Distributed Reactive Synthesis using Assume-Guarantee Reasoning and Games

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, communication protocols and mutual exclusion protocols. In our work we looked at two approaches - assume guarantee synthesis and rational verification for the synthesis problem and various tools such as EVE, Agnes, Praline, STV+AGR, MCMAS. We also explored the possibility of deriving distributed algorithms with the help of distributed synthesis.

<u>CERTIFICATE</u>

*This is to certify that the work contained in this thesis entitled "**Distributed Reactive Synthesis using Assume-Guarantee Reasoning and Games**" is a bonafide work of **Mesharya M Choudhary and Bhuvan Aggarwal (Roll No. 190101053, 190101025**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Prof. Purandar Bhaduri**

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 Goal

Our aim is to try out two methods - Rational Verification [1] and Assume Guarantee Synthesis [2] for distributed reactive synthesis and try to verify and synthesize distributed algorithms using either for which we explored a variety of tools. 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 that we are able to find assume-guarantee pairs A_i, G_i for each of the components such that they can satisfy their assume-guarantee contract (A_i, G_i) and the guarantees G_i imply that the assumptions A_i hold it can be shown that the local specification of the components ϕ_i 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 $v \upharpoonright_X$ and $v \upharpoonright_Y$ 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 $(v_0 \upharpoonright_Y)(v_1 \upharpoonright_Y) \ldots$ 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 $v_0 v_1 \dots$ is denoted using $proj_X(\rho)$, the sequence of external inputs $w_0 w_1 \dots$ using $proj_W(\rho)$, and the sequence of outputs $h(v_0)h(v_1)\dots$ using $proj_Y(\rho)$. Given a subset $V \subseteq W$ over the set of external variables the notation $proj_V(\rho)$ denotes the sequence $(w_0 \restriction V)(w_1 \restriction V)\dots$.

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 $X = X_1 \uplus X_2$, $x_{\text{in}} = x_{\text{in}1} \oplus x_{\text{in}2}$, $W = W_1 \cup W_2 \backslash (Y_1 \cup Y_2)$, $\delta : \mathcal{V}(X) \times \mathcal{V}(W) \times U_1 \times U_2 \to \mathcal{V}(X)$, 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

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

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 $v_0 \xrightarrow{w_0,u_0} v_1 \xrightarrow{w_1,u_1} v_2 \xrightarrow{w_2,u_2} \dots$ is said to *satisfy* the specification if $(v_0, w_0)(v_1, w_1), \dots \in \varphi$. 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 $\{(v_0, w_0), (v_1, w_1), \dots \in (\mathcal{V}(X) \times \mathcal{V}(W))^\omega \mid v_0 v_1 \dots\}$ 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. 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 from [2] 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
 1: Initialize $A_0 := Y_1^\omega, G_0 := X_0^\omega, A_1 := Y_0^\omega, G_1 := X_1^\omega$
 2: while true do
 3:     if $C_i$ realizes $\phi_i$ under $(A_i, G_i)$ for all $i \in \{0,1\}$ then
 4:         return $(A_0, C_0)$ and $(A_1, C_1)$
 5:     end if
 6:     if $C_i$ surely cannot realize $\phi_i$ under $(A_i, G_i)$ for any $i \in \{0,1\}$ then
 7:         return DoesNotExist
 8:     end if
 9:     $L_0 := FindAssumption(C_0, (A_0, G_0), \phi_0)$
10:     $A_0 := A_0 \cap L_0, G_1 := G_1 \cap h_1^{-1}(L_0)$
11:     $L_1 := FindAssumption(C_1, (A_1, G_1), \phi_1)$
12:     $A_1 := A_1 \cap L_1, G_0 := G_0 \cap h_0^{-1}(L_1)$
13: 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 $\geq_i$. We write $\rho_1 \geq_i \rho_2$ 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 $M = (P_1, \ldots, P_n, \geq_1, \ldots, \geq_n)$. We say that for a multi-agent system $M = (P_1, \ldots, P_n, \geq_1, \ldots, \geq_n)$, a strategy profile $\bar{\sigma} = (\sigma_1, \ldots, \sigma_n)$ is a *Nash equilibrium* for M if for all deviating strategies $\sigma'_i \in \Sigma(P_i)$, $\rho(\bar{\sigma}) \geq_i \rho(\sigma_1, \ldots, \sigma'_i, \ldots, \sigma_n)$ 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 $NE(M) \neq \emptyset$?

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

Tools

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 the tool on a simple example we came up with. 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*

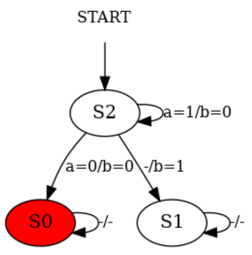


Fig. 4.1: C_0

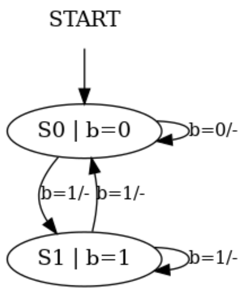


Fig. 4.2: C_1

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.

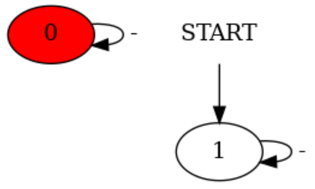


Fig. 4.3: G_0, Guarantee given by C_0

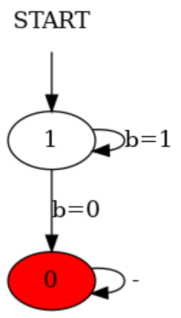


Fig. 4.4: G_1, Guarantee given by C_1

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 from [2] 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).

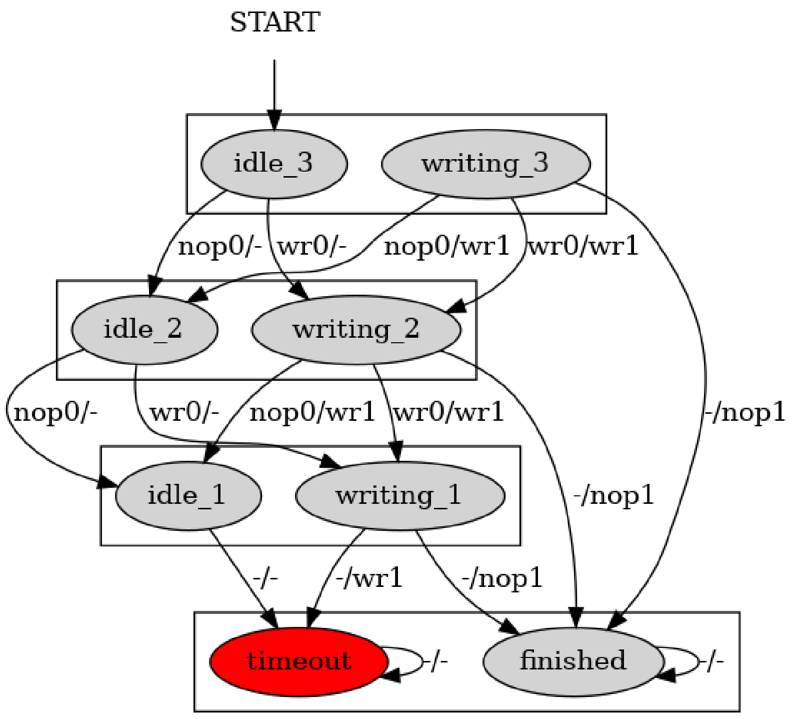


Fig. 4.5: Safety automaton for C_0

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

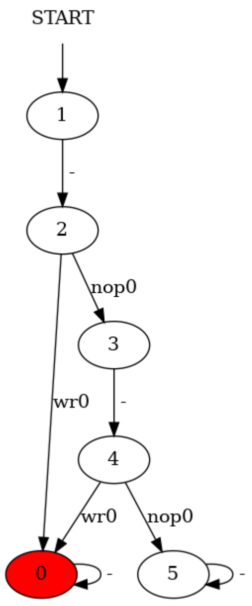


Fig. 4.6: G_0, Guarantee given by C_0

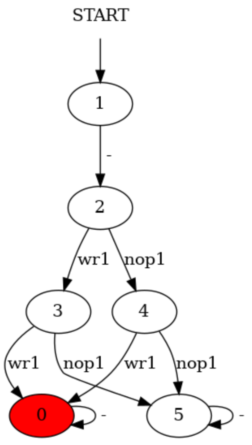


Fig. 4.7: G_1, Guarantee given by C_1

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.

4.1.3 Limitations

We encountered the following limitations with the tool.

1. The tool only supports two component systems currently.

2. It only supports safety and deterministic Buchi specifications as of now.

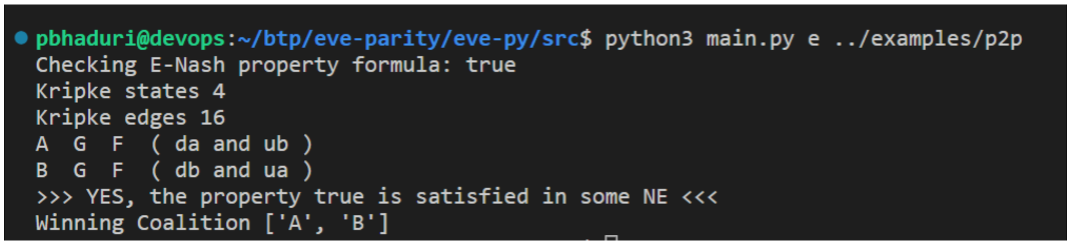
3. It accepts input in form of safety automaton which is specified by attributes such as number of states, number of inputs, list of initial states, list of safe states, transition matrix etc.This makes the input error prone and cumbersome compared to the other input formats for example SRML in case of EVE.

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)

We consider the Peer to Peer example which was provided as part of EVE. 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: Output given by EVE on running the example

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

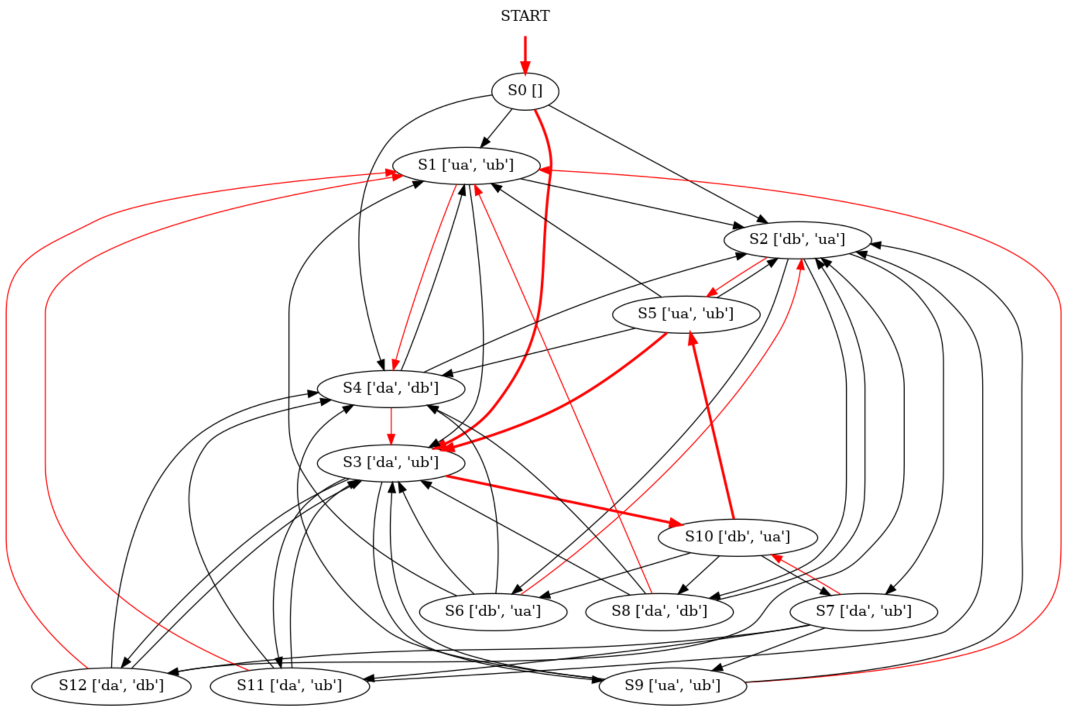


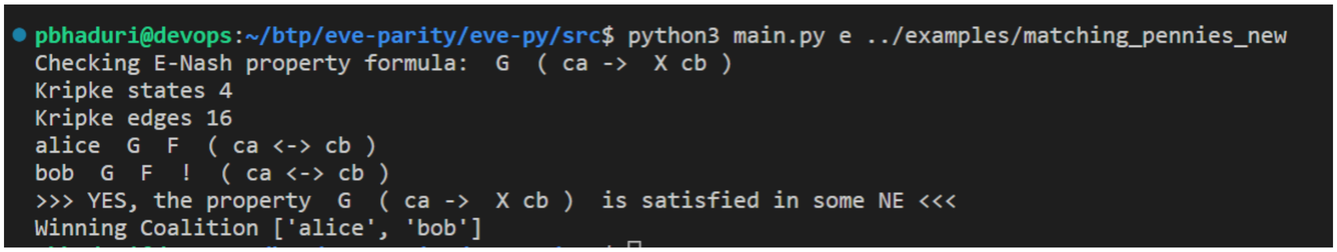
Fig. 4.9: Witness generated by EVE

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

We consider the Matching Pennies example which was provided with EVE. It 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.

4.2.3 Limitations

We encountered the following limitations with the tool.

1. The witness generated by EVE contains all Nash Equilibria but not all paths in the witness are Nash Equilibria thus one has to manually extract the strategy using the witness by ensuring that the chosen set of edges constitute a Nash Equilibrium.

4.3 PRALINE (Nash Equilibria Synthesis)

PRALINE [5] is a tool that computes Nash equilibria in games played over graphs. The model of the game which is considered by it is concurrent games where at each state the joint action tuple of the agents determines the next state. The preferences of each player p_i is specified using a payoff function $payoff_i$ which is a mapping from each state (characterized by the value of the variables in that state) to integers. The payoff for a run in the game is given as the limit superior of this function and the goal is to maximize it which is a generalization of buchi objective with payoff 0 or 1 assigned to each state.

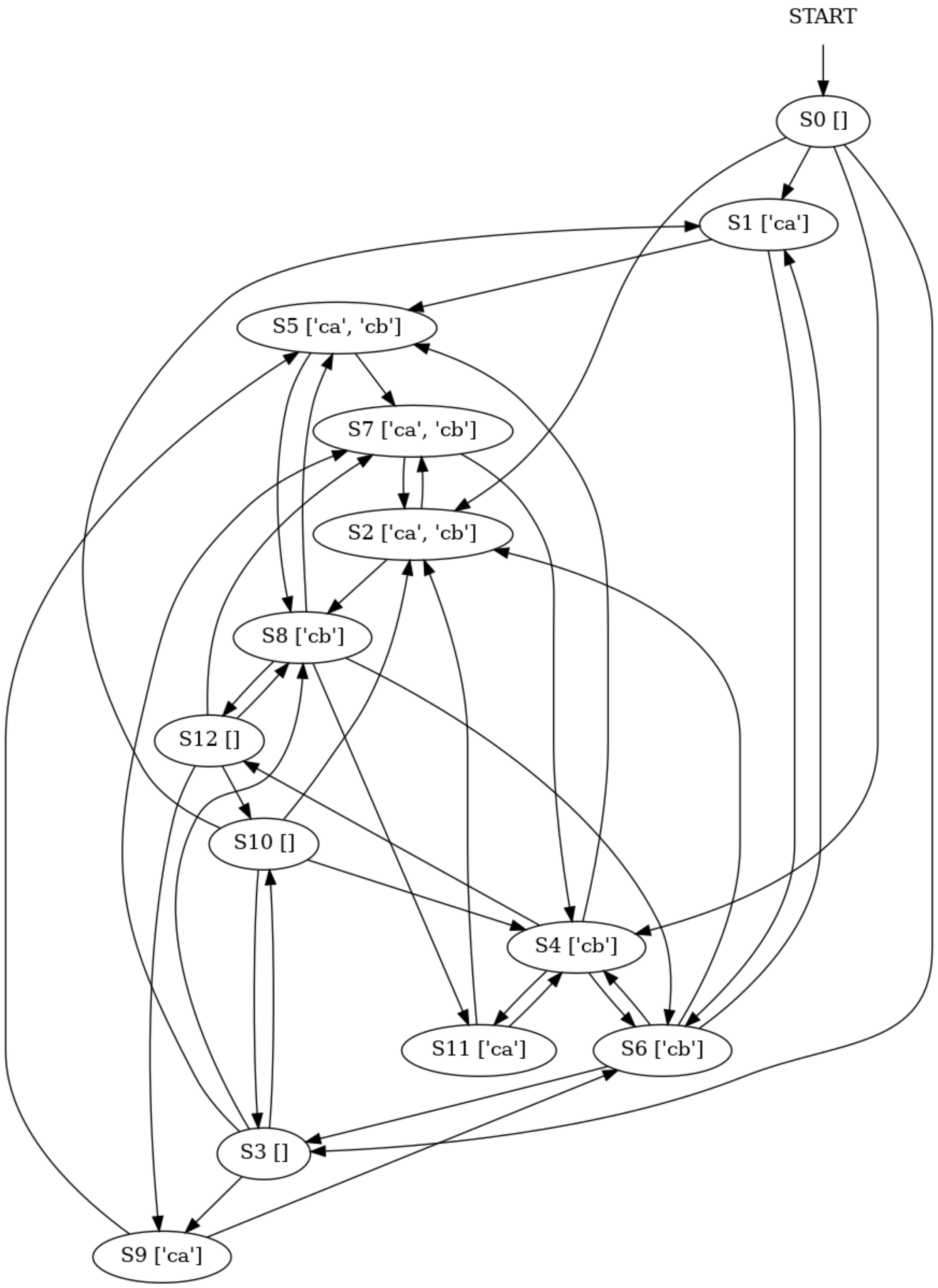
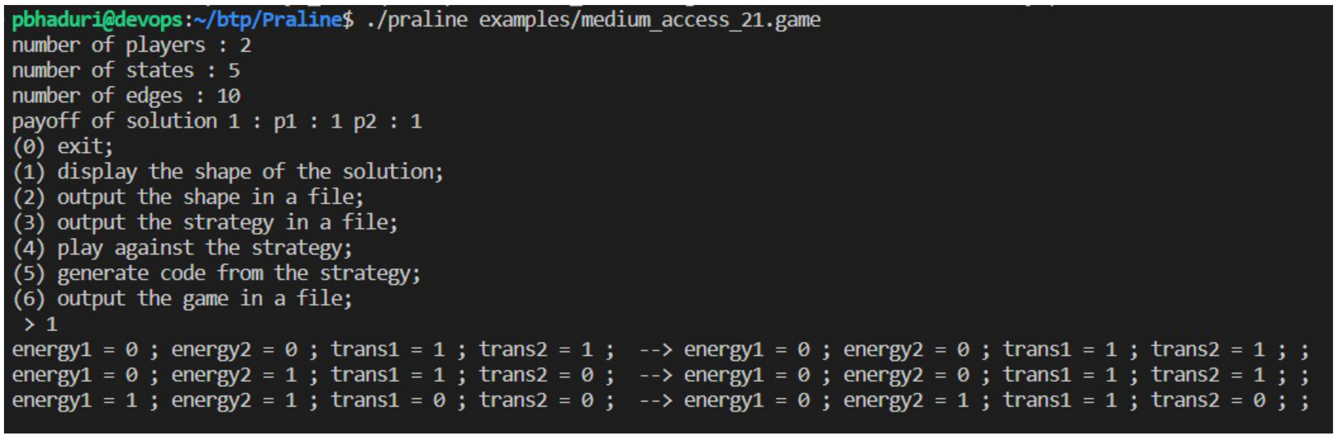


Fig. 4.11: Witness generated by EVE

4.3.1 Example 1

We consider the medium access example which was provided with the tool. It is similar to the distributed shared bus problem discussed for Agnes with some variation. There are two components C_0 and C_1 where both have to send a single packet with no deadline and the payoff function for each component is given by number of packets successfully sent i.e. maximum value can be 1. If both packets are sent at the same time they will be wasted and the payoff function will come out to be 0. The tool accepts the description of the game in text format with specified payoff functions, variables, actions and updates for given valuations of variables.



```
pbhaduri@devops:~/btp/Praline$ ./praline examples/medium_access_21.game
number of players : 2
number of states : 5
number of edges : 10
payoff of solution 1 : p1 : 1 p2 : 1
(0) exit;
(1) display the shape of the solution;
(2) output the shape in a file;
(3) output the strategy in a file;
(4) play against the strategy;
(5) generate code from the strategy;
(6) output the game in a file;
 > 1
energy1 = 0 ; energy2 = 0 ; trans1 = 1 ; trans2 = 1 ;  --> energy1 = 0 ; energy2 = 0 ; trans1 = 1 ; trans2 = 1 ; ;
energy1 = 0 ; energy2 = 1 ; trans1 = 1 ; trans2 = 0 ;  --> energy1 = 0 ; energy2 = 0 ; trans1 = 1 ; trans2 = 1 ; ;
energy1 = 1 ; energy2 = 1 ; trans1 = 0 ; trans2 = 0 ;  --> energy1 = 0 ; energy2 = 1 ; trans1 = 1 ; trans2 = 0 ; ;
```

Fig. 4.12: Output given by Praline on running the example

4.3.2 Limitations

We encountered the following limitations with the tool.

1. The tool doesn't accept LTL specification rather the payoff function itself is supposed to be provided in the form of mathematical expression and not every LTL specification may be expressed as a closed form expression.

4.4 STV+AGR (Strategy Synthesis and Verification using Assume Guarantee Reasoning)

STV+AGR [6] is a tool which allows model checking of relevant properties in asynchronous multi agent systems through assume guarantee reasoning. It takes model file as input in which we need to define the agents and the formula which we need to verify. Agents have two types of variables local and interface variables. Agents themselves read and write the local variables but can only read interface variables. Additionally, an agent is defined by states and transitions between them to depict actions available to the agent. After defining the agents, the formula to be verified needs to be expressed in Alternating-time Temporal Logic (ATL).

4.4.1 Alternating-time Temporal Logic (ATL)

ATL is an extension of CTL to multi agent systems. ATL provides constructs that allow reasoning about the actions and knowledge of agents in a multi-agent system. For example, $<< Agent_1 >> Fp$ is an ATL formula which states that $Agent_1$ has a strategy to ensure that property p holds sometime in the future. We can also specify multiple agents as a coalition, for example, $<< Agent_1, Agent_2 >> Fp$ states that $Agent_1$ and $Agent_2$ have a joint strategy to ensure that property p holds sometime in the future.

4.4.2 Example 1

We consider the example of distributed shared bus from [2], wherein the 2 components try to send a packet on distributed bus. If both components try to send their packets simultaneously, the packets are lost. In our implementation on STV+AGR we provide three chances to each components to send the packets. If any components is unable to send their packets in three chances then they are unable to reach the *sent* state by design. Thus our formula to be verified is $<< \{Sender1, Sender2\} >> F(Sender1\_state = sent \,\&\, Sender2\_state = sent)$.

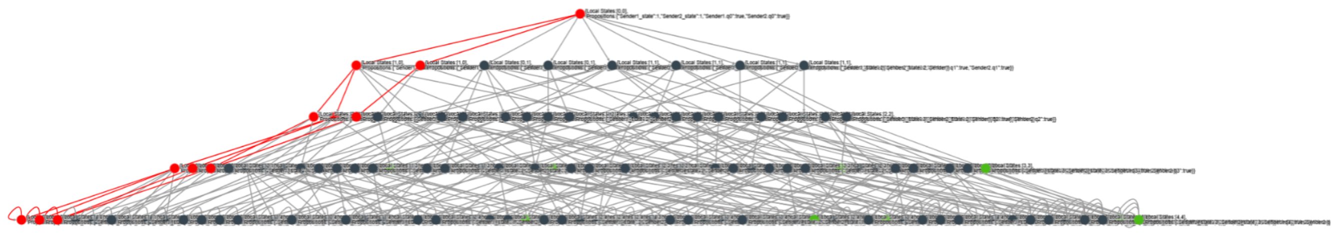


Fig. 4.13: Output given by STV+AGR on running distributed bus example

Upon running the tool we find that the formula holds true. This means $Sender1$ and $Sender2$ have a joint strategy to together reach the *sent* state. We also get graphical representation of the model as shown in Fig. 4.13 (we just show the scaled down figure since it contains large number of states and transitions). The strategy to make the formula hold is represented via red states and red transitions.

4.4.3 Limitations

We encountered the following limitations with the tool.

1. The tool sometimes throws error on some of the ATL formulas. For a concrete example, a formula like $<< Peer1 >> GF(Download2 = 1)$ throws 'Unknown Formula' error. Even if the formula $<< Peer1 >> FG(Download2 = 1)$ with G and F LTL operators interchanged works perfectly fine.

2. This tool is for asynchronous models only. We cannot model synchronous systems with this tool.

4.5 MCMAS (Model Checker for Verification of Multi Agent Systems)

MCMAS is a model checking tool which takes input MAS (Multi Agent System) in ISPL(Interpreted Systems Programming Language) format. It accepts LTL formulas and produces counterexamples for false formulae and witnesses (example over which the formula holds) for true formulae. In MCMAS each agent is characterized by a set of local states, a set of actions, a rule which describes which actions can be performed in which local states by the agent, an evolution function which describes how local states of agent changes based on the current local state and actions taken all the agents.

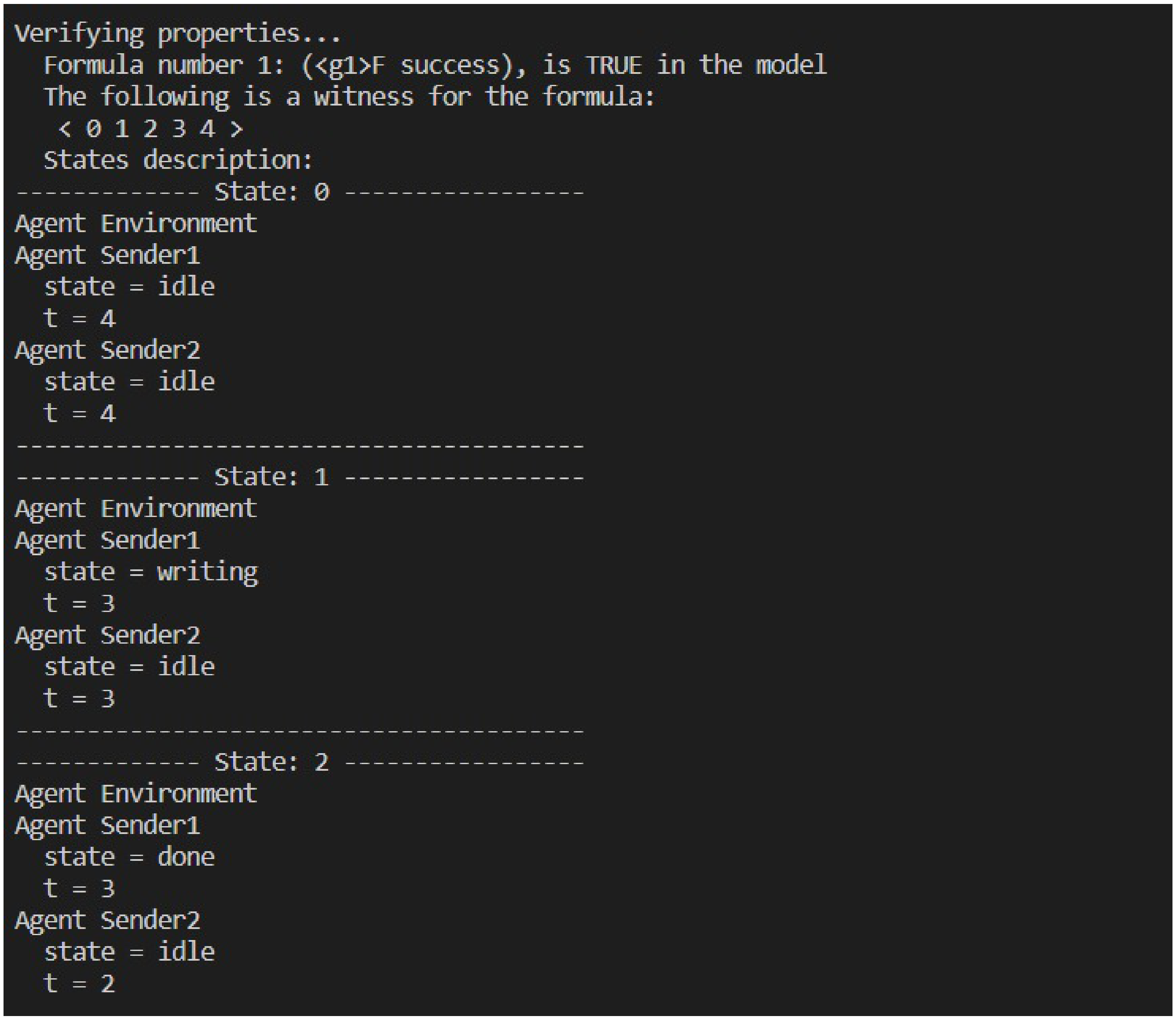
4.5.1 Example 1

We consider the distributed bus example from [2] where there are two senders and the aim of each is to send the packet within 4 time steps. We provide the specification as a text file and verify whether the property $<< Sender_1, Sender_2 >> F((Sender1.state = done)\&(Sender2.state = done))$ is satisfied where done denotes that both the senders have successfully sent the packet. The tool generates a witness as well where the specification is satisfied.

4.5.2 Limitations

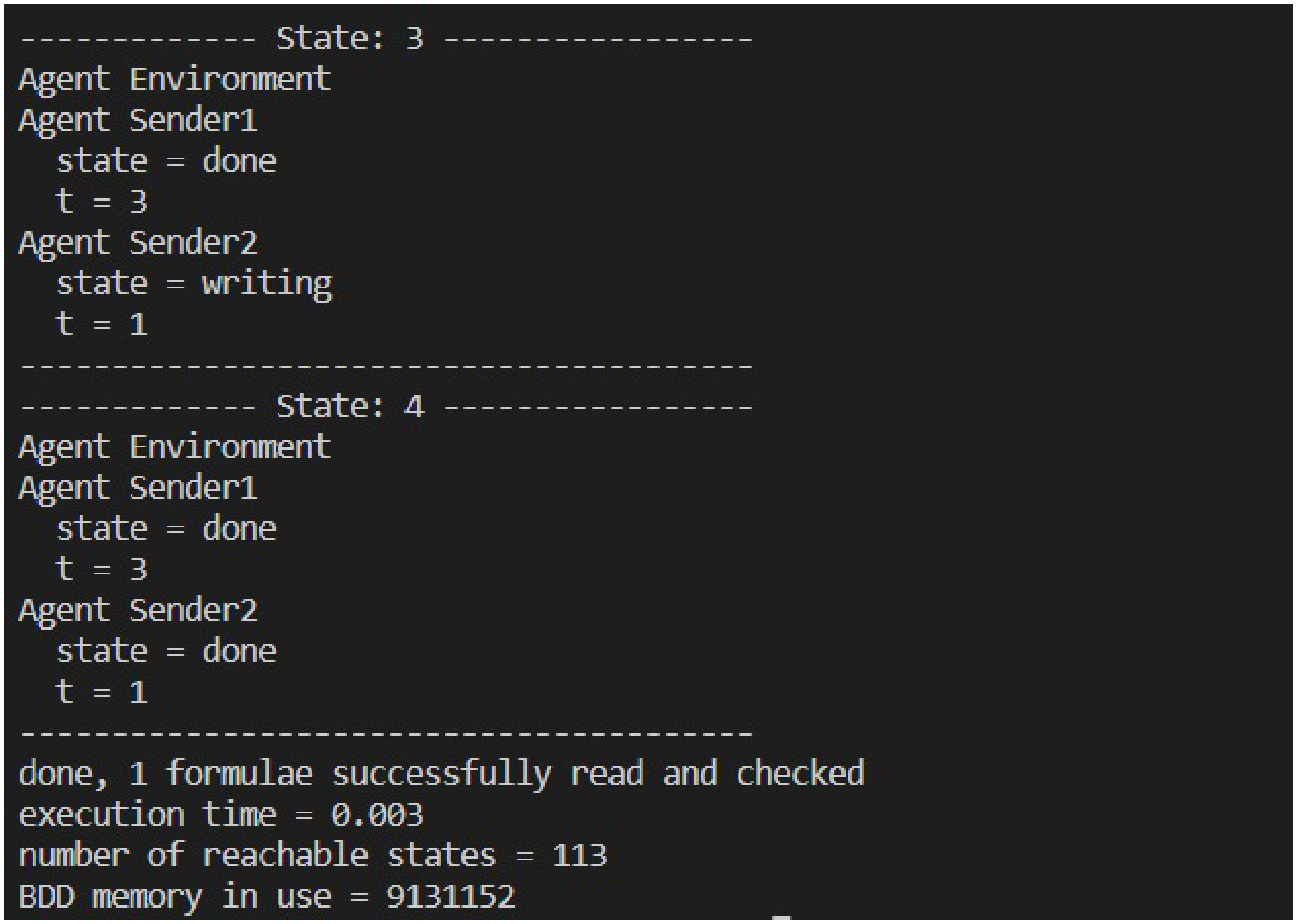
We encountered the following limitations with the tool.

1. The tool only produces a witness satisfying the formula but figuring out the winning strategy just from the witness isn't always possible. For cases where one fixes the strategy of the all other agents one can find how a given agent should behave by analyzing their witness as discussed in the Tian Ji example[7].



```
Verifying properties...
  Formula number 1: (<g1>F success), is TRUE in the model
  The following is a witness for the formula:
   < 0 1 2 3 4 >
  States description:
------------- State: 0 -----------------
Agent Environment
Agent Sender1
  state = idle
  t = 4
Agent Sender2
  state = idle
  t = 4
----------------------------------------
------------- State: 1 -----------------
Agent Environment
Agent Sender1
  state = writing
  t = 3
Agent Sender2
  state = idle
  t = 3
----------------------------------------
------------- State: 2 -----------------
Agent Environment
Agent Sender1
  state = done
  t = 3
Agent Sender2
  state = idle
  t = 2
```

Fig. 4.14: Output given by MCMAS on running the example



```
------------- State: 3 -----------------
Agent Environment
Agent Sender1
  state = done
  t = 3
Agent Sender2
  state = writing
  t = 1
----------------------------------------
------------- State: 4 -----------------
Agent Environment
Agent Sender1
  state = done
  t = 3
Agent Sender2
  state = done
  t = 1
----------------------------------------
done, 1 formulae successfully read and checked
execution time = 0.003
number of reachable states = 113
BDD memory in use = 9131152
```

Fig. 4.15: Output given by MCMAS on running the example

Chapter 5

Experiments

We decided to proceed with EVE for further experiments keeping in mind the shortcomings of the other tools that we tested.

5.1 Szymanski's Mutual Exclusion Algorithm

The algorithm is modeled using a waiting room with an entry and exit door. Each process requesting access to the critical section at nearly the same time enter the waiting room with the last one closing the entry door and opening the exit door. The processes then proceed to enter the critical section one by one with the last process closing the exit door and reopening the entry door while leaving the critical section. This process is repeated indefinitely. The implementation uses a **flag** variable for each process which is written by that process only and read by all others. The flag variable assumes 5 values:-

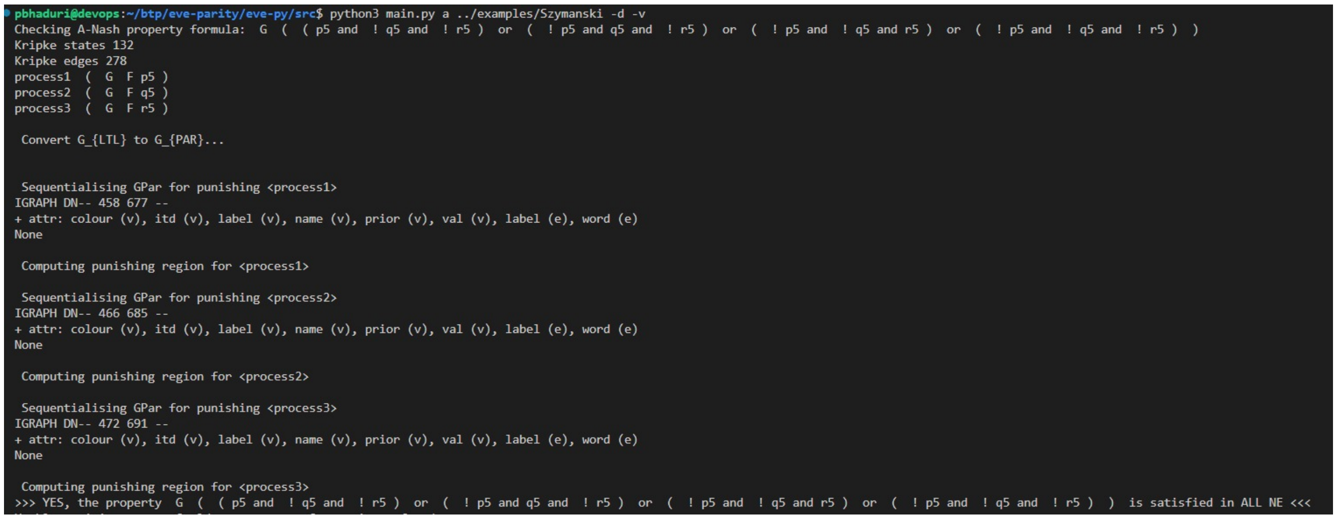
- **0** - indicates that the process is in noncritical section.
- **1** - indicates the intent of the process to enter the critical section.
- **2** - indicates that the process waits for the other processes to enter the waiting room via the entry door.
- **3** - indicates that the process has just entered the waiting room.

- **4** - indicates that the process has entered the critical section via the exit door

Algorithm 2 Szymanski's Mutual Exclusion Algorithm

```
1: $flag[i] := 0$
2: **while** true **do**
3:     $non - critical\ section$
4:     $flag[i] := 1$
5:     **await** $\forall j, flag[j] < 3$
6:     $flag[i] := 3$
7:     **if** $\exists j, flag[j] := 1$ **then**
8:         $flag[i] := 2$
9:         **await** $\exists j, flag[j] := 4$
10:    **end if**
11:    $flag[i] := 4$
12:    **await** $\forall j < i, flag[j] < 2$
13:    $critical\ section$
14:    **await** $\forall j > i, flag[j] < 2\ or\ flag[j] > 3$
15:    $flag[i] := 0$
16: **end while**
```

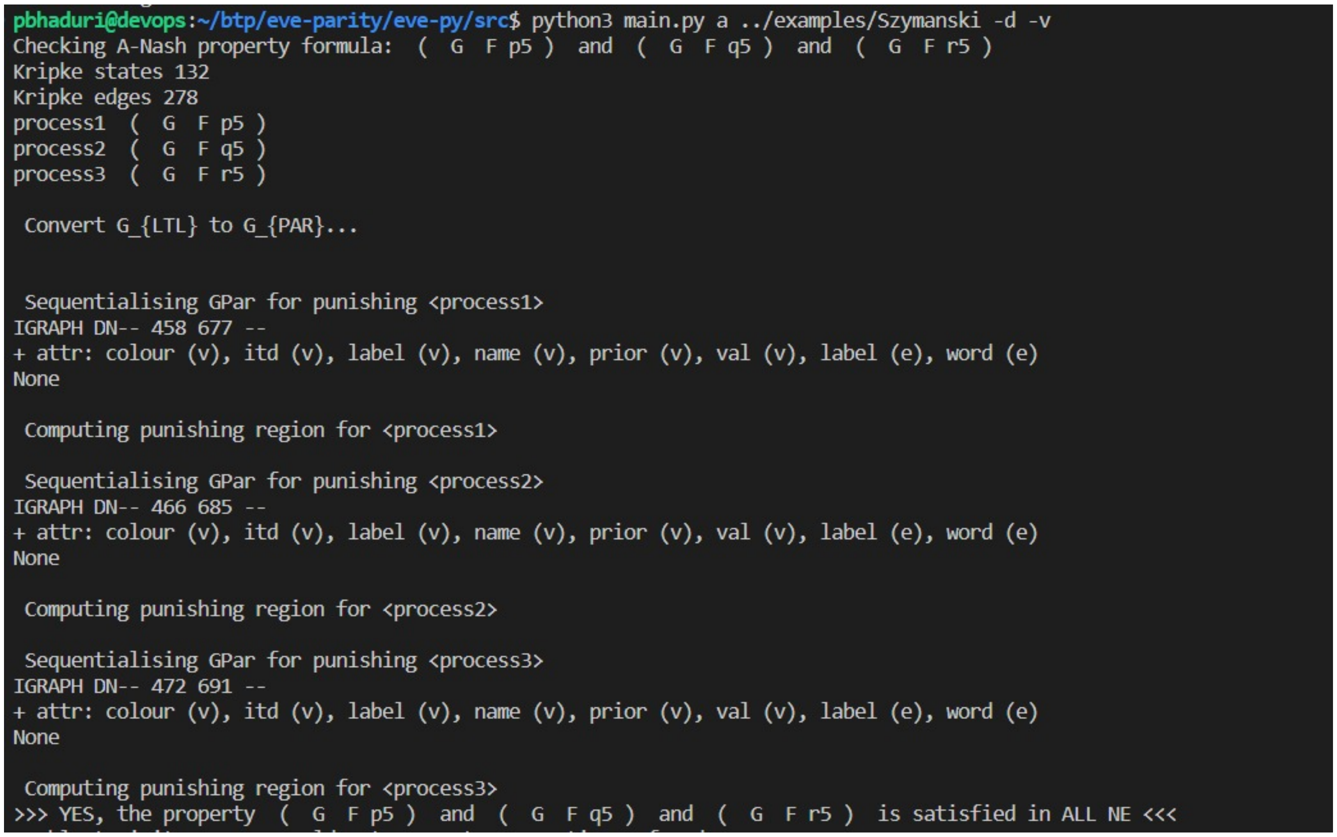
To verify mutual exclusion we set the goal of each process to be ensuring that the critical section is accessed infinitely often and the property of the system to be verified to be mutual exclusion. We find that mutual exclusion holds in all Nash equilibria. Thus Szymanski's Algorithm satisfies mutual exclusion.



```
pbhaduri@devops:~/btp/eve-parity/eve-py/src$ python3 main.py a ../examples/Szymanski -d -v
 Checking A-Nash property formula:  G  (  ( p5 and  ! q5 and  ! r5 )  or  (  ! p5 and q5 and  ! r5 )  or  (  ! p5 and  ! q5 and r5 )  or  (  ! p5 and  ! q5 and  ! r5 )  )
 Kripke states 132
 Kripke edges 278
 process1  (  G  F p5 )
 process2  (  G  F q5 )
 process3  (  G  F r5 )

  Convert G_{LTL} to G_{PAR}...


  Sequentialising GPar for punishing <process1>
 IGRAPH DN-- 458 677 --
 + attr: colour (v), itd (v), label (v), name (v), prior (v), val (v), label (e), word (e)
 None

  Computing punishing region for <process1>

  Sequentialising GPar for punishing <process2>
 IGRAPH DN-- 466 685 --
 + attr: colour (v), itd (v), label (v), name (v), prior (v), val (v), label (e), word (e)
 None

  Computing punishing region for <process2>

  Sequentialising GPar for punishing <process3>
 IGRAPH DN-- 472 691 --
 + attr: colour (v), itd (v), label (v), name (v), prior (v), val (v), label (e), word (e)
 None

  Computing punishing region for <process3>
 >>> YES, the property  G  (  ( p5 and  ! q5 and  ! r5 )  or  (  ! p5 and q5 and  ! r5 )  or  (  ! p5 and  ! q5 and r5 )  or  (  ! p5 and  ! q5 and  ! r5 )  )  is satisfied in ALL NE <<<
```

Fig. 5.1: Verifying Mutual Exclusion

To verify liveness we set the goal of each process to be ensuring that the critical section is accessed infinitely often and the property of the system to be verified as the condition that each of the process accesses the critical section infinitely often. We find that liveness holds in all nash equilibria. Thus Szymanski's Algorithm satisfies liveness.



```
pbhaduri@devops:~/btp/eve-parity/eve-py/src$ python3 main.py a ../examples/Szymanski -d -v
Checking A-Nash property formula:  (  G  F p5 )  and  (  G  F q5 )  and  (  G  F r5 )
Kripke states 132
Kripke edges 278
process1  (  G  F p5 )
process2  (  G  F q5 )
process3  (  G  F r5 )

 Convert G_{LTL} to G_{PAR}...


 Sequentialising GPar for punishing <process1>
IGRAPH DN-- 458 677 --
+ attr: colour (v), itd (v), label (v), name (v), prior (v), val (v), label (e), word (e)
None

 Computing punishing region for <process1>

 Sequentialising GPar for punishing <process2>
IGRAPH DN-- 466 685 --
+ attr: colour (v), itd (v), label (v), name (v), prior (v), val (v), label (e), word (e)
None

 Computing punishing region for <process2>

 Sequentialising GPar for punishing <process3>
IGRAPH DN-- 472 691 --
+ attr: colour (v), itd (v), label (v), name (v), prior (v), val (v), label (e), word (e)
None

 Computing punishing region for <process3>
>>> YES, the property  (  G  F p5 )  and  (  G  F q5 )  and  (  G  F r5 )  is satisfied in ALL NE <<<
```

Fig. 5.2: Verifying Liveness

5.2 Simpson's 4-slot Algorithm

Simpson's 4-slot algorithm allows a single writer and reader process to access shared memory without interference between concurrent writes and reads. The reader always accesses the most recently written data and neither process has to wait for the other. Thus it is a means to implement wait-free atomic register. The control values are assumed to be stored in safe registers i.e. registers which return the most recently written value when read and write are not concurrent and otherwise return arbitrary value. The algorithm considers 4-slots to which reads and writes are performed, arranged in two pairs *left* and *right* and within each pair the two slots are referred as *top* and *bottom*. Apart from this, control bits *latest* (denoting the most recently written pair), *index* (denoting slot of the pair which was written to most recently) are used, both of which are controlled by the writer. For reader the control bit *reading* indicates which pair is currently being read.

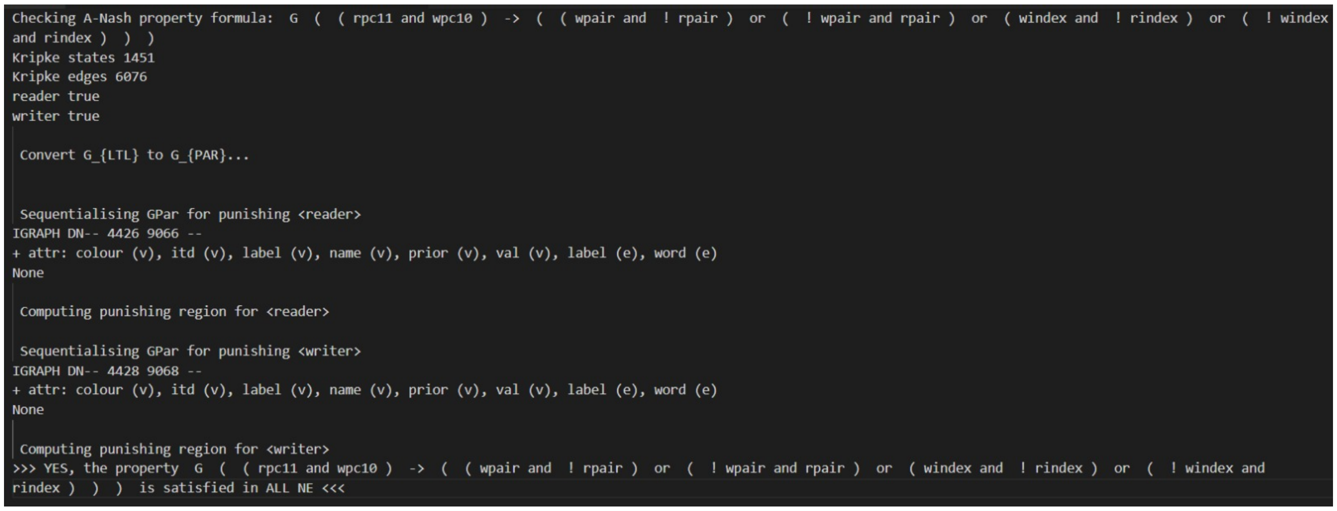
Algorithm 3 Writer Process

```
1: INPUT: $in$
2: VAR: $wpair, windex, index, slot, latest$
3: $wpair := !reading$
4: $windex := !index[wpair]$
5: $slot[wpair][windex] := in$
6: $index[wpair] := windex$
7: $latest := wpair$
```

Algorithm 4 Reader Process

```
1: OUTPUT: $out$
2: VAR: $rpair, rindex, reading$
3: $rpair := latest$
4: $reading := rpair$
5: $rindex := index[rpair]$
6: $out := slot[rpair][rindex]$
```

In our specification we represent the execution of each statement of the writer using state variables $wpc00, wpc01, wpc10, wpc11, wpc100$ and each statement of the reader using state variables $rpc00, rpc01, rpc10, rpc11$. Say the statement-3 (from the figure) of the writer is currently being executed, then currently $wpc00$ must be true and the transition should be to the next state where $wpc01$ will be true and the value of $wpair$ will be set to $!reading$. To model the fact that the registers in which control variables are stored are safe, we make sure that if simultaneous read/write is performed to the same variable, the value being read could be arbitrary. An example of this is when statement 3 of writer and statement 4 of reader are executing simultaneously. To verify that mutual exclusion happens, we keep the property as, 'whenever statement 5 of writer and statement 6 of reader execute simultaneously ($rpc11$ and $wpc10$ are true) the variables $wpair, windex$ and $rpair, rindex$ must not be equal'. We find the mutual exclusion property holds in all Nash equilibria.



```
Checking A-Nash property formula:  G  (  ( rpc11 and wpc10 )  ->  (  ( wpair and  ! rpair )  or  (  ! wpair and rpair )  or  ( windex and  ! rindex )  or  (  ! windex
and rindex )  )  )
Kripke states 1451
Kripke edges 6076
reader true
writer true

 Convert G_{LTL} to G_{PAR}...


 Sequentialising GPar for punishing <reader>
IGRAPH DN-- 4426 9066 --
+ attr: colour (v), itd (v), label (v), name (v), prior (v), val (v), label (e), word (e)
None

 Computing punishing region for <reader>

 Sequentialising GPar for punishing <writer>
IGRAPH DN-- 4428 9068 --
+ attr: colour (v), itd (v), label (v), name (v), prior (v), val (v), label (e), word (e)
None

 Computing punishing region for <writer>
>>> YES, the property  G  (  ( rpc11 and wpc10 )  ->  (  ( wpair and  ! rpair )  or  (  ! wpair and rpair )  or  ( windex and  ! rindex )  or  (  ! windex and
rindex )  )  )  is satisfied in ALL NE <<<
```

Fig. 5.3: Verifying Mutual Exclusion

5.3 Mutual Exclusion and Peterson's Algorithm

Mutual exclusion is the problem of ensuring that only one process can exist in critical section at any given point of time. Here we consider two processes and implement models of varying complexity to study the problem.

First we model a system with only $cs0$ and $cs1$ boolean variables controlled by Process 0 and Process 1 respectively. $cs0 == true$ denotes that Process 0 is in critical section. Goal of Process 0 is to infinitely often go into its critical section and always follow mutual exclusion. In LTL, this can be described as $(GF\ cs0)\ and\ G!(cs0\ and\ cs1)$. Same follows for Process 1. This is a memoryless system because there are no extra variables apart from $cs0$ and $cs1$. Upon running the implementation on EVE tool we get the witness graph containing all Nash Equilibrium runs. In the graph, we see a path where the systems alternates between $cs0$ and $cs1$ which also fulfils the goal of the processes. The witness graph generated by the following implementation is shown in Fig 5.4. The path is shown in red color. This path suggests round-robin type of algorithm. But we do not know yet how to individually implement this strategy. We need memory to implement round-robin algorithm. This brings us to our next implementation.

We now introduce memory in the form of variables. We add a shared variable *turn* to our implementation. EVE does not allow shared variables. So, to simulate the effect of shared variable

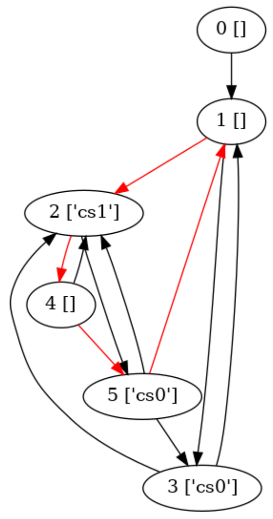


Fig. 5.4: Witness graph for mutual exclusion with a Nash Equilibrium path

we keep to variables *turn*0 and *turn*1 out of which one and only one is true at a time. The goals of the processes remain the same and *turn* variables are allowed all possible behaviours at each step. Upon generating a witness graph we see the graph contains a path where turn variable is set before the process goes into its critical section. The path is highlighted in red color in Fig. 5.5.

We can implement a round-robin algorithm with only a single *turn* variable. But in the scenario, where Process 0 has the turn but does not want to go into the critical section, Process 1 would keep on waiting. Peterson's algorithm solves this problem.

We now fully implement skeleton structure of the Peterson's algorithm. Each process can be in 5 different states $Q0$, $Q1$, $Q2$, $Q3$, $Q4$ which are formed according to the statements in Peterson's algorithm. The states along with the Peterson's algorithm are shown in the Fig. 5.6. Notice that $Q0$, $Q1$, $Q2$ are the states in which a variable is assigned some value. Assuming that we know which variable is to be assigned in each state, we try out different combinations of value assignments including the correct one's mentioned in the Peterson's algorithm itself. We find that only the value assignment which is in accordance with Peterson algorithm satisfies mutual exclusion, others fail. This way we can derive the statements $Q0$, $Q1$, $Q2$ using the tool.

We also set the tool to allow all possible transitions at the states $Q0$, $Q1$, $Q2$. With appropriate goals of the processes, we may be able to derive the assignments directly from the witness graph,

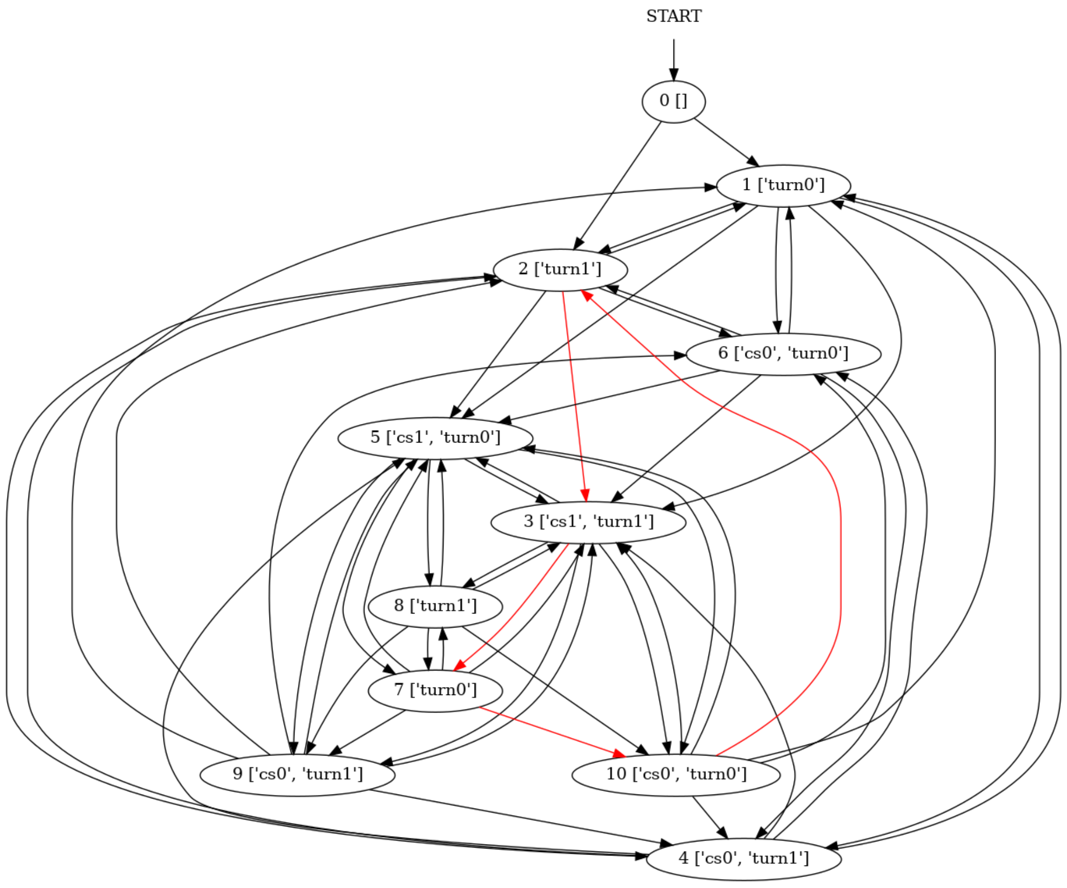
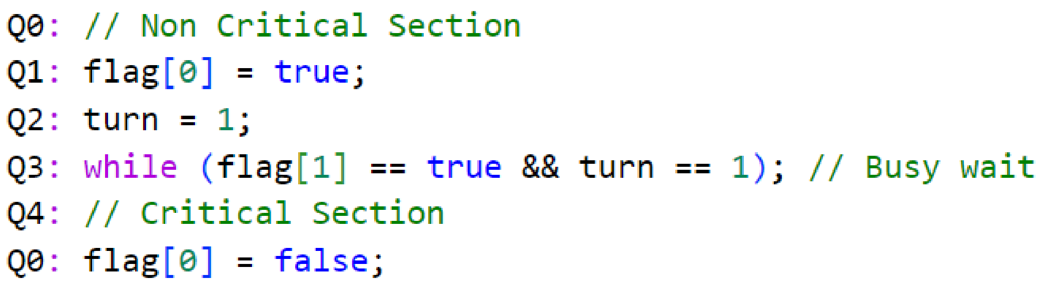


Fig. 5.5: Witness graph for mutual exclusion system with memory



```
Q0: // Non Critical Section
Q1: flag[0] = true;
Q2: turn = 1;
Q3: while (flag[1] == true && turn == 1); // Busy wait
Q4: // Critical Section
Q0: flag[0] = false;
```

Fig. 5.6: Peterson's Algorithm with the states of the process denoted by Q_is

but it contains 362 states and 1219 edges making it difficult to interpret by hand.

Chapter 6

Conclusion and Future Work

In this paper, we looked at two methods Rational Verification and Assume Guarantee Synthesis, as well as various tools for solving synthesis and verification problem for multi agent systems. For each tool, we worked with examples from simple to higher difficulty in order to understand and establish its correctness. First, we started working with EVE tool. It has an intuitive input and output format. Goals are represented in LTL format. But we learnt that the witness graph returned by EVE contains all Nash equilibrium runs, but not all paths in the witness are Nash equilibrium runs. Thus extracting the strategy out of the graph requires manual work. Then we tried out Agnes tool which currently supports only two components and accepts the goals as a safety automaton whose states and transitions are represented in multi-dimensional lists. This makes the input error prone. Also, it gives out assumptions and guarantees for both components but not the complete strategy, thus we moved on to PRALINE tool. PRALINE accepted goals of players as a mathematical expression whose limit superior is to be maximised. But several LTL goals cannot be represented in this format, so we were unable to proceed with the tool. Then we tried out STV+AGR which threw some internal errors on some of the simple ATL formulae. Also, it models asynchronous systems thus we were unable to properly model examples such as distributed shared bus. Finally, we worked on MCMAS which was a model checking tool and didn't produce the complete strategy but rather produced only an example satisfying the specification if it holds.

After evaluating all of the tools, we proceeded with EVE tool. Each of them had some shortcomings, but in EVE, we could represent goals in LTL, it tells the winning coalition according to Nash equilibrium and generates a witness containing all the Nash equilibria. LTL synthesis is 2EXPTIME-complete [8]. Owing to this fact, at even slightly complex examples EVE took several hours and didn't even terminate at several instances. We ran our tools on an Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz machine with 16 GB RAM. We verified the properties of Szymanski's Mutual Exclusion algorithm as well as the Simpson's 4-slot algorithm. We took the mutual exclusion problem and looked at how we can use the tool to derive the Peterson's algorithm in some sense.

The problem of deriving algorithms in a human comprehensible format via strategy profile obtained from distributed synthesis is difficult because it requires conversion of automata to algorithm. For example, as discussed in the Peterson's Algorithm example it is tough to determine the variables and statements just by looking at the automata and one must also know the variables required beforehand to determine how they are assigned. We weren't able to find any tool which generates the the set of all winning strategies as well as accepts LTL goals, but in the future if such a tool is created then the automata generated may be directly used by the computer. Although the automata containing the set of all winning strategies may not be convenient for humans but a computer may be able to utilize it to infer the algorithm. An alternative for this is to utilize verification and try out the possible combinations of statements and find the correct algorithm satisfying the specification via brute force which we have discussed in the Peterson's Algorithm example.

Chapter 7

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.

5. Brenguier, R. (2013). PRALINE: A Tool for Computing Nash Equilibria in Concurrent Games. In: Sharygina, N., Veith, H. (eds) Computer Aided Verification. CAV 2013. Lecture

Notes in Computer Science, vol 8044. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-39799-8_63

6. Damian Kurpiewski, Łukasz Mikulski, Wojciech Jamroga. STV+AGR: Towards Practical Verification of Strategic Ability Using Assume-Guarantee Reasoning. arXiv:2203.01033 [cs.MA]

7. Lomuscio, A., Qu, H. Raimondi, F. MCMAS: an open-source model checker for the verification of multi-agent systems. Int J Softw Tools Technol Transfer 19, 9–30 (2017). https://doi.org/10.1007/s10009-015-0378-x

8. Pnueli A, Rosner R (1989) On the synthesis of an asynchronous reactive module. In: Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programs