Summary

Bounded Synthesis

Steffen Metzger

June 26, 2008

1 Introduction

The general synthesis problem is to construct an implementation given some temporal logic specification. On distributed architectures however, the synthesis problem is undecidable in general. Additionally, even on simple architectures without an *information fork* the synthesis problem is of nonelementary complexity. Bounded Synthesis takes a more practical view of the synthesis problem. By introducing bounds on the size of individual processes considered for an implementation as well as an overall bound on the size of the whole implementation, the solution search space is reduced. Instead of searching for all possible implementations only a fraction that satisfies the bounds are considered. This can be helpfull in different ways. First of all real world restrictions may be considered when searching for a solution, such that only implementations are synthesized that satisfy the bounds dictated by the physical world (e.g. for applications on mobile devices efficiency is crucial). By restricting ourselves to a subset of all possible implementations, the resulting description of the solutions can be quite compact. Finally, in general undecidable problems can be tackled by iteratively increasing the bounds and thereby searching for a minimal solution.

2 Overview

Given an architecture and a specification, the bounded synthesis problem is to find an (distributed) implementation that satisfies the specification and the bounds on the number of states (for each process as well as for the overall implementation). In the following a solution for this problem based on SAT constraint solving is presented. The outline to transform a bounded synthesis problem into a constraint system is as follows:

1. Transform the specification given in LTL into a universal co-Büchi automaton
2. Characterize the existance of an implementation satisfying the specification and some bounds by the existence of an annotation on the run-graph nodes
3. Synthesize by solving a constraint system that represents the properties of this annotation on the given specification

Given any LTL specification φ, first we obtain $\neg\varphi$. Now, this can be turned into a nondeterministic Büchi automaton from which we can then construct a co-Büchi automaton that specifies φ. The details of these transformations in the first step are sufficiently discussed in standard literature. So in the following we will assume that the specification is given as a co-Büchi automaton and inspect the remaining two steps. For a specification given as a co-Büchi automaton, see figure 1.

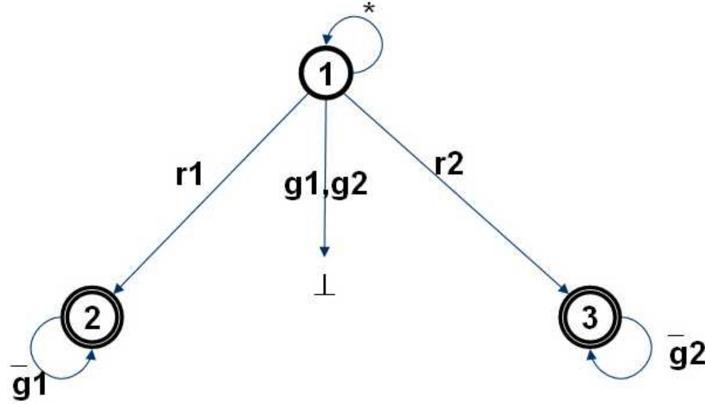

Figure 1: A specification for the pedestrian crossing example or an arbitrary arbiter. The main properties are that any request will finally result in a grant and that it can never happen that both grants are issued at the same time.

3 Preliminaries

Definition 3.1 (Architecture)**.** An *architecture* A is a tuple (P, env, V, I, O), where P is a set of processes including the environment process *env*. V is a set of boolean system variables, $I = \{I_p \subseteq V | p \in P\}$ is a family of input variables while $O = \{O_p \subseteq V | p \in P\}$ is a set of output variables. So if a variable x is in O_p and in $I_{p'}$ this simulates broadcasting the value of x from p to p'.

Definition 3.2 (fully informed)**.** If $O_{env} \subseteq I_p$ holds for all $p \in P \backslash \{env\}$, the architecture is *fully informed*.

Implementations are represented as transition systems.

Definition 3.3 (Transition system)**.** For a given finite set of directions Υ and a finite set of labels Σ, a Σ-labeled Υ-transition system is a tuple (T, t_0, τ, o) where T is a set of states, $t_o \in T$ is an initial state, $\tau : T \times \Upsilon \to T$ is a transition function and $o : T \to \Sigma$ is a labeling function.

So in a distributed implementation each system process is represented by a 2^{O_p}-labeled 2^{I_p}-transition system (T_p, t_p, τ_p, o_p). The complete system is then represented by their composition, the 2^V-labeled $2^{O_{env}}$-transition system (T, t_0, τ, o) where:

- $T = \bigotimes_{p \in P \backslash \{env\}} T_p \times 2^{O_{env}}$

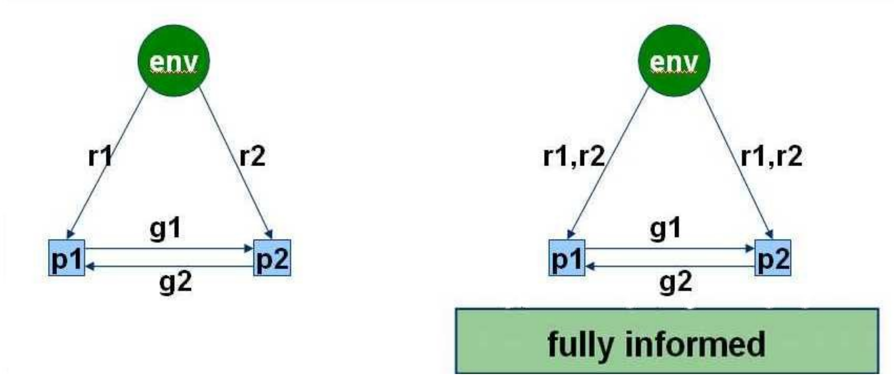

Figure 2: 2 single processes and the composed system - the labels are not shown here for the sake of simiplicity.

- $t_0 = \bigotimes_{p \in P \setminus \{env\}} t_p \times r$ where r is a freely chosen root direction in $2^{O_{env}}$
- τ updates for each p the T_p part in some state t according to τ_p and updates the $2^{O_{env}}$ part of the state with the current output of env.
- o labels each state t with the union of t's $2^{O_{env}}$ part with the results of all o_p applied to the corresponding T_p part of t

Definition 3.4 (input-preserving)**.** We call a transition system *input preserving*, if at all states the label reflects the input received from the environment process.

Be aware that by construction all composed systems are input-preserving and in the following we are only concerned with input-preserving transition systems.

Definition 3.5 (co-Büchi automaton)**.** A co-Büchi automaton is denoted by a tuple $(\Sigma, \Upsilon, Q, q_0, \delta, F)$, where:

- Σ is a set of labels,
- Υ is a set of directions,
- Q is a finite set of states,
- q_0 is an initial state,
- $\delta : Q \times \Sigma \rightarrow \mathbb{B}^+(Q \times \Upsilon)$ is a transition function
- and $F \subseteq Q$ denotes the set of rejecting states.

A *run* on a co-Büchi automaton *is accepting* if and only if any rejecting state appears only finitely often (or does not appear at all).

Definition 3.6 (universal co-Büchi automaton)**.** We call a co-Büchi automaton *universal* iff for all $q \in Q$ and $x \in \Sigma$, $\delta(q, x)$ is a conjunction.

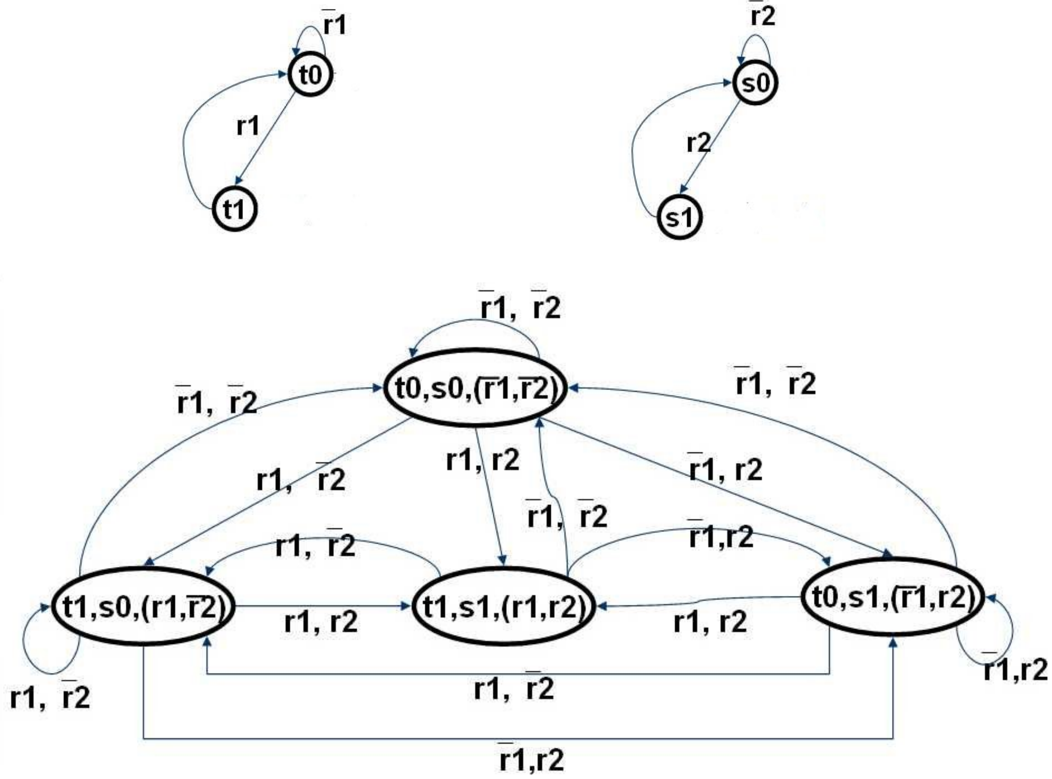

Figure 3: 2 single processes and the composed system

Definition 3.7 (run graph)**.** A run graph of a co-Büchi automaton $B = (\Sigma, \Upsilon, Q, q_0, \delta, F)$ on a Σ-labeled Υ-transition system (T, t_0, τ, o) is a minimal directed graph $G = (V, E)$ that satisfies the following constraints:

- $V \subseteq Q \times T$
- $(q_0, t_0) \in V$
- for all $q, t \in V$ the set $\{(q', v) \in Q \times \Upsilon | ((q, t), (q', \tau(t, v))) \in E\}$ satisfies $\delta(q, o(t))$.

A run graph is *accepting* if in every infinite path $(q_1, t_1), (q_2, t_2), ...$ in the run graph there is a position i such that no node (q_j, t_j) where q_j is rejecting and $j > i$ exists (or in other words, there is a finite number of appearances of rejecting states in every possible path).

4 Annotations

In order to decide if an accepting run graph exists, *annotations* on the run graph vertices are introduced. Annotations provide an ordering on the states of the run graph enforcing rejecting nodes to be strictly *greater* than their predecessors. If an annotation is valid then on every path there is a maximal rejecting state such that none can follow after this one (because it would have to be annotated higher than any predecessor). Finally this will give us the theorem that if and only if a *valid* annotation exists, then there is an accepting run graph.

Definition 4.1 (Annotation)**.** An annotation of a transition system (T, t_0, τ, o) on a universal co-Büchi automaton $(\Sigma, \Upsilon, Q, q_0, \delta, F)$ is a function $\lambda : Q \times T \to \{_\} \cup \mathbb{N}$.
We call an annotation *c-bounded* iff there is a maximal value c such that it holds for all $q \in Q, t \in T$ $\lambda(q,t) \leq c \vee \lambda(q,t) = _$.

Definition 4.2 (valid annotation)**.** An annotation is called *valid* iff it satisfies the following conditions:

- $\lambda(q_0, t_0) \neq _$
- for all $q \in Q, t \in T$ it holds:
  From $\lambda(q,t) = n \in \mathbb{N}$ follows $\lambda(q', \tau(t,v)) \geq_{q'} n$
  where $(q', v) \in \delta(q, o(t))$
  and $\geq_{q'}$ is $>$ in case $q' \in F$ while $\geq_{q'}$ is $\geq$ otherwise

So the two conditions simply say that the annotation value along a path starting from the initial node does not decrease. Additionally whenever some rejecting state is part of some node on a path from the root node, this node has to have a strictly greater annotation value than its predecessor(s). Finally this means any node reachable from the initial node is assigned a natural number as annotation value.

Theorem 4.1. *A finite state Σ-labeled Υ-transition (T, t_0, τ, o) is accepted by a universal co-Büchi automaton $(\Sigma, \Upsilon, Q, q_0, \delta, F)$ iff it has a valid $(|T| \cdot |F|)$-bounded annotation.*

5 Constraint system

The previous theorem states that, to decide whether there is an accepting run graph it is sufficient to decide whether there is a valid annotation. Thereby given a specification as a co-Büchi automaton $(\Sigma, \Upsilon, Q, q_0, \delta, F)$, the synthesis problem can be described as a constraint system specifying the existence of a valid annotation λ on some Σ-labeled Υ-transition (T, t_0, τ, o). To specify the constraint system the following predicates are introduced:

- $\tau_v(t) = \tau(t, v)$
- for each $\alpha \in V$ there is a unary predicate α such that $\alpha(t) \Leftrightarrow \alpha \in o(t)$
- $\forall q \in Q. \lambda_q^{\#}(t) = \lambda(q,t)$ iff $\lambda(q,t) \in \mathbb{N}$
- $\forall q \in Q. \lambda_q^{\mathbb{B}}(t)$ is true iff $\lambda(q,t) \in (N)$

5.1 Fully informed architectures

First we consider fully informed architectures. In this case the constraint system can be specified by the following constraints:

1. $\forall t \in T. \forall \alpha \in V. \forall v \subseteq \Upsilon.$

   $\alpha(\tau_v(t))$ iff $\alpha \in v$
   $\neg\alpha(\tau_v(t))$ otherwise

2. $\lambda_{q_0}^{\mathbb{B}}(t_0)$

3. $\forall t \in T.\forall q \in Q.\lambda_q^{\mathbb{B}}(t) \wedge (q', v) \in \delta(q, o(t)) \rightarrow \lambda_{q'}^{\mathbb{B}}(\tau_v(t)) \wedge \lambda_{q'}^{\#}(\tau_v(t)) \geq_q \lambda_q^{\#}(t)$
   where:

   $\geq_q \equiv >$ iff $q' \in F$
   $\geq_q \equiv \geq$ otherwise

The first constraint ensures that the transition system is input-preserving. The other two conditions represent the annotation properties. The second condition ensures that the initial state is annotated with a natural number while the third constraint ensures that the annotation does not decrease along a path and strictly increases whenever a rejecting node is visited. An example is given in figure 4.

5.2 Distributed synthesis

To solve the distributed synthesis problem for arbitrary distributed architectures, we have to take into account that single processes may have only partial knowledge about the environment and act independently of each other. So instead of a composed/single system we have to specify the family of transition systems $\{T_p, t_{0_p}, \tau_p, o_p\}$. Hence a unary function d_p is introduced which decomposes a global state into a particular state of a process p. So it maps from some $t \in T_A$ to the p-component $t_p \in T_p$. To adapt the constraint system, some additional properties need to be specified. First a process has to act only accordingly to its own knowledge, e.g. it has to act equally on any two histories it cannot distinguish. Secondly any output of a certain process may only depend on the state of this process, so any occurence of $\alpha(t)$ is replaced by $\alpha(d_p(t))$. This results in the following constraints:

1. $\forall t.d_p(\tau_v(t)) = d_p(\tau_{v'}(t))$ for all $v, v' \subseteq O_{env}$ where $v \cap I_p = v' \cap I_p$ (for p the input v and v' looks equal)
2. $\forall t, u \in T.d_p(t) = d_p(u) \bigwedge_{\alpha \in I_p \setminus O_{env}} (\alpha(d_{p_\alpha}(t)) \leftrightarrow \alpha(d_{p_\alpha}(u))) \rightarrow d_p(\tau_v(t)) = d_p(\tau_v(u))$ for all $v \in O_{env} \cap I_p$. (p_a denotes the process responsible for output variable α ($\alpha \in O_{p_\alpha}$))

References

[SF] Sven Schewe and Bernd Finkbeiner, *Bounded Synthesis*.

[SF07] ______, *Smt-based synthesis of distributed systems*.

1. $\forall t. r_1(\tau_{r_1 r_2}(t)) \wedge r_2(\tau_{r_1 r_2}(t)) \wedge r_1(\tau_{r_1 \overline{r_2}}(t)) \wedge \neg r_2(\tau_{r_1 \overline{r_2}}(t)) \wedge \neg r_1(\tau_{\overline{r_1 r_2}}(t)) \wedge \neg r_2(\tau_{\overline{r_1 r_2}}(t)) \wedge \neg r_1(\tau_{\overline{r_1} r_2}(t)) \wedge r_2(\tau_{\overline{r_1} r_2}(t))$

2. $\lambda_1^{\mathbb{B}}(t_0) \wedge \neg r_1(t_0) \wedge \neg r_2(t_0)$

3. $\forall t. \lambda_1^{\mathbb{B}}(t) \rightarrow$
$$\begin{aligned} & \lambda_1^{\mathbb{B}}(\tau_{\overline{r_1},\overline{r_2}}(t)) \wedge \lambda_1^{\#}(\tau_{\overline{r_1},\overline{r_2}}(t)) \geq \lambda_1^{\#}(t) \\ \wedge\ & \lambda_1^{\mathbb{B}}(\tau_{\overline{r_1},r_2}(t)) \wedge \lambda_1^{\#}(\tau_{\overline{r_1},r_2}(t)) \geq \lambda_1^{\#}(t) \\ \wedge\ & \lambda_1^{\mathbb{B}}(\tau_{r_1,\overline{r_2}}(t)) \wedge \lambda_1^{\#}(\tau_{r_1,\overline{r_2}}(t)) \geq \lambda_1^{\#}(t) \\ \wedge\ & \lambda_1^{\mathbb{B}}(\tau_{r_1,r_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1,r_2}(t)) \geq \lambda_1^{\#}(t) \end{aligned}$$

4. $\forall t. \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(t) \vee \neg g_2(t)$

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

6. $\forall t. \lambda_1^{\mathbb{B}}(t) \wedge r_2(t) \rightarrow$
$$\begin{aligned} & \lambda_3^{\mathbb{B}}(\tau_{\overline{r_1},\overline{r_2}}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r_1},\overline{r_2}}(t)) > \lambda_1^{\#}(t) \\ \wedge\ & \lambda_3^{\mathbb{B}}(\tau_{\overline{r_1},r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r_1},r_2}(t)) > \lambda_1^{\#}(t) \\ \wedge\ & \lambda_3^{\mathbb{B}}(\tau_{r_1,\overline{r_2}}(t)) \wedge \lambda_3^{\#}(\tau_{r_1,\overline{r_2}}(t)) > \lambda_1^{\#}(t) \\ \wedge\ & \lambda_3^{\mathbb{B}}(\tau_{r_1,r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1,r_2}(t)) > \lambda_1^{\#}(t) \end{aligned}$$

7. $\forall t. \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(t) \rightarrow$
$$\begin{aligned} & \lambda_2^{\mathbb{B}}(\tau_{\overline{r_1},\overline{r_2}}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r_1},\overline{r_2}}(t)) > \lambda_2^{\#}(t) \\ \wedge\ & \lambda_2^{\mathbb{B}}(\tau_{\overline{r_1},r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r_1},r_2}(t)) > \lambda_2^{\#}(t) \\ \wedge\ & \lambda_2^{\mathbb{B}}(\tau_{r_1,\overline{r_2}}(t)) \wedge \lambda_2^{\#}(\tau_{r_1,\overline{r_2}}(t)) > \lambda_2^{\#}(t) \\ \wedge\ & \lambda_2^{\mathbb{B}}(\tau_{r_1,r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1,r_2}(t)) > \lambda_2^{\#}(t) \end{aligned}$$

8. $\forall t. \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(t) \rightarrow$
$$\begin{aligned} & \lambda_3^{\mathbb{B}}(\tau_{\overline{r_1},\overline{r_2}}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r_1},\overline{r_2}}(t)) > \lambda_2^{\#}(t) \\ \wedge\ & \lambda_3^{\mathbb{B}}(\tau_{\overline{r_1},r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r_1},r_2}(t)) > \lambda_3^{\#}(t) \\ \wedge\ & \lambda_3^{\mathbb{B}}(\tau_{r_1,\overline{r_2}}(t)) \wedge \lambda_3^{\#}(\tau_{r_1,\overline{r_2}}(t)) > \lambda_3^{\#}(t) \\ \wedge\ & \lambda_3^{\mathbb{B}}(\tau_{r_1,r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1,r_2}(t)) > \lambda_3^{\#}(t) \end{aligned}$$

Figure 4: Constraint system example for a fully informed architecture. The constraint system represents the specification shown in figure 1 on the fully informed architecture shown in 2.

4. $\forall t.\lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(d_1(t)) \vee \neg g_2(d_2(t))$

7. $\forall t.\lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(d_1(t)) \rightarrow$
$$\begin{aligned} & \lambda_2^{\mathbb{B}}(\tau_{\overline{r_1},\overline{r_2}}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r_1},\overline{r_2}}(t)) > \lambda_2^{\#}(t) \\ \wedge\ & \lambda_2^{\mathbb{B}}(\tau_{\overline{r_1},r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r_1},r_2}(t)) > \lambda_2^{\#}(t) \\ \wedge\ & \lambda_2^{\mathbb{B}}(\tau_{r_1,\overline{r_2}}(t)) \wedge \lambda_2^{\#}(\tau_{r_1,\overline{r_2}}(t)) > \lambda_2^{\#}(t) \\ \wedge\ & \lambda_2^{\mathbb{B}}(\tau_{r_1,r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1,r_2}(t)) > \lambda_2^{\#}(t) \end{aligned}$$

8. $\forall t.\lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(d_2(t)) \rightarrow$
$$\begin{aligned} & \lambda_3^{\mathbb{B}}(\tau_{\overline{r_1},\overline{r_2}}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r_1},\overline{r_2}}(t)) > \lambda_2^{\#}(t) \\ \wedge\ & \lambda_3^{\mathbb{B}}(\tau_{\overline{r_1},r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r_1},r_2}(t)) > \lambda_3^{\#}(t) \\ \wedge\ & \lambda_3^{\mathbb{B}}(\tau_{r_1,\overline{r_2}}(t)) \wedge \lambda_3^{\#}(\tau_{r_1,\overline{r_2}}(t)) > \lambda_3^{\#}(t) \\ \wedge\ & \lambda_3^{\mathbb{B}}(\tau_{r_1,r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1,r_2}(t)) > \lambda_3^{\#}(t) \end{aligned}$$

9. $\forall t.d_1(\tau_{r_1,r_2}(t)) = d_1(\tau_{r_1,\overline{r_2}}(t)) \wedge d_1(\tau_{\overline{r_1},r_2}(t)) = d_1(\tau_{\overline{r_1},\overline{r_2}}(t)) \wedge d_2(\tau_{r_1,r_2}(t)) = d_2(\tau_{\overline{r_1},r_2}(t)) \wedge d_2(\tau_{\overline{r_1},\overline{r_2}}(t)) = d_2(\tau_{r_1,\overline{r_2}}(t))$

10. $\forall t,u.d_1(t) = d_1(u) \wedge (g_2(d_2(t)) \leftrightarrow g_2(d_2(u))) \rightarrow d_1(\tau_{r_1,r_2}(t)) = d_1(\tau_{r_1,r_2}(u)) \wedge d_1(\tau_{\overline{r_1},r_2}(t)) = d_1(\tau_{\overline{r_1},r_2}(u))$

11. $\forall t,u.d_2(t) = d_2(u) \wedge (g_1(d_1(t)) \leftrightarrow g_1(d_1(u))) \rightarrow d_2(\tau_{r_1,r_2}(t)) = d_2(\tau_{r_1,r_2}(u)) \wedge d_2(\tau_{r_1,\overline{r_2}}(t)) = d_2(\tau_{r_1,\overline{r_2}}(u))$

Figure 5: Constraint system example for a distributed synthesis. The figure shows the constraints that have to be modified or added with respect to the example in figure 4. The resulting constraint system represents the specification shown in figure 1 on the distributed architecture shown in figure 2 (the left architecture).