnueli arbiter [6] has n input lines in which clients request permissions and n output lines in which the clients are granted permission.

We seperate the assumption and the guarantee into two parts and go into further detail below.

$$\text{Assumption} \rightarrow \text{Guarantee}$$

We assume that

- initially all requests are set to zero,
- once a request has been made it cannot be withdrawn,
- and that the clients are fair, that is once a grant to a certain client has been given, the client eventually releases the resource by lowering its request line.

The assumption on the environment can be found in Figure 7.4.

As for the guarantee, we expect

- the arbiter to initially give no grants,
- give at most one grant at a time (mutual exclusion),

- give only requested grants,
- maintain a grant as long as it is requested,
- to supply (eventually) every request,
- and to take grants that are no longer needed.

The guarantee can be found in Figure 7.5.

$$\bigwedge_i (\overline{r_i} \wedge \Box((r_i \neq g_i) \rightarrow (r_i = \bigcirc r_i)) \wedge \Box((r_i \wedge g_i) \rightarrow \Diamond \overline{r_i}))$$

Figure 7.4: Pnueli Arbiter - Assumption [6]

$$\bigwedge_{i \neq j} \Box\neg(g_i \wedge g_j) \wedge \bigwedge_i \left(g_i \wedge \left(\begin{array}{l} \Box((r_i = g_9) \rightarrow g_i = \bigcirc g_i)) \wedge \\ \Box((r_i \wedge \overline{g_i}) \rightarrow \Diamond g_i) \wedge \\ \Box((\overline{r_i} \wedge g_i) \rightarrow \Diamond g_i) \end{array} \right) \right)$$

Figure 7.5: Pnueli Arbiter - Guarantee [6]

7.3.3 Full Arbiter

The full arbiter, as shown in Figure 7.6, has never spurious grants and every grant is lowered unless it keeps getting requested. Every request is eventually granted and there is at most one grant given at a time (mutual exclusion).

$$\bigwedge_{i \neq j} \Box\neg(g_i \wedge g_j) \wedge \bigwedge_i \left(\begin{array}{l} \neg((\overline{r_i} \wedge \overline{g_i}) \,\mathcal{U}\, (\overline{r_i} \wedge g_i)) \wedge \\ \neg\Diamond(g_i \wedge \bigcirc(\overline{r_i} \wedge \overline{g_i}) \wedge \bigcirc((\overline{r_i} \wedge \overline{g_i}) \,\mathcal{U}\, (g_i \wedge \overline{r_i}))) \wedge \\ \Box((\overline{r_i} \wedge g_i) \rightarrow \Diamond((r_i \wedge g_i) \vee \overline{g_i}))) \wedge \\ \Box(r_i \rightarrow \Diamond g_i) \end{array} \right)$$

Figure 7.6: Full Arbiter

7.4 Experiments

We tested the request response specification, the Pnueli arbiter and the full arbiter with all architectures that can be found in the Appendix A.2 on Page 65 in the appropriate sizes. The Pnueli and the full arbiter timed out for all architectures that had more than three system processes and even for some with three system processes. The request response specification could be tested for architectures with up to 7 clients.

7.5 Results

This section shows results of our experiments. We used several abbreviations in the legends of the figures which can be found in Table 7.1. More results can be found in the Appendix A.3 starting on Page 69.

C	Consistency
WS	Weak Symmetry
Me	Mealy
Mo	Moore
C4	CVC4
Y	Yices

Table 7.1: Legend Abbreveations

7.5.1 Request Response Specification

Let us first take a look at the results of the request response specification with architectures of type 2b or 3a1, i.e., architectures that get the grants from all other system processes as inputs, but only the request they can grant as input from the environment. The architectures can be found in the appendix starting on Page A.2

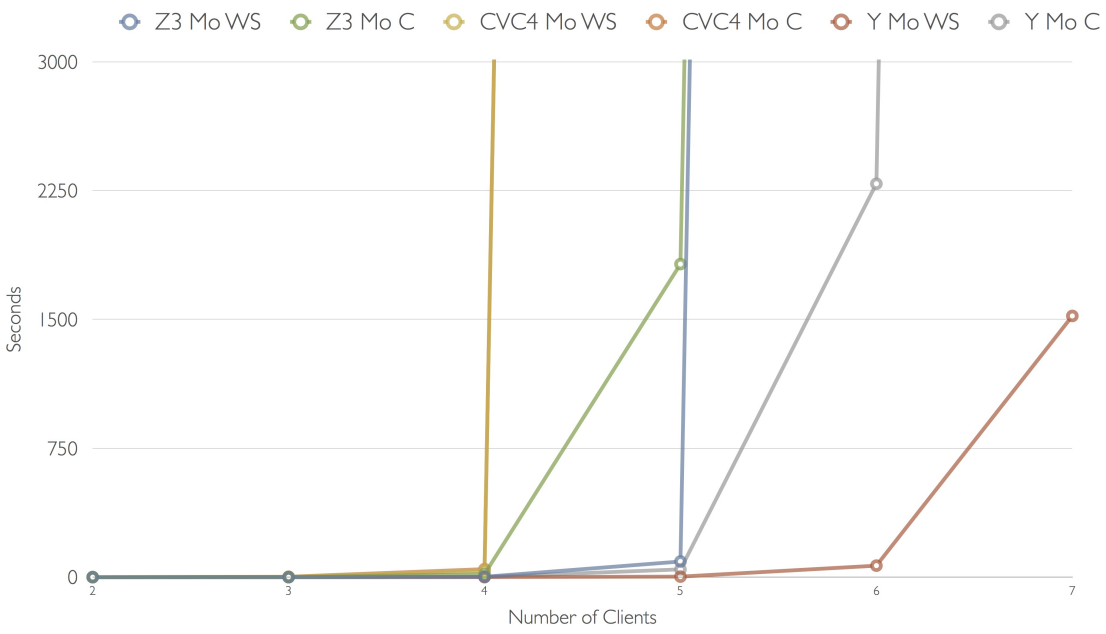


Figure 7.7: Request Response Specification - Results for Moore

Consistency vs. Weak Symmetry In Figure 7.7, we show all results for moore transition systems. Weak Symmetry provided the better results for all three solvers throughout the iteration over the number of clients. Figure 7.8 shows the results for mealy transition systems. We can see the same effect as for moore transition systems, namely that weak symmetry always outperformed consistency.

Mealy vs. Moore Figure 7.9 shows the results for consistency constraints. Let us first take a look at Z3. Here, moore outperformed mealy when the number of clients was smaller than 5. However for 5 clients mealy was faster than moore. For Yices, moore was clearly faster than

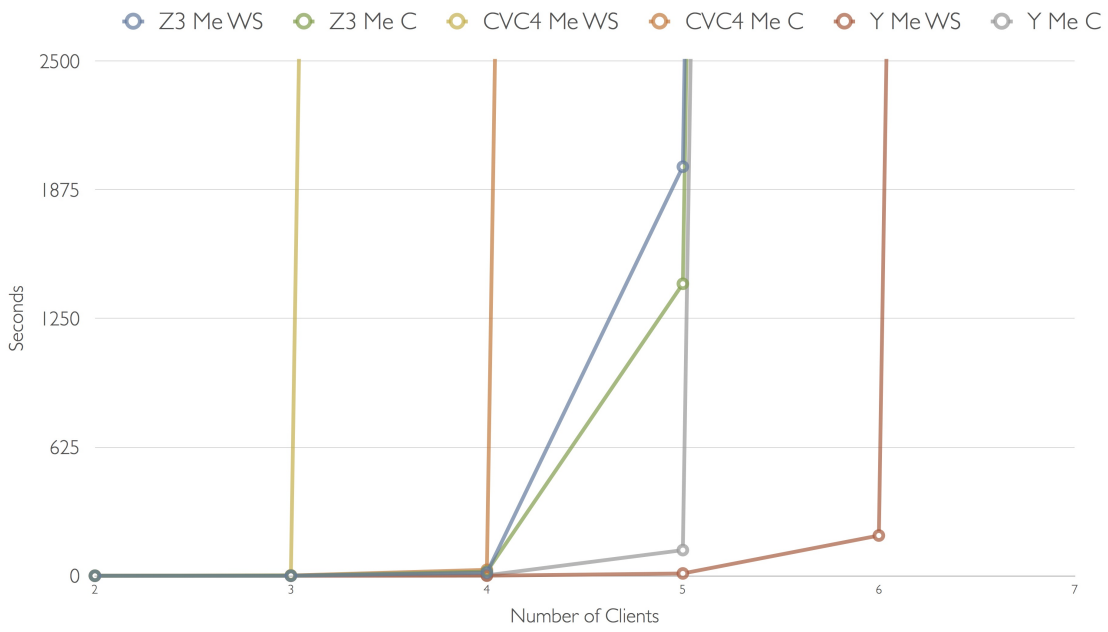


Figure 7.8: Request Response Specification - Results for Mealy

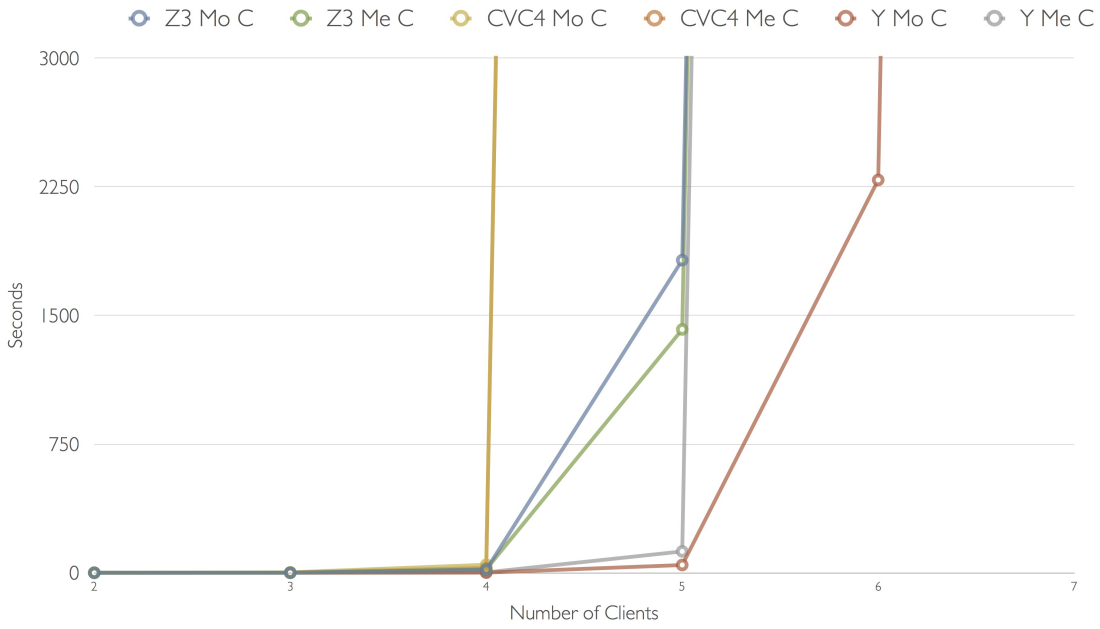


Figure 7.9: Request Response Specification - Results for Consistency

mealy and even managed to find a model for 6 clients. The results for CVC4 were better when using mealy than when using a moore constraint system.

In Figure 7.10 we can see the results for weak symmetry constraints. For all three solvers, the moore constraint systems clearly outperformed the mealy constraint systems.

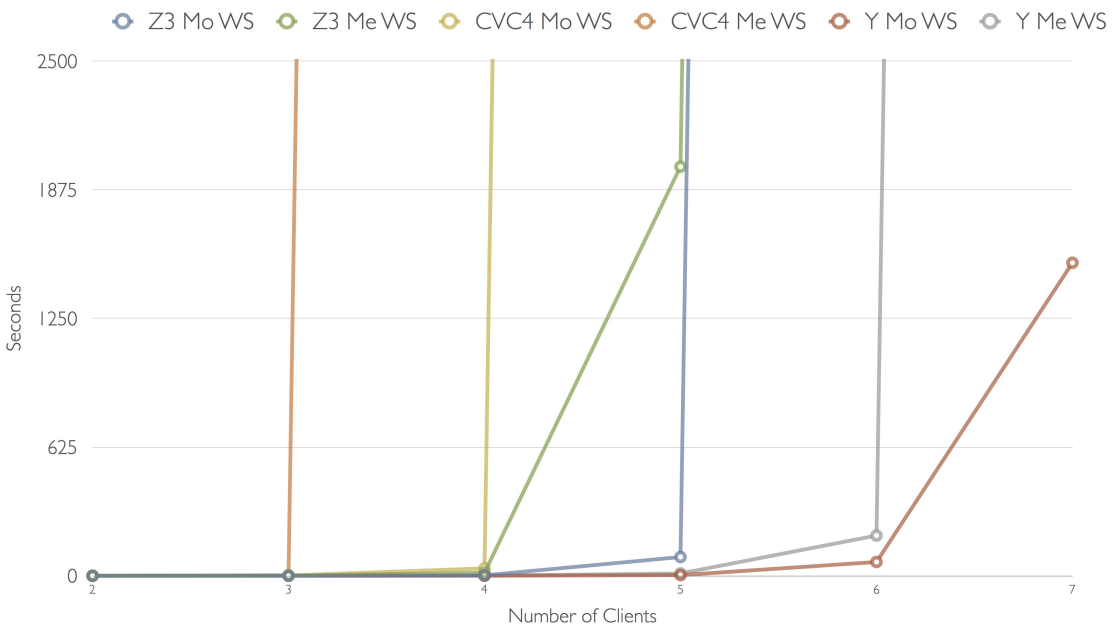


Figure 7.10: Request Response Specification - Results for Weak Symmetry

CVC4 vs. Yices vs. Z3 When taking a look at Figures 7.7, 7.8, 7.9 and 7.10, we can see that Z3 and Yices clearly outperform CVC4. This was the case in almost all of the experiments we did. In most of the experiments, Yices also clearly outperforms Z3.

Summary In this experiment, the combination of using Yices as the solver with weak symmetry constraints and a moore transition system as output delivered the best results.

7.5.2 Types of Architectures - Request Response Specification

We tested quite a few different architectures and in this section we are looking at them when used for a request response specification, which can be Found in Figure 7.3.

Architectures with Two System Parameters The architectures we are looking at can be found in Examples 7.1, 7.2 and in the Appendix A.2 on Page 65.

Figure 7.11 shows the comparison of two architectures of size two. The lines marked with *Bound 2* are for the same architectures as the ones without, however the size of the system process transition systems was bound to two here. There are not many mentionable differences to see here. Z3 still did better than CVC4 and Yices did better than Z3. Constraint systems looking for a moore output did slightly better for Z3, but other than that all results are very close together. We can also see that for this specification and these types of architectures, it does not matter whether we bound the size of the system process transition systems or not.

Architectures with Three System Parameters The architectures used can be found in the Appendix on Page A.2.

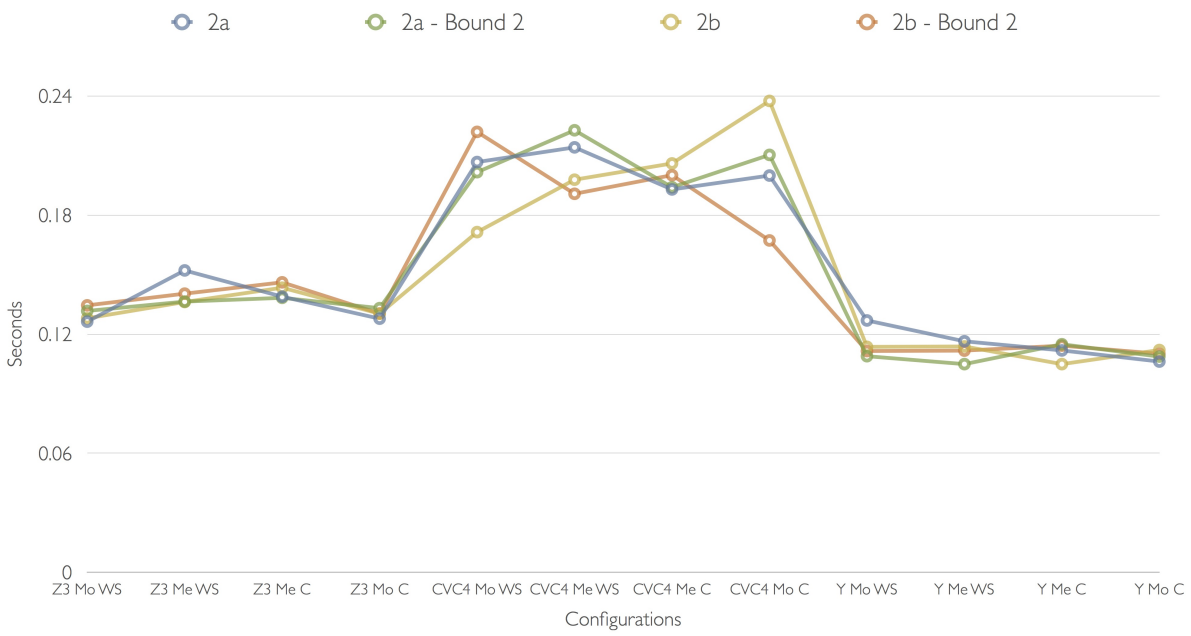


Figure 7.11: Request Response and Comparison of Architectures of Size 2

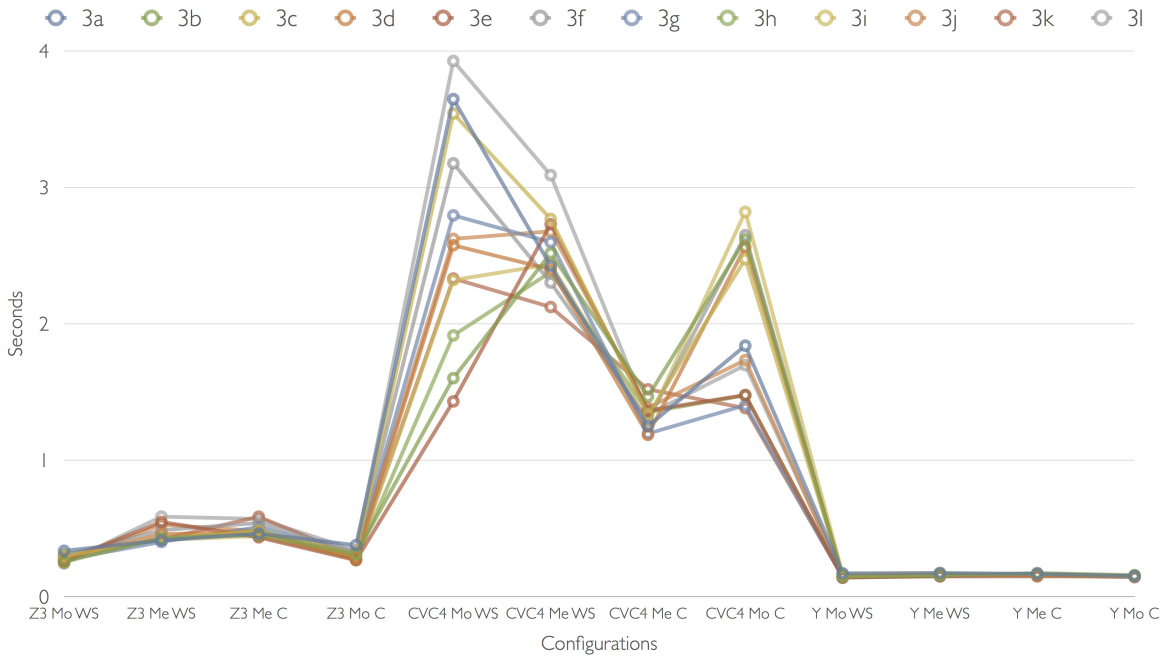


Figure 7.12: Request Response and Comparison of Architectures of Size 3

Figure 7.12 shows the comparison of architectures of size three. Z3 again outperformed CVC4 and Yices outperformed Z3. We can also again see, that Z3 provides slightly faster results when we are looking for a moore model. Yices does not show any differences whether we are using consistency or weak symmetry constraints or whether we are looking for a mealy or moore model. CVC4 reacts differently for any architecture. However, when using consistency constraints and looking for a mealy model, the time used is very close together for all architectures.

In this comparison we cannot see whether there is one architecture or a set of architecture for which faster results can be provided than for others.

7.5.3 Types of Architectures - Pnueli Arbiter

In this section we look at different sets of architectures and compare them for the Pnueli Arbiter. We left out the results for CVC4, because this solver timed out for all of the experiments in this subsection.

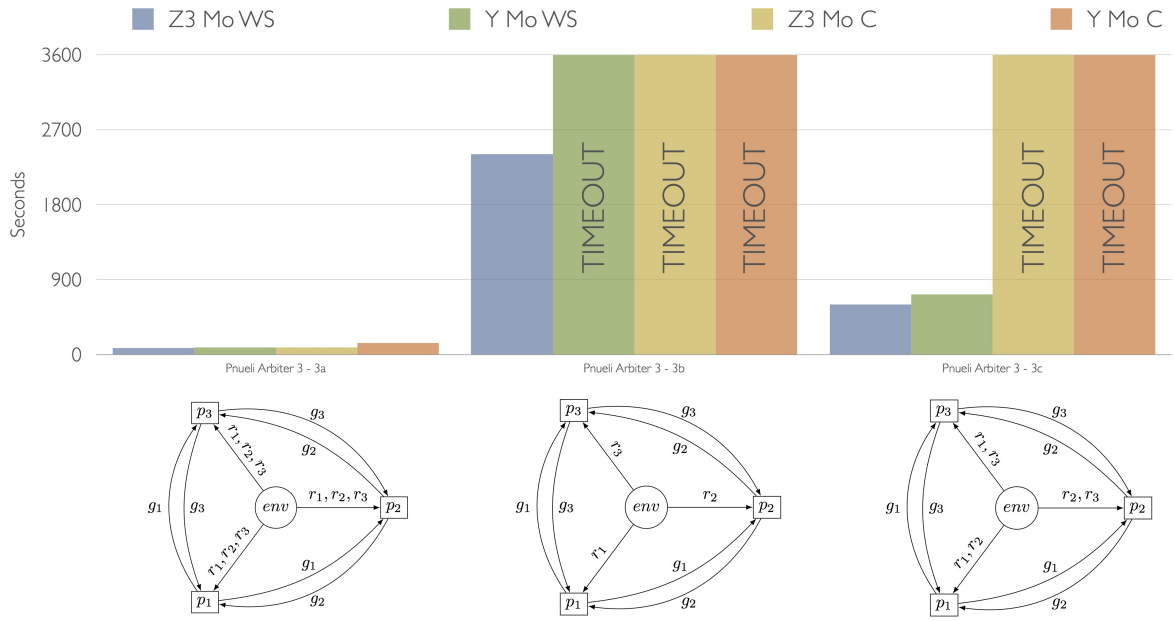


Figure 7.13: Pnueli Arbiter and Comparison of Architectures of Size 3 - Part 1

Figure 7.13 compares architectures where each system gets the grants from all other system processes as input. Figure 7.14 compares architectures where each system process only gets the grant of one other system process as input. Last but not least, Figure 7.15 compares architectures where system process p_2 gets the grants from the two other system processes as input, but p_1 and p_3 only get the grant from p_2 as input. The difference of the architectures in all three figures is the number of environment outputs a system process gets as input.

All three sets of examples show that the fastest results can be found when all system processes get all outputs of the environment as inputs. In Figure 7.13 and Figure 7.14 the results for the other two architectures were not very good. If results were found at all before the timeout, the combination of weak symmetry constraints and looking for a moore model delivered better results in most cases. Looking at Figure 7.15, there are huge differences whether or not the combination weak symmetry and moore was used or not. In all nine experiments, Z3 and Yices shared the spot of fastest solver.

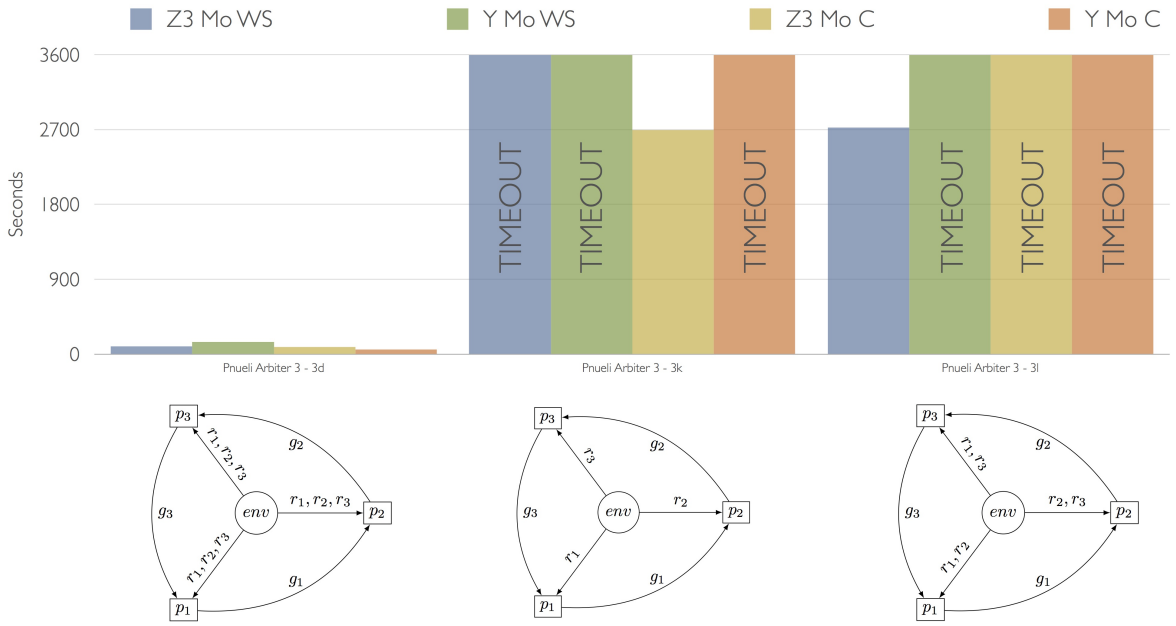


Figure 7.14: Pnueli Arbiter and Comparison of Architectures of Size 3 - Part 2

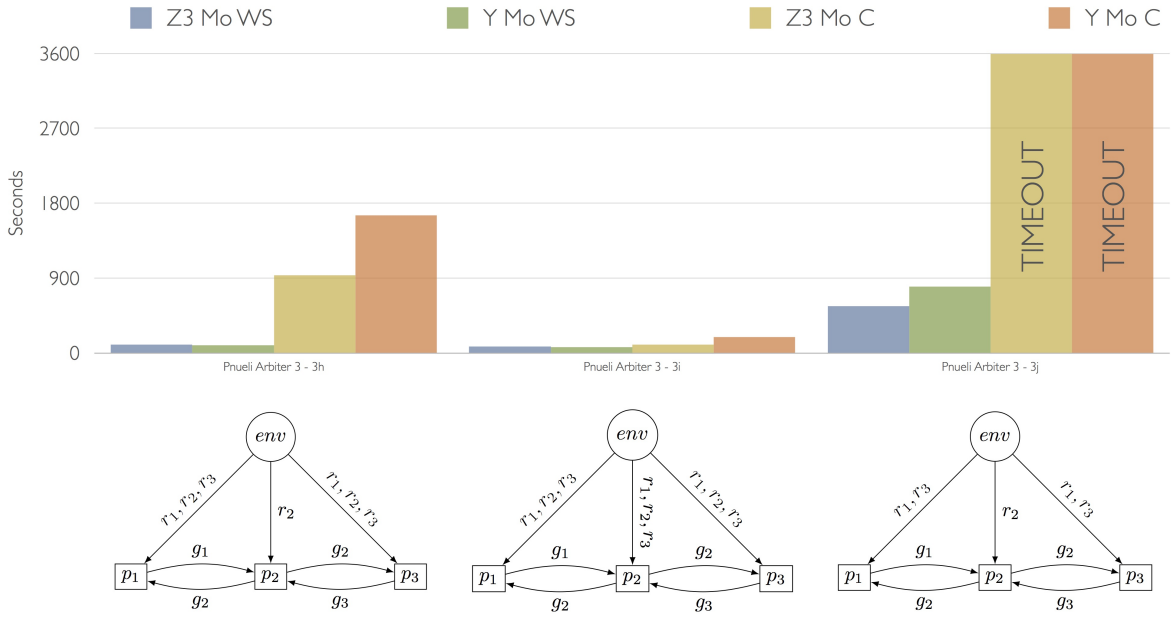


Figure 7.15: Pnueli Arbiter and Comparison of Architectures of Size 3 - Part 3

7.5.4 General Comparison

Overall, the combination of using Yices as the solver with weak symmetry constraints and a moore transition system as output delivered the best results. We think that using weak symmetry constraints and getting a moore transition system was part of this combination because of the way it influences the constraint system. The weak symmetry constraints are fewer than the

consistency constraints, meaning less constraints have to be solved. If we look for a moore transition system, the uninterpreted functions do not get any input directions, meaning that it is easier to find an interpretation.

7.6 Summary

We saw several architectures and tested them with three different types of specifications. We looked at different aspects and overall found that using Yices as solver, weak symmetry constraints and getting a moore model as output, delivered the best results. When looking at the architectures in detail, we found that, if possible, all system processes should get all environment outputs as inputs.

CHAPTER 8
CONCLUSION

We saw some background information and looked in detail at bounded synthesis for monolithic systems. We built a constraint system and extended it for bounded synthesis of distributed architectures. We introduced an input format, looked at the implementation of bounded synthesis for distributed architectures and last, but not least, did experiments and compared their results.

We found that CVC4 was the slowest of the three solvers tested and that Yices outperfmed Z3. We also found that using weak symmetry constraints instead of consistency constraints delivered better results. Also looking for a moore instead of a mealy model delivered also better results, even though mealy models can be smaller. In fact, we found that the best combination when doing bounded synthesis for distributed architectures was using Yices as the solver and weak symmetry constraints with the goal of finding a moore model. We also found that we usually get the best results, if all system processes get all outputs of the environment as input.

For future work, experiments could be run on a much larger scale with larger timeouts and more runs. There are many more specifications that can be tested. Another interesting topic that can be looked into, is testing bounded synthesis of distributed architectures with more solvers. More experiments with the size of the system process transition systems being bounded would also be interesting. It could also be tested how much the computation time differs from run to run when an instance is tested several times.

Overall we saw very interesting results. Even though the biggest model we found with a timeout of 3600 seconds was for a specification and an architecture with 7 clients, this is bound to change in the future. SMT solvers get faster, just as computation power rises. It is interesting to follow that development, especially because today's world relies more and more on technical systems. And it will be very interesting to see, how bounded synthesis will be able to help that development in the future.

Bibliography

[1] Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. LTL to büchi automata translation: Fast and more deterministic. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2012.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, May 2008.

[3] Adrian Balint, Daniel Diepold, Daniel Gall, Simon Gerber, Gregor Kapler, and Robert Retz. Edacc - an advanced platform for the experiment design, administration and analysis of empirical algorithms. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 586–599, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[6] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Saar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, May 2012.

[7] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for ltl synthesis. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 652–657, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Bruno Dutertre and Leonardo de Moura. The yices smt solver. Technical report, SRI International, 2006.

[10] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539, 2013.

[11] Ayrat Khalimov, Swen Jacobs, and Roderick Bloem. Party: Parameterized synthesis of token rings. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 928–933, Berlin, Heidelberg, 2013. Springer-Verlag.

[12] Ayrat Khalimov, Swen Jacobs, and Roderick Bloem. Towards efficient parameterized synthesis. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*, pages 108–127. Springer Berlin Heidelberg, 2013.

[13] Orna Kupferman and Moshe Y. Vardi. Safraless decision procedures. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '05, pages 531–542, Washington, DC, USA, 2005. IEEE Computer Society.

APPENDIX A

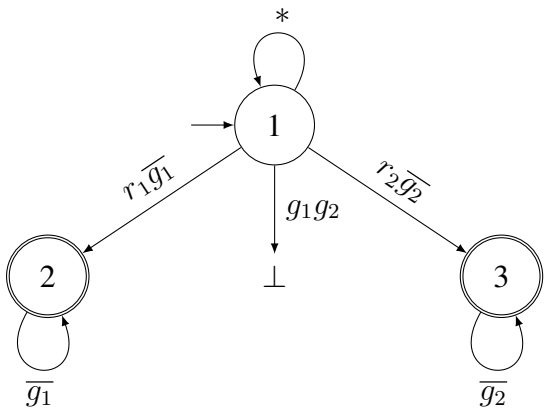
APPENDIX

A.1 Universal co-Büchi Automaton

The specification:

$$\varphi = \Box(r_1 \rightarrow \Diamond g_1) \wedge \Box(r_2 \rightarrow \Diamond g_2) \wedge \Box(\neg g_1 \vee \neg g_2)$$

The universal Co-Büchi Automaton



$\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, F)$
$\Sigma = \{r_1, r_2, g_1, g_2\}$
$\Upsilon = \{r_1, r_2\}$
$Q = \{1, 2, 3, \bot\}$
$q_0 = \{1\}$
$F = \{2, 3\}$

$\delta(1, r_1r_2g_1g_2) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge (\bot, r_1r_2) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge false = false$
$\delta(1, r_1r_2g_1\overline{g_2}) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge (3, r_1r_2) \wedge (3, \overline{r_1}r_2)$
$\delta(1, r_1r_2\overline{g_1}g_2) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge (2, r_1r_2) \wedge (2, r_1\overline{r_2})$
$\delta(1, r_1r_2\overline{g_1g_2}) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge (2, r_1r_2) \wedge (2, r_1\overline{r_2}) \wedge (3, r_1r_2) \wedge (3, \overline{r_1}r_2)$

$\delta(1, r_1\overline{r_2}g_1g_2) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge (\bot, r_1\overline{r_2}) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge false = false$

$\delta(1, r_1\overline{r_2}g_1\overline{g_2}) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2})$
$\delta(1, r_1\overline{r_2}\overline{g_1}g_2) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge (2, r_1r_2) \wedge (2, r_1\overline{r_2})$
$\delta(1, r_1\overline{r_2}\overline{g_1}\overline{g_2}) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge (2, r_1r_2) \wedge (2, r_1\overline{r_2})$

$\delta(1, \overline{r_1}r_2g_1g_2) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge (\bot, \overline{r_1}r_2) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge false = false$
$\delta(1, \overline{r_1}r_2g_1\overline{g_2}) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge (3, r_1r_2) \wedge (3, \overline{r_1}r_2)$
$\delta(1, \overline{r_1}r_2\overline{g_1}g_2) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2})$
$\delta(1, \overline{r_1}r_2\overline{g_1g_2}) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge (3, r_1r_2) \wedge (3, \overline{r_1}r_2)$

$\delta(1, \overline{r_1r_2}g_1g_2) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge (\bot, \overline{r_1r_2}) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2}) \wedge false = false$
$\delta(1, \overline{r_1r_2}g_1\overline{g_2}) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2})$
$\delta(1, \overline{r_1r_2}\overline{g_1}g_2) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2})$
$\delta(1, \overline{r_1r_2}\overline{g_1}\overline{g_2}) = (1, r_1r_2) \wedge (1, r_1\overline{r_2}) \wedge (1, \overline{r_1}r_2) \wedge (1, \overline{r_1r_2})$

$\delta(2, r_1r_2g_1g_2) = true$
$\delta(2, r_1r_2g_1\overline{g_2}) = true$
$\delta(2, r_1r_2\overline{g_1}g_2) = (2, r_1r_2) \wedge (2, r_1\overline{r_2}) \wedge (2, \overline{r_1}r_2) \wedge (2, \overline{r_1r_2})$
$\delta(2, r_1r_2\overline{g_1g_2}) = (2, r_1r_2) \wedge (2, r_1\overline{r_2}) \wedge (2, \overline{r_1}r_2) \wedge (2, \overline{r_1r_2})$

$\delta(2, r_1\overline{r_2}g_1g_2) = true$
$\delta(2, r_1\overline{r_2}g_1\overline{g_2}) = true$
$\delta(2, r_1\overline{r_2}\overline{g_1}g_2) = (2, r_1r_2) \wedge (2, r_1\overline{r_2}) \wedge (2, \overline{r_1}r_2) \wedge (2, \overline{r_1r_2})$
$\delta(2, r_1\overline{r_2}\overline{g_1}\overline{g_2}) = (2, r_1r_2) \wedge (2, r_1\overline{r_2}) \wedge (2, \overline{r_1}r_2) \wedge (2, \overline{r_1r_2})$

$\delta(2, \overline{r_1}r_2g_1g_2) = true$
$\delta(2, \overline{r_1}r_2g_1\overline{g_2}) = true$
$\delta(2, \overline{r_1}r_2\overline{g_1}g_2) = (2, r_1r_2) \wedge (2, r_1\overline{r_2}) \wedge (2, \overline{r_1}r_2) \wedge (2, \overline{r_1r_2})$
$\delta(2, \overline{r_1}r_2\overline{g_1}\overline{g_2}) = (2, r_1r_2) \wedge (2, r_1\overline{r_2}) \wedge (2, \overline{r_1}r_2) \wedge (2, \overline{r_1r_2})$

$\delta(2, \overline{r_1r_2}g_1g_2) = true$
$\delta(2, \overline{r_1r_2}g_1\overline{g_2}) = true$
$\delta(2, \overline{r_1r_2}\overline{g_1}g_2) = (2, r_1r_2) \wedge (2, r_1\overline{r_2}) \wedge (2, \overline{r_1}r_2) \wedge (2, \overline{r_1r_2})$
$\delta(2, \overline{r_1r_2}\overline{g_1}\overline{g_2}) = (2, r_1r_2) \wedge (2, r_1\overline{r_2}) \wedge (2, \overline{r_1}r_2) \wedge (2, \overline{r_1r_2})$

$\delta(3, r_1r_2g_1g_2) = true$
$\delta(3, r_1r_2g_1\overline{g_2}) = (3, r_1r_2) \wedge (3, r_1\overline{r_2}) \wedge (3, \overline{r_1}r_2) \wedge (3, \overline{r_1r_2})$
$\delta(3, r_1r_2\overline{g_1}g_2) = true$
$\delta(3, r_1r_2\overline{g_1}\overline{g_2}) = (3, r_1r_2) \wedge (3, r_1\overline{r_2}) \wedge (3, \overline{r_1}r_2) \wedge (3, \overline{r_1r_2})$

$\delta(3, r_1\overline{r_2}g_1g_2) = true$
$\delta(3, r_1\overline{r_2}g_1\overline{g_2}) = (3, r_1r_2) \wedge (3, r_1\overline{r_2}) \wedge (3, \overline{r_1}r_2) \wedge (3, \overline{r_1r_2})$

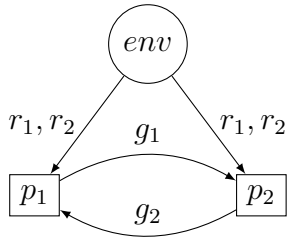
$\delta(3, r_1\overline{r_2 g_1} g_2) = true$
$\delta(3, r_1\overline{r_2 g_1 g_2}) = (3, r_1 r_2) \wedge (3, r_1\overline{r_2}) \wedge (3, \overline{r_1} r_2) \wedge (3, \overline{r_1 r_2})$

$\delta(3, \overline{r_1} r_2 g_1 g_2) = true$
$\delta(3, \overline{r_1} r_2 g_1\overline{g_2}) = (3, r_1 r_2) \wedge (3, r_1\overline{r_2}) \wedge (3, \overline{r_1} r_2) \wedge (3, \overline{r_1 r_2})$
$\delta(3, \overline{r_1} r_2\overline{g_1} g_2) = true$
$\delta(3, \overline{r_1} r_2\overline{g_1 g_2}) = (3, r_1 r_2) \wedge (3, r_1\overline{r_2}) \wedge (3, \overline{r_1} r_2) \wedge (3, \overline{r_1 r_2})$

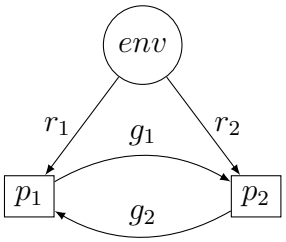
$\delta(3, \overline{r_1 r_2} g_1 g_2) = true$
$\delta(3, \overline{r_1 r_2} g_1\overline{g_2}) = (3, r_1 r_2) \wedge (3, r_1\overline{r_2}) \wedge (3, \overline{r_1} r_2) \wedge (3, \overline{r_1 r_2})$
$\delta(3, \overline{r_1 r_2 g_1} g_2) = true$
$\delta(3, \overline{r_1 r_2 g_1 g_2}) = (3, r_1 r_2) \wedge (3, r_1\overline{r_2}) \wedge (3, \overline{r_1} r_2) \wedge (3, \overline{r_1 r_2})$

A.2 Architectures

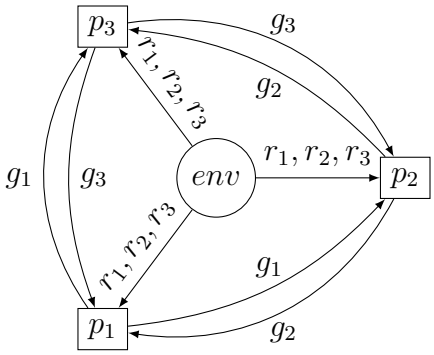
A.2.1 Architecture 2a



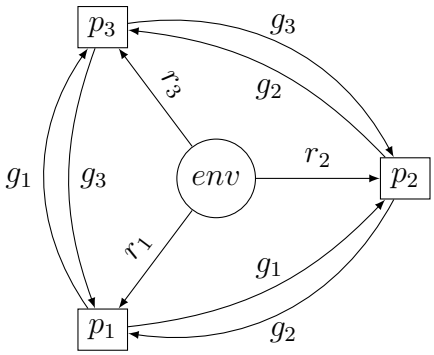
A.2.2 Architecture 2b



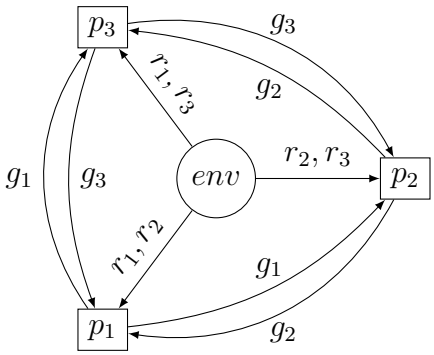
A.2.3 Architecture 3a



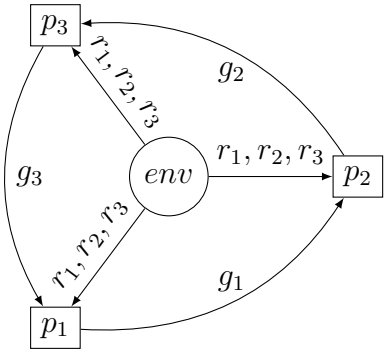
A.2.4 Architecture 3b



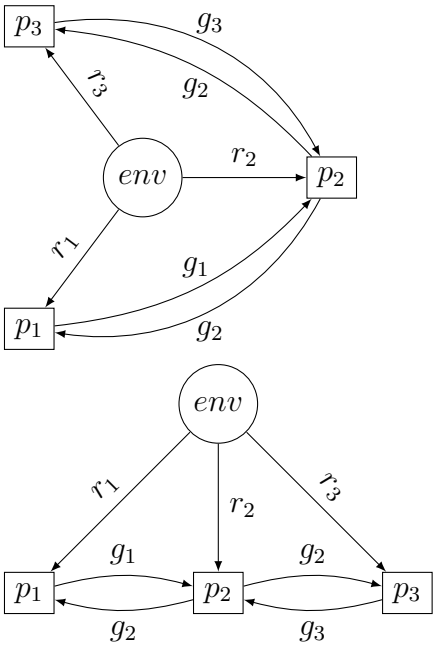
A.2.5 Architecture 3c



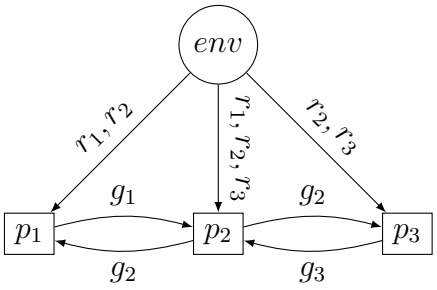
A.2.6 Architecture 3d



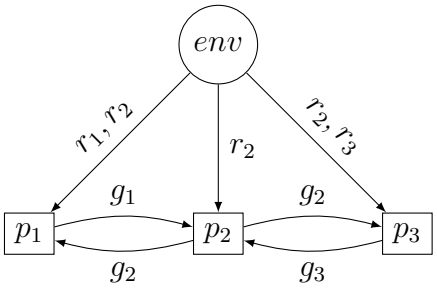
A.2.7 Architecture 3e



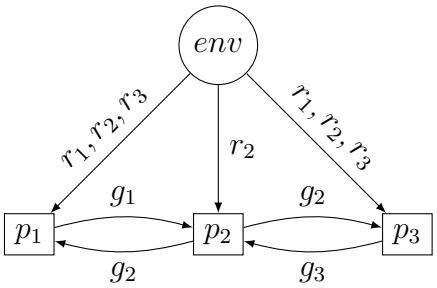
A.2.8 Architecture 3f



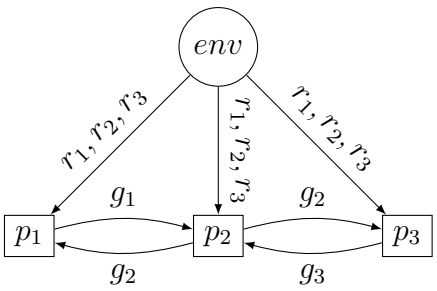
A.2.9 Architecture 3g



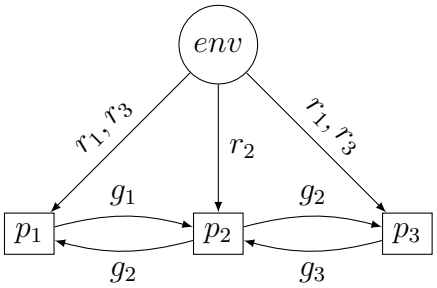
A.2.10 Architecture 3h



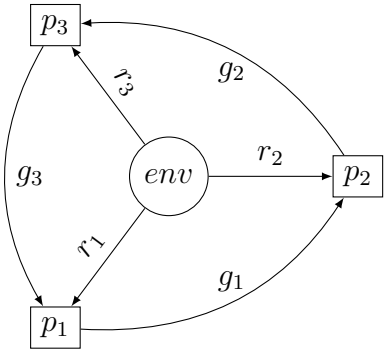
A.2.11 Architecture 3i



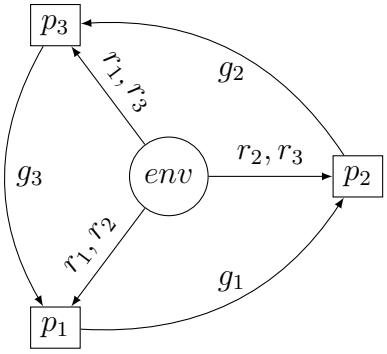
A.2.12 Architecture 3j



A.2.13 Architecture 3k



A.2.14 Architecture 3l



A.2.15 Architecture with 4 - 7 system processes

Architectures of Type xa1 Architectures which names end with a 1 have x system processes. Each system process gets the grants from all other system processes as inputs, however the only environment output this system process gets is the one corresponding to the grant it can give. An example architecture with 3 system processes can be found on Page A.2.4 in Subsection A.2.4.

Architectures of Type xax Architectures which names end with the same number they started with have x system processes. Each system process gets the grants from all other system processes as inputs and all outputs of the environment. An example architecture with 3 system processes can be found on Page A.2.3 in Subsection A.2.3.

A.3 Results

Results that have the tag *TO7200* behind the architecture name were tested with a timeout of 7200 seconds instead of 3600.

C	Consistency
WS	Weak Symmetry
Me	Mealy
Mo	Moore
C4	CVC4
Y	Yices

Table A.1: Full Arbiter 2

Architecture	Z3 Mo WS	Z3 Me WS	Z3 Me C	Z3 Mo C	C4 Mo WS	C4 Me WS	C4 Me C	C4 Mo C	Y Mo WS	Y Me WS	Y Me C	Y Mo C
2a	0.3441	0.2352	0.2463	0.3731	2.2103	70.793	0.8158	2.2073	0.1828	0.146	0.1501	0.1809
2b	170.177	0.4569	0.4607	3600	3600	5.0245	6.3434	3600	71.3535	0.2278	0.2189	3600

Table A.2: Full Arbiter 3

Architecture	Z3 Mo WS	Z3 Me WS	Z3 Me C	Z3 Mo C	C4 Mo WS	C4 Me WS	C4 Me C	C4 Mo C	Y Mo WS	Y Me WS	Y Me C	Y Mo C
3a	3600	269.174	68.4966	3600	3600	3600	3600	3600	78.8983	44.4466	38.2753	513.881
3c	3600	3600	3600	3600	3600	3600	3600	3600	2,016.71	3,079.58	3600	3600
3d	3600	100.323	559.694	3600	3600	3600	3600	3600	204.06	75.135	55.385	138.996
3h	3600	3600	3600	3600	3600	3600	3600	3600	1,878.84	1,894.46	3600	3600
3i	3600	144.057	114.183	3600	3600	3600	3600	3600	204.796	83.9674	47.9657	149.686

Table A.3: Pnueli Arbiter 2

Architecture	Z3 Mo WS	Z3 Me WS	Z3 Me C	Z3 Mo C	C4 Mo WS	C4 Me WS	C4 Me C	C4 Mo C	Y Mo WS	Y Me WS	Y Me C	Y Mo C
2a	0.6504	1.0691	0.9797	0.6521	39.4439	2086.16	115.074	72.4299	0.4801	0.6732	0.7102	0.4446
2a - Bound 2	0.6784	1.0666	0.8995	0.6854	72.4962	3600	1430.17	40.4078	0.4898	0.5431	0.8721	0.449
2b	1.5233	2.8836	2.922	1.5873	3600	3600	3600	3600	1.4119	2.7981	2.231	1.261

Table A.4: Pnueli Arbiter 3

Architecture	Z3 Mo WS	Z3 Me WS	Z3 Me C	Z3 Mo C	C4 Mo WS	C4 Me WS	C4 Me C	C4 Mo C	Y Mo WS	Y Me WS	Y Me C	Y Mo C
3a	74.6162	373.604	3600	86.1255	3600	3600	3600	3600	84.7713	791.43	850.819	138.766
3b	2407.71	3600	3600	3600	3600	3600	3600	3600	3600	3600	3600	3600
3c	597.106	3600	3600	3600	3600	3600	3600	3600	722.732	3600	3600	3600
3d	93.2843	3600	3600	85.9631	3600	3600	3600	3600	146.409	1719.04	865.822	56.4414
3h	98.1373	1673.3	3600	933.88	3600	3600	3600	3600	92.3118	795.069	3600	1655.06
3i	74.1474	3600	1828.97	100.195	3600	3600	3600	3600	72.0747	1224.45	1470.18	191.161
3j	560.258	3600	3600	3600	3600	3600	3600	3600	798.623	3600	3600	3600
3k	3600	3600	3600	2690.33	3600	3600	3600	3600	3600	3600	3600	3600
3l	2723.65	3600	3600	3600	3600	3600	3600	3600	3600	3600	3600	3600

Table A.5: Response Request 2

Architecture	Z3 Mo WS	Z3 Me WS	Z3 Me C	Z3 Mo C	C4 Mo WS	C4 Me WS	C4 Me C	C4 Mo C	Y Mo WS	Y Me WS	Y Me C	Y Mo C
2a	0.1263	0.1521	0.1389	0.1278	0.2067	0.2141	0.1929	0.1999	0.1269	0.1164	0.1118	0.1062
2a - Bound 2	0.1317	0.1364	0.1383	0.1331	0.2016	0.2227	0.194	0.2103	0.1089	0.1049	0.1149	0.1087
2b	0.128	0.1362	0.1434	0.1303	0.1714	0.1978	0.206	0.2375	0.1136	0.1138	0.1049	0.112
2b - Bound 2	0.1345	0.1404	0.1461	0.1304	0.2219	0.1907	0.2001	0.1672	0.1115	0.1117	0.1142	0.1101

Table A.6: Response Request 3

Architecture	Z3 Mo WS	Z3 Me WS	Z3 Me C	Z3 Mo C	C4 Mo WS	C4 Me WS	C4 Me C	C4 Mo C	Y Mo WS	Y Me WS	Y Me C	Y Mo C
3a	0.3359	0.4127	0.4622	0.3774	3.6473	2.4243	1.249	1.8401	0.1707	0.1735	0.1669	0.1509
3b	0.2547	0.4303	0.453	0.3122	1.6014	2.5177	1.4668	2.6185	0.1475	0.1553	0.1727	0.1575
3c	0.2959	0.4249	0.4825	0.3116	3.5418	2.7685	1.3293	2.4744	0.1498	0.1603	0.1663	0.1517
3d	0.2789	0.4582	0.4751	0.2756	2.5759	2.3992	1.1859	2.5649	0.1549	0.1623	0.1516	0.1458
3e	0.2619	0.5476	0.4349	0.267	1.4312	2.7297	1.3639	1.4778	0.1388	0.1507	0.1613	0.1503
3f	0.2986	0.4871	0.5404	0.3249	3.1767	2.3027	1.2931	2.6495	0.1497	0.1552	0.1535	0.1423
3g	0.2703	0.4008	0.51	0.3177	2.7952	2.5986	1.1955	1.4029	0.142	0.1538	0.1539	0.1465
3h	0.3101	0.4159	0.4893	0.271	1.9148	2.379	1.3519	1.476	0.141	0.1647	0.1566	0.1528
3i	0.3047	0.4151	0.4444	0.2753	2.3206	2.436	1.2847	2.8205	0.1669	0.1717	0.1598	0.1489
3j	0.2492	0.5276	0.4816	0.3012	2.623	2.6778	1.3849	1.7342	0.1428	0.1562	0.159	0.1451
3k	0.2895	0.4198	0.5864	0.288	2.3321	2.1232	1.5195	1.3819	0.1429	0.1482	0.1486	0.1501
3l	0.2428	0.5866	0.5694	0.3407	3.9271	3.0901	1.3354	1.697	0.1488	0.1533	0.1539	0.1455

Table A.7: Response Request 4

Architecture	Z3 Mo WS	Z3 Me WS	Z3 Me C	Z3 Mo C	C4 Mo WS	C4 Me WS	C4 Me C	C4 Mo C	Y Mo WS	Y Me WS	Y Me C	Y Mo C
4a1	2.2696	15.1618	20.8889	18.0806	36.2413	3600	29.223	47.0483	0.4121	0.5645	2.4099	1.1357
4a4	9.3356	16.2604	30.6486	26.0005	74.9408	106.899	57.5583	55.82	1.234	1.4474	1.9214	0.9711

Table A.8: Response Request 5

Architecture	Z3 Mo WS	Z3 Me WS	Z3 Me C	Z3 Mo C	C4 Mo WS	C4 Me WS	C4 Me C	C4 Mo C	Y Mo WS	Y Me WS	Y Me C	Y Mo C
5a1	91.2792	1987.4	1418.65	1822.59	3600	3600	3600	3600	3.2087	11.4354	124.802	45.4882
5a1 - TO7200s	64.0529	806.437	2147.39	1266.96	7200	7200	7200	2179.88	3.5751	8.8085	134.144	43.8551
5a5	168.01	620.815	3600	3600	3600	3600	3600	1035.37	37.7221	45.4034	221.258	77.5826

Table A.9: Response Request 6

Architecture	Z3 Mo WS	Z3 Me WS	Z3 Me C	Z3 Mo C	C4 Mo WS	C4 Me WS	C4 Me C	C4 Mo C	Y Mo WS	Y Me WS	Y Me C	Y Mo C
6a1	3600	3600	3600	3600	3600	3600	3600	3600	67.1348	195.649	3600	2290.28
6a1 - TO7200	7200	7200	7200	7200	7200	7200	7200	7200	67.5934	123	3292.22	
6a6	3600	3600	3600	3600	3600	3600	3600	3600	2797.65	1801.28	3600	3600

Table A.10: Response Request 7

Architecture	Z3 Mo WS	Z3 Me WS	Z3 Me C	Z3 Mo C	C4 Mo WS	C4 Me WS	C4 Me C	C4 Mo C	Y Mo WS	Y Me WS	Y Me C	Y Mo C
7a1	3600	3600	3600	3600	3600	3600	3600	3600	1520.73	3600	3600	3600
7a1 - TO7200	7200	7200	7200	7200	7200	7200	7200	7200	1694.45	3618.21	7200	7200
7a7	3600	3600	3600	3600	3600	3600	3600	3600	3600	3600	3600	3600

Acknowledgements

A very big thanks to Peter Faymonville for answering any questions and helping with any problems I had. Also a big thanks to Christopher Hahn, Pascal Berrang, Jeanette Daum and Johannes Krupp for proof reading. Another big thanks to Curd Becker for helping with installing preliminary programs because the Mac compiler did not want to work. And last but not least, a big thanks to my parents for printing this and handing it in and just for being awesome parents.