Distributed Reactive Synthesis

B. Tech Project Report Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Bachelor of Technology

by

Mesharya M Choudhary and Bhuvan Aggarwal

(190101053,190101025)

under the guidance of

Prof. Purandar Bhaduri

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, GUWAHATI

GUWAHATI - 781039, ASSAM

Abstract

Distributed Reactive Synthesis deals with the design and implementation of multi-component systems working in an environment. These components interact with one another and the environment by the means of shared variables. Each component has its own goal which it tries to fulfill. It finds application in areas of systems engineering particularly synthesis of controllers for robots, hardware circuits and communication protocols. In our work we look at two approaches - assume guarantee synthesis and rational verification and how they can be used to solve the synthesis problem.

Contents

Chapter 1

Introduction

Distributed Reactive Systems comprise of multiple components each with their own specification and communicating via a set of shared variables. The problem of synthesizing distributed reactive systems has been a topic of study for a long period of time finding applications in several areas of systems engineering such as synthesizing hardware circuits, device drivers, controllers for robots and communication protocols. The aim of synthesis is to construct systems automatically given the logical specification. Since it eliminates the need for manual work, development of efficient synthesis methods can transform the development process of reactive systems. Rosner and Pnueli discovered that the synthesis problem is undecidable for majority of the distributed systems [3]. Thus, all methods are necessarily incomplete.

We consider a setting which is similar to that in [2]. We consider the distributed reactive system as a set of components each of which controls a set of variables some of which are local and others shared. Each shared variable can be written by only one component but can be read by the others. The components communicate via these shared variables. Thus, each component only knows the valuations for its own local and shared variables which means that it has only a partial knowledge about the overall state of the system. The components have local specifications defined by a set of sequences over valuations of its variables. The objective is to synthesize local controllers in the form of strategies for each component to make decisions based on the partial knowledge of the

overall system in such a way that the local specification of each of the components is met.

1.1 Aim

Our aim is to try out two methods - Rational Verification [1] and Assume Guarantee Synthesis [2] for distributed reactive synthesis. In assume-guarantee synthesis each component assumes certain behaviour of its environment (comprised of other components and global environment, if any) and apart from it provides a guarantee on its own behaviour. Given we are able to find assume-guarantee pairs for each of the components such that if they satisfy their assume-guarantee specification, each of them can satisfy their specification and the guarantees imply that the assumptions hold it can be shown that the local specification of the components is also met. The method provides an algorithm which explores the possibilities of assume-guarantee pairs in search of a set of pairs which allow all the systems to satisfy their specifications. It iteratively refines assumptions and guarantees of the components so as to find the minimally restrictive assumptions on the environment until convergence. If the algorithm terminates and finds a set of compatible assumptions and guarantees we can find the strategies from distributed controllers, however, since the problem is undecidable, there is no guarantee of termination.

The alternative approach to distributed synthesis is to leverage the rational behaviour of the components using rational verification and apply it to solve the distributed synthesis problem. Game theory allows one to think distributed reactive systems as a set of participants in a game each acting rationally in pursuit of its goals. Rational verification establishes whether given a temporal logic formula ϕ , there exists a game-theoretic equilibria satisfying it or if it is satisfied for all game-theoretic equilibria i.e. whether the system will behave according to ϕ if the participants act rationally. If say a Nash equilibrium exists then none of the components have any incentive for deviating from the strategy unilaterally and such strategy profile can be used to find the behaviors of controllers.

1.2 Setting of the problem

We give the formal definition of the distributed synthesis problem following [2]. Given the alphabet Σ we represent the set of finite words over this alphabet as Σ^* and the set of infinite words over this alphabet as Σ^ω respectively and define $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. We define $\Sigma^+ = \Sigma^* \backslash \{\epsilon\}$ where ϵ is the empty word, as the set of all nonempty words over the alphabet Σ. We say $u \in \Sigma^*$ is a prefix of $v \in \Sigma^\infty$, represented by $u \leqslant v$ if there exists a $w \in \Sigma^\infty$ such that $uw = v$. The prefix relation can be stated for language $L \subseteq \Sigma^\infty$ as $pref(L) = \{u \in \Sigma^* \mid \exists v \in L, u \leqslant v\}$.

Given a set of variables X a *valuation* of X is a mapping of each variable present in the set X to a value in its domain i.e. for some $X = \{x_1, \ldots, x_n\}$ we define the valuation g over X as an $(d_1, \ldots, d_n)$ where $g(x_i) = d_i$ for $1 \leqslant i \leqslant n$. We use the notation $\mathcal{V}(X)$ to denote the set of all valuations over X. Given two disjoint sets of variables X and Y we define $\mathcal{V}(X \cup Y) = \mathcal{V}(X) \times \mathcal{V}(Y)$. Given a valuation $v \in \mathcal{V}(X) \times \mathcal{V}(Y)$, we denote the restrictions of v to the subsets X and Y by $v \restriction_X$ and $v \restriction_Y$ respectively. For $v_X \in \mathcal{V}(X)$ and $v_Y \in \mathcal{V}(Y)$ where X and Y are disjoint, we denote the valuation of $X \cup Y$ that maps variables in X and Y according to v_X and v_Y respectively as $v_X \oplus v_Y$. Given sets of variables Y and X where $Y \subset X$ and a word $v = v_0 v_1 \ldots \in \mathcal{V}(X)^\infty$ the restriction of the sequence of valuations of X over the set Y is given by $(v_0 \restriction_Y)(v_1 \restriction_Y)\ldots$ which is denoted by $proj_Y(v)$.

A *system* is denoted as $S = (X, v_{\text{in}}, U, W, \delta, Y, h)$ where X denotes a finite set of *state variables*, $v_{\text{in}} \in \mathcal{V}(X)$ denotes the *initial state*, U denotes a finite set of *input actions*, W denotes a finite set of *external variables*, $\delta : \mathcal{V}(X) \times \mathcal{V}(W) \times U \to \mathcal{V}(X)$ is the *transition function*, Y denotes a finite set of *output variables*, and $h : V(X) \to \mathcal{V}(Y)$ is the *output labelling function*. Our assumption is that sets of variables X, Y, and W are pairwise disjoint.

We define the *run* of a system S starting from a starting state v_0 as a sequence $\rho = v_0 \xrightarrow{w_0,u_0} v_1 \xrightarrow{w_1,u_1} v_2 \xrightarrow{w_2,u_2} \ldots$, such that $v_i \in \mathcal{V}(X)$, $w_i \in \mathcal{V}(W)$, $u_i \in U$ and $\delta(v_i, w_i, u_i) = v_{i+1}$ for all $i \geqslant 0$. Our interest particularly is in the runs starting at $v_0 = v_{\text{in}}$. We define the *output* of the run as the sequence $h(v_0)h(v_1)\ldots$. Given the initial state to be v_0 we determine at each step i the

next state v_{i+1} with the help of the transition function δ whose inputs are the current state v_i, the input w_i from the environment and the current input action u_i. Given the run ρ, the sequence of states $v_0v_1\ldots$ is denoted using $proj_X(\rho)$, the sequence of external inputs $w_0w_1\ldots$ using $proj_W(\rho)$, and the sequence of outputs $h(v_0)h(v_1)\ldots$ using $proj_Y(\rho)$. Given a subset $V \subseteq W$ over the set of external variables the notation $proj_V(\rho)$ denotes the sequence $(w_0 \restriction V)(w_1 \restriction V)\ldots$.

Let S_1 and S_2 be two systems such that X_1, X_2, Y_1, Y_2 are all pairwise disjoint where X_1 is the state variable and Y_1 is the output variable for S_1 and similarly for X_2, Y_2. We define the *parallel composition* of S_1 and S_2, $S_1 \parallel S_2$ as the system $(X, x_{\text{in}}, U_1 \times U_2, W, \delta, Y_1 \cup Y_2, h)$ where $X = X_1 \uplus X_2$, $x_{\text{in}} = x_{\text{in1}} \oplus x_{\text{in2}}$, $W = W_1 \cup W_2 \backslash (Y_1 \cup Y_2)$, $\delta : \mathcal{V}(X) \times \mathcal{V}(W) \times U_1 \times U_2 \to \mathcal{V}(X)$, and $h : \mathcal{V}(X) \to \mathcal{V}(Y_1) \times \mathcal{V}(Y_2)$ where transition function is given by $\delta(v, w, (u_1, u_2)) = v'$ iff $\delta_1(v \restriction X_1, (w \oplus h_2(v \restriction X_2)) \restriction W_1, u_1) = v' \restriction X_1$ and $\delta_2(v \restriction X_2, (w \oplus h_1(v \restriction X_1)) \restriction W_2, u_2) = v' \restriction X_2$ and the output function is given by $h(v) = h_1(v \restriction X_1) \oplus h_2(v \restriction X_2)$. A run ρ of the system $S_1 \parallel S_2$ is a sequence

$$v_0 \xrightarrow{w_0,(u_{10},u_{20})} v_1 \xrightarrow{w_1,(u_{11},u_{21})} v_2 \longrightarrow \ldots$$

where $v_i \in \mathcal{V}(X)$, $w_i \in \mathcal{V}(W)$, $u_{1i} \in U_1$, $u_{2i} \in U_2$, $v_0 = v_{\text{in}}$ and $\delta(v_i, w_i, (u_{1i}, u_{2i})) = v_{i+1}$ for all $i \geqslant 0$. The intuition of the parallel composition is that both the systems run parallely changing state synchronously and the outputs of the other system and the environment is the input for each system and based on the valuation of the external variables it picks an input action and updates its own state.

A *specification* for a system is defined by a set of sequences $\varphi \subseteq (\mathcal{V}(X) \times \mathcal{V}(W))^\omega$ i.e. an infinite word over the valuations of all variables of the system and a run $v_0 \xrightarrow{w_0,u_0} v_1 \xrightarrow{w_1,u_1} v_2 \xrightarrow{w_2,u_2} \ldots$ is said to *satisfy* the specification if $(v_0, w_0)(v_1, w_1), \ldots \in \varphi$. A *local specification* is defined as a set $\varphi \in \mathcal{V}(X)^\omega$ i.e. an infinite word over the variables controlled by that particular component. The run described above is said to satisfy the local specification φ if it satisfies the specification $\{(v_0, w_0), (v_1, w_1), \ldots \in (\mathcal{V}(X) \times \mathcal{V}(W))^\omega \mid v_0v_1\ldots\}$ given arbitrary sequences of valuations for the external variables.

A *strategy* for the system is a mapping from the sequence of valuations over the variables to input action $\sigma : (\mathcal{V}(X) \times \mathcal{V}(W))^+ \rightarrow U$ that tells the system what input action to perform given the current history of the system. Similarly, an *environment strategy* is a mapping from the sequence of the valuations of all variables and the current state of the components to the valuation of output variables $\pi : (\mathcal{V}(X) \times \mathcal{V}(W))^* \times \mathcal{V}(X) \rightarrow \mathcal{V}(W)$. A run $v_0 \xrightarrow{w_0,u_0} v_1 \xrightarrow{w_1,u_1} v_2 \xrightarrow{w_2,u_2} \ldots$ is said to be consistent with the system strategy σ if for each $i \in \mathbb{N}$, we have $u_i = \sigma((v_0, w_)), (v_1, w_1), \ldots ((v_i, w_i))$ and similarly consistency with strategy of environment is also defined. The *unique* run consistent with σ and π is denoted by $\rho(\sigma, \pi)$.

A system S *can realize* a specification φ if there is a system strategy σ such that all runs ρ consistent with σ satisfy $proj_{X \times Y}(\rho) \in \varphi$ and such a strategy is called the *realization* or *winning strategy* for φ. Thus, realizability can be modelled as a game between the system and the environment where the specification is realizable by the system S if S has a winning strategy, *i.e.*, the resulting run belongs to φ no matter what strategy the enviroment uses.

In distributed synthesis each system S_i can use strategies that depend only on the *current state* which is known as memoryless strategies. A *memoryless strategy* of system S_i is a map $\mathcal{V}(X_i) \times \mathcal{V}(W_i) \rightarrow U_i$. Formally, the *distributed synthesis* problem $(S_1, \varphi_1, S_2, \varphi_2)$ for the composition $S_1 \parallel S_2$ where thw local specifications are $\varphi_1 \subseteq \mathcal{V}(X_1)^\omega$ and $\varphi_2 \subseteq \mathcal{V}(X_2)^\omega$ is to determine the existence of strategies $\sigma_1 : \mathcal{V}(X_1) \times \mathcal{V}(W_1) \rightarrow U_1$ for S_1 and $\sigma_2 : \mathcal{V}(X_2) \times \mathcal{V}(W_2) \rightarrow U_2$ for S_2 such that for all environment strategies $\pi : (\mathcal{V}(X) \times \mathcal{V}(W))^* \times \mathcal{V}(X) \rightarrow \mathcal{V}(W)$ we have that $proj_{X_i}(\rho(\sigma_1, \sigma_2, \pi) \in \varphi_i$ for $i \in \{1, 2\}$. In such a case, we say that $S_1 \parallel S_2$ can *realize* the distributed synthesis problem. It is possible that S_1 and S_2 individually fail to realize their local specifications φ_1 and φ_2 respectively but $S_1 \parallel S_2$ realizes the distributed synthesis problem. This is because S_1 and S_2 can cooperate and choose strategies that are mutually beneficial.

Chapter 2

Assume-Guarantee Synthesis

This section explains the approach of Assume-Guarantee synthesis taken from Majumdar et al [2]. The components, in general, cannot fulfill their specification by themselves. Thus, a component would require that other components make certain promises on their behaviour. These promises provided by other components together make the *assumption* for our component. In return, the component itself would make certain promises to other components to help them fulfill their goals. This promise is called a *guarantee*.

The assume-guarantee approach is a modular one. This is in contrast to centralized synthesis, where a central server can compute the strategy and then instruct the components to execute certain behaviour. The central server would have access to all the state information and goals of the individual components, which it would use in computing the strategy. However in this setting, there is no central server, and the components have only a partial view of the other components. The components cannot view the state of other components, they only communicate through shared variables. This prevents one of the component acting as a central server. In our setting, a system consists of:

1. *state variables* with full read and write access
2. *external variables* which are read-only and provide a view of the environment
3. *output variables* which the component shares with the outside world. Each states is mapped

to an output variables. Multiple states may map to same output variable.

4. *input actions* which are used to determine the transition between the states

2.1 Example - Distributed Shared Bus

We take example of distributed shared bus to explain the setting of the problem. Consider a synchronous system comprising of two components C_0 and C_1. These components want to send a single data packet on a shared bus. Sending a data packet requires one unit time. Components can send packet at any time they want. But if both the components send data packets at the same time step, both packets are lost. Each component has a deadline of 4 time steps to send the packet.
When discussing with respect to C_0, the environment would comprise of only C_1. Same would follow for C_1. We describe the component C_0 as follows:-

state var s $\in$ *idle*, *writing*, *done*,

$t \in$ 1,2,3,4

external var $env \in idle, busy$

output var $out \in idle, busy$

input action $U \in wait, wr$

init $s = idle, t = 4$

transition

s $= idle \xrightarrow{\text{wait}} s' = idle \wedge t' = t - 1$

$s = idle \xrightarrow{\text{wr}} s' = writing \wedge t' = t - 1$

$s = writing \wedge t \geqslant 2 \wedge env = idle \xrightarrow{\text{wr}} s' = done$

$s = writing \wedge t \geqslant 2 \wedge env = busy \xrightarrow{\text{wr}} s' = writing \wedge t' = t - 1$

output $out = busy\ if\ s = writing\ and\ idle\ otherwise$

The initial state is $(idle, 4)$. Different Values of state variables gives us states. Thus, we have a total of $|s| * |t| = 3 * 4 = 12$ states. Input actions *wait* and *wr* are used to determine the next states. These are not visible outside of the component. Intuitively, when $s = idle$, and input action

wr is taken then s changes to *writing* and t reduces by 1. Now, if $s = writing$, $t >= 2$, and the other component is not writing, i.e., $env = idle$ then, s changes to *done*. $s = done$ denotes that the component successfully wrote the packet before the deadline.
Now, the goal of a component is represented in terms of state variables, using linear temporal logic (LTL). The goal is to eventually reach *done*. In terms of LTL, it is $F(s = done)$.

2.2 Assumptions and Guarantees

We realise that if $env = busy$ all the time, then we are not able to send our packet. We require some assumption on external variables. Some of the different assumptions can be:

1. **Assume worst case environment.** An assumption of *true* would mean no restriction on environment at all. This does not help us achieve our specification.

2. **Assume that environment fulfills its own specification.** In our case, this assumption would be, the other component *eventually* does not write to the bus. Knowing this assumption is of little use, the packets could still collide.

3. **Assume that environment is** *idle* **in second time step.** This assumption is strong enough to enable the component to fulfill its specification. This assumption can be written as $A = (busy + idle).idle.(busy + idle)^{\omega}$.

In general, we need a stronger assumption than the specification of the environment. We see with the above example that the assumption is a set of infinite words over the external variable. Below we describe some of the important concepts.

2.2.1 Assume-Guarantee Contracts

Contract. For a component $C = (X, v_{\text{in}}, U, W, \delta, Y, h)$, (A, G), a pair of $safety$ languages is called a contract, where $A \subseteq V^{\omega}$ for some $V \subseteq W$, and $G \subseteq X^{\omega}$. A is called $assumption$, and G is called *guarantee*.

Realizing a contract. A component C *realizes* a contract (A, G), if there exist a strategy, such that for all runs compliant with the strategy, violation of guarantee happens only if it was preceded by violation of assumption. Such a strategy is called *realization* strategy for (A, G).

Realizing a specification under a contract. A component C can *realize* a specification ϕ under a contract (A, G), if there exists a strategy π, which is a *realization* strategy for (A, G), as well as a *realization* strategy for $(A \implies \phi)$.

Compatible contracts. For a system composition $C_0||C_1$, let (A_0, G_0), (A_1, G_1) be the contracts for C_0, C_1 respectively. The contracts are compatible if for both $i \in \{0, 1\}$ the following conditions are met:

i. G_i is strong enough to imply A_{1-i}.

ii. C_i *realizes* (A, G).

Theorem - Assume Guarantee Decomposition. We state an important theorem from Majumdar et al. without proof. If $(C_0, \phi_0, C_1, \phi_1)$ is a distributed synthesis problem. (A_0, G_0) and (A_1, G_1) are the contracts of C_0 and C_1 respectively. $C_0||C_1$ can *realize* distributed synthesis problem if:

i. (A_0, G_0) and (A_1, G_1) are compatible.

ii. For both $i \in \{0, 1\}$, C_i can realize ϕ_i under (A_i, G_i).

Now, in order to solve distributed synthesis problem, our aim is to find the contracts which satisfy the conditions stated in the above theorem. We see about that in the next section.

2.3 Negotiation

Negotiation is a *sound*, *iterative* algorithm which tries to compute a pair of *compatible* contracts.

Let us look how the process of negotiation may look like in the distributed shared bus example described previously. The process starts with assumptions and guarantees which allow all behaviour and then refines them in each iteration. We start the negotiation process with C_0. C_0

sees that it is not able to fulfill its specification on its own. Thus C_0 may come up with an assumption that C_1 is idle in the first two time steps, i.e. $A_0 = (idle_1).(idle_1).(busy_1 + idle_1)^w$. This assumption is *sufficient* for A_0. Now, A_1 checks if it can fulfill its specification as well as guarantee the assumption A_0. It cannot do that. So, it may come up with the assumption $A_1 = (busy_0 + idle_0).(busy_0 + idle_0).(idle_0).(busy_0 + idle_0)^w$. Under current assumptions, both C_0 and C_1 can fulfill their specification and guarantees. So, we stop the process here. Finally, we have $A_0 = G_1 = (idle_1).(idle_1).(busy_1 + idle_1)^w$ and $A_1 = G_0 = (busy_0 + idle_0).(busy_0 + idle_0).(idle_0).(busy_0 + idle_0)^w$.

We see that at each iteration we need to check if a component is able to satisfy the specification and the guarantee under its current assumption. If not we further refine the assumption. Assume a function $FindAssumptions$ which helps us refine the assumptions. It takes input as the current contract, and returns a new assumption. We now describe the broad structure for the algorithm.

Algorithm 1 Negotiation Algorithm

Input: $(C_0, \phi_0, C_1, \phi_1)$
Output: Pair of contracts (A_0, C_0) and (A_1, C_1) or **DoesNotExist**

Initialize $A_0 := Y_1^\omega$, $G_0 := X_0^\omega$, $A_1 := Y_0^\omega$, $G_1 := X_1^\omega$
while true **do**
  if C_i *realizes* ϕ_i under (A_i, G_i) for all $i \in \{0, 1\}$ **then**
    return (A_0, C_0) and (A_1, C_1)
  end if
  if C_i surely cannot *realize* ϕ_i under (A_i, G_i) for any $i \in \{0, 1\}$ **then**
    return DoesNotExist
  end if
  $L_0 := FindAssumption(C_0, (A_0, G_0), \phi_0)$
  $A_0 := A_0 \cap L_0, G_1 := G_1 \cap h_1^{-1}(L_0)$
  $L_1 := FindAssumption(C_1, (A_1, G_1), \phi_1)$
  $A_1 := A_1 \cap L_1, G_0 := G_0 \cap h_0^{-1}(L_1)$
end while

Recall that assumptions and guarantees are a subset of infinite words over appropriate alphabet, thus by taking its intersection with newly calculated assumption L_i further restricts the assumption. Also note that assumptions are not simultaneously updated.

2.3.1 Finding Assumptions

Assumption A calculated by $FindAssumptions$ should be $sufficient$ i.e. the component can *realize* $(A \implies \phi)$ as well as guarantee. In principle, they should also be *maximally permissive*, i.e. they should place minimal restrictions on the environment. A is a *maximally permissive* assumption if for any proper superset A' of A, the component cannot *realize* $(A' \implies \phi)$. Though *maximally permissive sufficient assumption* is useful in certain ways, it have certain limitations. It may take more time to compute, and it may leave a single realization strategy for the component. Thus, $FindAssumptions$ resorts to a computing an under-approximation of maximal assumptions.

Chapter 3

Rational Verification

This section explains the approach of Rational Verification taken from Abate et al [1]. Rational Verification aims at checking if a temporal logic formula φ is satisfied for the game-theoretic equilibria of a multi-agent system. This is in contrast with classical verification since here we are only interested in the *runs* which arise from agents behaving rationally. Each of the agent in the multi-agent system have their own goals which may or may not be conflicting with the goals of the other agents. We can understand the joint behaviour of these agents through game theoretic equilibria such as the Nash Equilibrium or the Subgame-Perfect Equilibrium. Let $P_1, \ldots, P_n$ denote the agents of a multi-agent system. We assume that the agents are non-deterministic reactive programs i.e. the agents are always active (never terminate) and at each time step have a number of choices available. A *strategy* for P_i is a mapping from the state of the system to an action. Thus, a strategy is basically an implementation of an agent such that its nondeterminism is resolved. Formally, we define a strategy to be a function which maps the history of the system (represented by a finite sequences of states) to the actions which the agent can perform at the current time step. We denote the set of possible strategies for agent P_i using $\Sigma(P_i)$. We define a *strategy profile* as a tuple of strategies $\bar{\sigma} = (\sigma_1, \ldots, \sigma_n)$ such that $\sigma_i \in \Sigma(P_i)$ for all $i \geqslant 0$. Since strategies ensure that the agents behave in deterministic manner, the strategy profile thus induces a *unique run* of

the system which is denoted by $\rho(\bar{\sigma})$. We define the set of all possible runs by

$$R(P_1, \ldots, P_n) = \{\rho(\bar{\sigma}) \mid \bar{\sigma} \in \Sigma(P_1) \times \ldots \times \Sigma(P_n).$$

3.1 Linear Temporal Logic

Linear Temporal Logic (**LTL**) was proposed by Pnueli. It extends propositional logic with **X** (*next*) and **U** (*until*) operators. An LTL formula is composed of the set of propositional variables *PV*, logical operators and operators **U**, **V**. The set of LTL formulae over the set of propositional variables *PV* can be inductively defined as follows:

- If $p \in PV$ it is an LTL formula.
- If φ and ψ are two LTL formulas then $\neg\psi$, $\varphi \vee \psi$, $\mathbf{X}\psi$, $\varphi\mathbf{U}\psi$ are all LTL formulae.

We can define other operators such as **G** (*always*) and **F** (*eventually*) using the fundamental operators **X** and **U** and the logical operators. The semantics of the temporal operators can be explained with the following examples.

- $\mathbf{X}\varphi$ means φ holds for the next time step.
- $\psi\mathbf{U}\varphi$ means that φ must eventually be true at some time step and ψ must hold for all time steps before that instant.
- $\mathbf{G}\varphi$ means φ holds for the current time step as well as all future time steps.
- $\mathbf{F}\varphi$ means φ holds for some time step starting from the current one.

3.2 Equilibrium Checking

An LTL formula is satisfied by some infinite word formed over alphabet consisting of sets of valuations of propositional variables where the *ith* symbol represents the valuation at the *ith* time step. We express the properties of runs using LTL and denote that the run ρ satisfies the temporal

property φ using the notation $\rho \models \varphi$. Since the agents want to satisfy their own specification, it is natural that they would prefer some runs over the others. We define the preference relation for each agent using $\succeq_i$. We write $\rho_1 \succeq_i \rho_2$ to imply that P_i prefers ρ_1 to ρ_2 and in this case $\rho_2 \models \varphi_i$ implies $\rho_1 \models \varphi_i$ where the goal of the agent P_i is represented by φ_i. We represent a multi-agent system as $M = (P_1, \ldots, P_n, \succeq_1, \ldots, \succeq_n)$. We say that for a multi-agent system $M = (P_1, \ldots, P_n, \succeq_1, \ldots, \succeq_n)$, a strategy profile $\bar{\sigma} = (\sigma_1, \ldots, \sigma_n)$ is a *Nash equilibrium* for M if for all deviating strategies $\sigma_i' \in \Sigma(P_i)$, $\rho(\bar{\sigma}) \succeq_i \rho(\sigma_1, \ldots, \sigma_i', \ldots, \sigma_n)$ holds true. We denote the set of all Nash equilibria of M as $NE(M)$. We define the following equilibrium checking problems where we ask of some or all runs that are induced by a strategy profile belonging to the set of Nash equilibria for M satisfy the temporal property φ:-

Non-Emptiness: Given a multi-agent system M, is $NE(M) \neq \varnothing$?

E-Nash: Given a multi-agent system M and a temporal formula φ, is it the case that there exists a $\bar{\sigma} \in NE(M)$ such that $\rho(\bar{\sigma}) \models \varphi$?

A-Nash: Given a multi-agent system M and a temporal formula φ, is it the case that $\rho(\bar{\sigma}) \models \varphi$ hold for all $\bar{\sigma} \in NE(M)$?

3.3 Infinite Games over Graphs

The synthesis problem here is that apart from checking the existence of a Nash equilibrium, we also want to generate a strategy profile such that it induces a run which satisfies the temporal logic specification. This is done by converting the problem to solving graph games. These graph games are infinite games which are played on finite directed graphs. Each of the agent is a player and controls a set of vertices in the graph and a token is passed in each turn from one vertex to another by the player who is under control of the vertex where the token is currently present. Since the objectives of players may or may not be opposed to one another we model the system as multi-player non-zero-sum games and we seek to find a Nash equilibrium such that the global

specification of the system is specified and each of the player/agent try to maximize their own objectives. We formally define two-player turn-based infinite game played on a finite directed graph which can be extended for more players.

An *arena* is defined as $A = (V, E, V_1, V_2)$ where V is the set of vertices and E is the set of edges in the finite directed graph, V_1 is the set of vertices that is under the control of player 1 and V_2 is the set of vertices under the control of the player 2. Each of the vertex in this graph has at least one successor i.e. outgoing edge to another vertex which ensures that the game goes on infinitely.

A *play* is defined as an infinite sequence of vertices defined as $\rho = \rho_0\rho_1...$ such that $\rho \in V^\omega$ and for all $i \geqslant 0$ $(\rho_i, \rho_{i+1}) \in E$. We define *strategy* to be a function $\sigma_i : V^*V_i \rightarrow V$ where $\sigma(wv) = v'$ implies $(v,v') \in E$ for $w \in V^*$ and $v \in V_i$. The strategy determines the next vertex to be visited based on the history of the vertices visited till now and the current vertex. For simplicity we generally consider memoryless strategies which only depend upon the current vertex and not on the past. We want to find a set of strategies i.e. the strategy profile such that it forms a Nash equilibrium where each of the players behave rationally in pursuit of their objective and the specification of the system is also satisfied by the run induced by this strategy profile. Some examples of objectives are safety objectives (which enforce that that the play ρ mustn't visit some unsafe vertices), buchi objectives (which enforce that the play ρ must visit some of the vertices infinitely often), reachability objectives (which enforce that the play ρ should visit some vertices atleast once) etc.

Chapter 4

Experimentation

To solve the distributed synthesis problem by the approaches described above, we tried out various tools that are currently available.

4.1 Agnes (Assume-Guarantee Synthesis Tool)

Agnes is an implementation of the assume-guarantee negotiation algorithm mentioned above constructed by Majumdar et al [2]. It is a header-only library written in C++. It is in a nascent stage, it accepts systems comprising of two components, and local specification of each component is restricted to safety and deterministic Büchi. Some of the examples which we tried out in Agnes are mentioned below.

4.1.1 Example 1

We first demonstrate a simple example. There are two components C_0 and C_1 along with two Boolean variables a and b handled by C_0 and C_1 respectively. C_0 has local specification $\phi_0 = aUb$, i.e., a should be *true* until b is *true*. C_1 has a local specification of *true*. If C_1 never sets b to *true* then, C_0 is unable to fulfill its local specification. The components are modeled as safety automata as shown in the figures below.

The states shown in red are $unsafe$ states. The edges are labeled as *input action/ external*

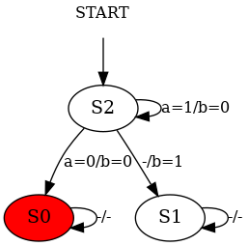

Fig. 4.1: C_0

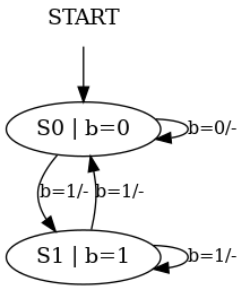

Fig. 4.2: C_1

variable. The label *dash* $'-'$, denotes the arbitrary input. We have set $S1$ as target state in C_0. If a becomes $false$ before b becomes true then, C_0 reaches an unsafe state. But if a remains true until b becomes *true* then, C_0 reaches the target state. For C_1 both states are set as target states and there is no unsafe state. Thus it is an arbitrary player without any goal. On running Agnes on it we get the following guarantees (in terms of safety automata) shown in the figures below.

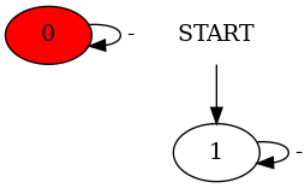

Fig. 4.3: G_0, Guarantee given by C_0

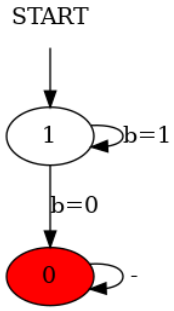

Fig. 4.4: G_1, Guarantee given by C_1

In guarantees the state 0 is always set to be the unsafe state. G_0 as we can see, is a guarantee permitting all behaviours, since there is no way to reach the unsafe state 0. G_1 forbids $b = 0$. Though this guarantee is not *maximally permissive*, but still it is *sufficient* for C_0 to fulfill its specification. We now see a slightly involved example.

4.1.2 Example 2

This is a slightly generalised setting of the distributed shared bus problem we discussed above. There are two components C_0 and C_1. C_0 has to send 1 packet within 3 time steps, whereas C_1 has

to send 2 packets within 4 time steps on a shared bus. The input for the components is given as a safety automaton. The figure shown below is the input given to Agnes for C_0(though the input is given in form of text, it is visualised here).

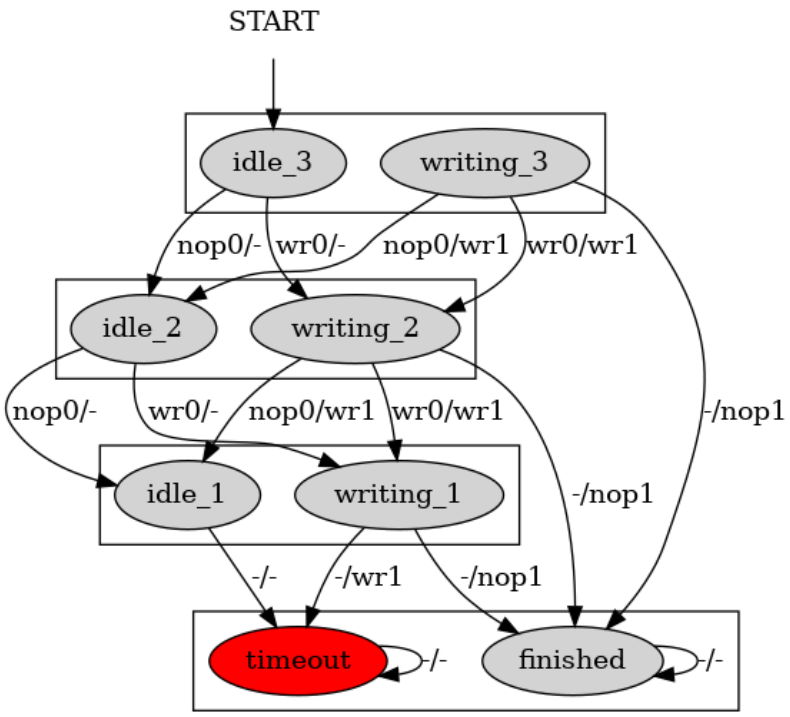

Fig. 4.5: Safety automaton for C_0

The edges are labeled as *input action/ external variable*. The label *dash* $'-'$, denotes the arbitrary input. The red state is the $unsafe$ state, which the component must not reach. Initial state is *idle*_3, denoting C_0 is not sending any packet and 3 time units are left. The transition from $idle7_3$ to $writing_2$ is labeled with $wr0/-$, i.e. if C_0 decides to take the input action of *wr* irrespective of the value of external variable received the component moves to *writing* state. Now the component successfully reaches $finished$ if external variable $nop1$ is seen. Similarly, the input for C_1 is given.

Now, the guarantees are output in format of safety automata with the convention that state 0 is unsafe. The output for guarantees are shown in the figures below.

Let us first understand the safety automaton in G_1. All runs of the safety automaton which do not visit the state 0 (the unsafe state shown in red) are valid runs. We see that all cases when

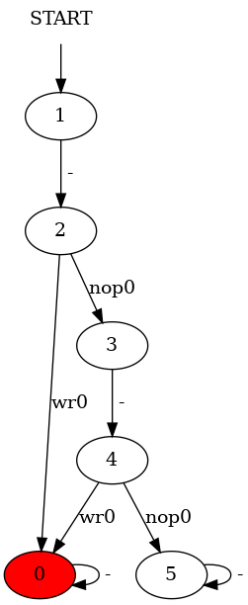

Fig. 4.6: G_0, Guarantee given by C_0

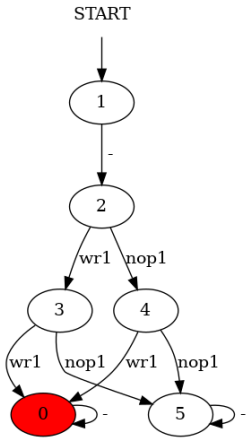

Fig. 4.7: G_1, Guarantee given by C_1

C_1 writes in the third time step are invalid. Thus C_1 provides a guarantee that it would not write in third time step. This is sufficient for C_0. Now similarly C_0 provides a guarantee that it would not write in time step second and fourth. This guarantee is also sufficient for C_1. Also, these two guarantees are compatible. Thus, we have a solution to the distributed synthesis problem.

4.2 EVE (Rational Verification Tool)

EVE (Equilibrium Verification Environment) [4] is a tool which takes game-theoretic approach to solve distributed synthesis. It assumes that components would act rationally in order to fulfill there specification. This way we do not have to ask for guarantees from other components. To give input to the tool, we need to encode the distributed system in *Simple Reactive Modules Language* (SRML). In EVE, apart from local specifications of the components, the system also has a global specification. We can then ask EVE the question, "Does there exist a strategy profile belonging to the Nash equilibrium between the components which satisfies the global specification?" We tested EVE for the answer to this question as well as the strategy for different examples.

4.2.1 Peer to Peer Network (2 clients)

This example has two components A and B which can remain in *upload* or *download* states. Component A can upload *ua* or download *da*. Similarly, component B can upload *ub* or download *db*. A can successfully download only if A is in download state and B is in upload state. Naturally, A wants to download often. Thus, specification for A is $G\,F(da\ and\ ub)$, i.e., A downloads successfully *infinitely often*. Vice versa for B. There is no overall global specification for system in this case that is all behaviours are accepted. EVE tells there exists some Nash equilibrium in which both components can satisfy there goals (EVE output shown in Fig. 4.8).

```
pbhaduri@devops:~/btp/eve-parity/eve-py/src$ python3 main.py e ../examples/p2p
Checking E-Nash property formula: true
Kripke states 4
Kripke edges 16
A  G  F  ( da and ub )
B  G  F  ( db and ua )
>>> YES, the property true is satisfied in some NE <<<
Winning Coalition ['A', 'B']
```

Fig. 4.8

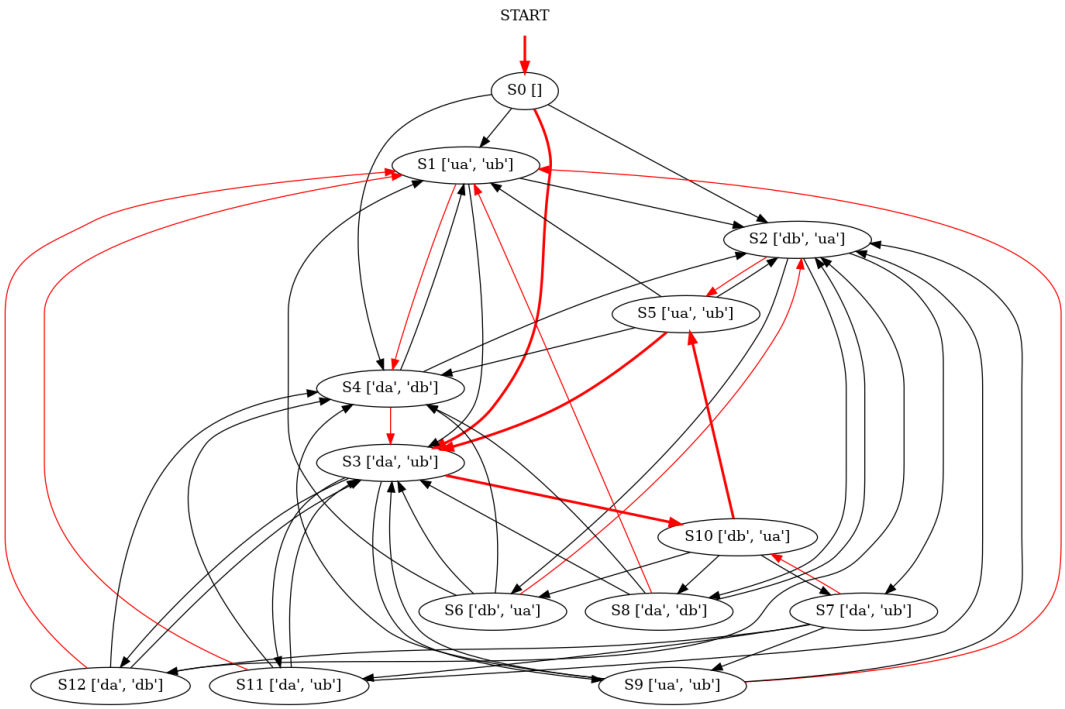

Fig. 4.9

EVE also outputs a graph containing all strategy profiles in the Nash equilibrium. The labels of node and transitions are mentioned in the text format. We have visualized the graph shown in Fig 4.9 for better understanding.

The graph contains all the strategy profiles which are Nash equilibrium. The strategy profiles given here are memoryless and fixing one transition for each node gives us one strategy profile. We have arbitrarily marked these transitions in *red* to show one example. This is a *winning* strategy for both the components. If both components follow their strategy, they would trace a path shown in *bold red* edges. Here, none of the components have an incentive to deviate from the given profile.

4.2.2 Matching Pennies

Matching Pennies example has two players Alice and Bob who can flip coins *ca* and *cb* respectively. *ca* and *cb* both take Boolean values. Alice wants that the coins should be matching *infinitely often*, i.e., $GF(ca \longleftrightarrow cb)$. Bob wants that the coins should be opposite *infinitely often*, i.e., $GF!(ca \longleftrightarrow cb)$. And the global specification is that whenever *ca* is true, in the next time step *cb* should be true, i.e., $G(ca \longrightarrow X\, cb)$. EVE gives the output, shown in figure below, that there exists some Nash equilibria satisfying the global specification, which is indeed true.

```
pbhaduri@devops:~/btp/eve-parity/eve-py/src$ python3 main.py e ../examples/matching_pennies_new
Checking E-Nash property formula:  G  ( ca ->  X cb )
Kripke states 4
Kripke edges 16
alice  G  F  ( ca <-> cb )
bob  G  F  !  ( ca <-> cb )
>>> YES, the property  G  ( ca ->  X cb )  is satisfied in some NE <<<
Winning Coalition ['alice', 'bob']
```

Fig. 4.10: Output given by EVE on running the example

We have visualized the output of graph below in Fig 4.11 for better understanding. We see that at every node where *ca* is true, all of the transitions go to a state with either $['cb']$ or $['ca','cb']$. Following one path in this graph gives us a strategy. All the strategy profiles given in the graph ensure the global specification, while maintaining the Nash equilibrium.

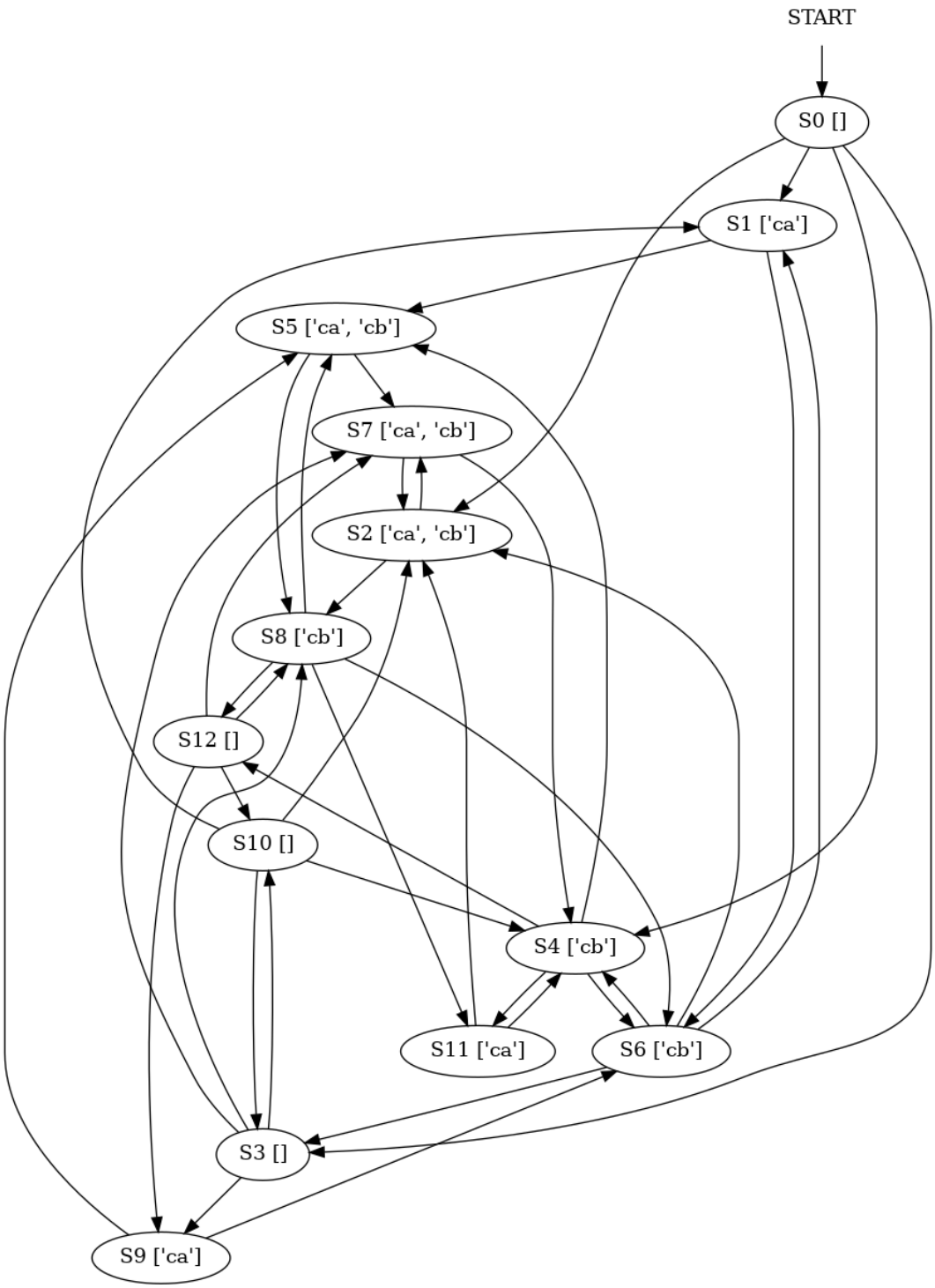
START
S0 []
S1 ['ca']
S5 ['ca', 'cb']
S7 ['ca', 'cb']
S2 ['ca', 'cb']
S8 ['cb']
S12 []
S10 []
S4 ['cb']
S11 ['ca']
S6 ['cb']
S3 []
S9 ['ca']

Fig. 4.11

Chapter 5

Conclusion and Future Work

In this semester we familiarized ourselves with the theory behind the two methods — Assume Guarantee Synthesis and Rational Verification and utilized the existing tools for these techniques on simple examples. As distributed synthesis is undecidable for any example, the termination of either method is not guaranteed however if it does terminate then we can find the strategies for the controllers.

Our plan for the next semester is to look at more complicated examples and check whether synthesizing controllers for them is feasible through either of these methods. We will also explore more tools for the distributed synthesis problem as per need. In particular we will be looking at examples from distributed systems such as the distributed mutual exclusion problem, consensus and agreement and more. In distributed mutual exclusion problem we are given concurrent processes which are required to execute the critical section and since in a distributed system shared variables can't be used for ensuring mutual exclusion, message passing has to be used. The algorithms which ensure distributed mutual exclusion must operate under partial knowledge of the system state. In the consensus problem, we have concurrent processes each of which propose a value and they need to agree on a single value. One of the approaches is to choose the value for which there is majority but there maybe faulty processes which can prevent this from happening and thus consensus is not reached. We will model such problems and try to solve them using the methods discussed.

Chapter 6

References

1. Alessandro Abate, Julian Gutierrez, Lewis Hammond, Paul Harrenstein, Marta Kwiatkowska, Muhammad Najib, Giuseppe Perelli, Thomas Steeples, and Michael Wooldridge. Rational verification: game-theoretic verification of multi-agent systems. Applied Intelligence, 51(9):6569–6584, 2021.

2. Rupak Majumdar, Kaushik Mallik, Anne-Kathrin Schmuck, and Damien Zufferey. Assume–guarantee distributed synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(11):3215–3226, 2020.

3. Amir Pneuli and Roni Rosner. Distributed reactive systems are hard to synthesize. In Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science, pages 746–757. IEEE, 1990.

4. Julian Gutierrez, Muhammad Najib, Giuseppe Perelli, and Michael Wooldridge. Eve: A tool for temporal equilibrium analysis. In International Symposium on Automated Technology for Verification and Analysis, pages 551–557. Springer, 2018.